Fast DDS Documentation

Release 2.12.2

eProsima

Mar 22, 2024

INTRODUCTION

1 Fast DDS-Gen **3**

2 RTPS Wire Protocol **5**

3 Main Features **7**

4 Contacts and Commercial support **9**

5 Contributing to the documentation **11**

6 Structure of the documentation **13**

6.1 DDS API . . . 13
6.2 Fast DDS-Gen . . . 14
6.3 RTPS Wire Protocol . . . 14
6.4 Main Features . . . 14
6.5 Contacts and Commercial support . . . 16
6.6 Contributing to the documentation . . . 16
6.7 Structure of the documentation . . . 16
6.8 Linux installation from binaries . . . 16
6.9 Windows installation from binaries . . . 18
6.10 Linux installation from sources . . . 20
6.11 Windows installation from sources . . . 30
6.12 Mac OS installation from sources . . . 40
6.13 QNX 7.1 installation from sources . . . 46
6.14 CMake options . . . 51
6.15 Getting Started . . . 55
6.16 Library Overview . . . 91
6.17 DDS Layer . . . 96
6.18 RTPS Layer . . . 260
6.19 Discovery . . . 268
6.20 Transport Layer . . . 299
6.21 Persistence Service . . . 340
6.22 Security . . . 345
6.23 Logging . . . 369
6.24 Statistics Module . . . 378
6.25 XML profiles . . . 390
6.26 Environment variables . . . 459
6.27 PropertyPolicyQos Options . . . 464
6.28 Dynamic Topic Types . . . 477
6.29 Typical Use-Cases . . . 497
6.30 ROS 2 using Fast DDS middleware . . . 560

6.31 Troubleshooting . . . 574
6.32 C++ API Reference . . . 575
6.33 Python API Reference . . . 862
6.34 Introduction . . . 1000
6.35 Usage . . . 1001
6.36 Building a publish/subscribe application . . . 1002
6.37 Building Python auxiliary libraries . . . 1007
6.38 Defining a data type via IDL . . . 1008
6.39 CLI . . . 1019
6.40 Docker Images . . . 1027
6.41 Version 2.12.2 . . . 1033
6.42 Previous versions . . . 1034

Index **1085**

eProsima Fast DDS is a C++ implementation of the DDS (Data Distribution Service) Specification, a protocol defined by the Object Management Group (OMG). The *eProsima Fast DDS* library provides both an Application Programming Interface (API) and a communication protocol that deploy a Data-Centric Publisher-Subscriber (DCPS) model, with the purpose of establishing efficient and reliable information distribution among Real-Time Systems. *eProsima Fast DDS* is predictable, scalable, flexible, and efficient in resource handling. For meeting these requirements, it makes use of typed interfaces and hinges on a many-to-many distributed network paradigm that neatly allows separation of the publisher and subscriber sides of the communication. *eProsima Fast DDS* comprises:

1. The *DDS API* implementation.
2. *Fast DDS-Gen*, a generation tool for bridging typed interfaces with the middleware implementation.
3. The underlying *RTPS* wire protocol implementation.

For all the above, *eProsima Fast DDS* has been chosen as the default middleware supported by the Robot Operating System 2 (ROS 2) in every long term (LTS) releases and most of the non-LTS releases.

The communication model adopted by DDS is a many-to-many unidirectional data exchange where the applications that produce the data publish it to the local caches of subscribers belonging to applications that consume the data. The information flow is regulated by Quality of Service (QoS) policies established between the entities in charge of the data exchange.

As a data-centric model, DDS builds on the concept of a "global data space" accessible to all interested applications. Applications that want to contribute information declare their intent to become publishers, whereas applications that want to access portions of the data space declare their intent to become subscribers. Each time a publisher posts new data into this space, the middleware propagates the information to all interested subscribers.

The communication happens across domains, i. e. isolated abstract planes that link all the distributed applications able to communicate with each other. Only entities belonging to a same domain can interact, and the matching between entities subscribing to data and entities publishing them is mediated by topics. Topics are unambiguous identifiers that associate a name, which is unique in the domain, to a data type and a set of attached data-specific QoS.

DDS entities are modeled either as classes or typed interfaces. The latter imply a more efficient resource handling as knowledge of the data type prior to the execution allows allocating memory in advance rather than dynamically.

Fig. 1: Conceptual diagram of how information flows within DDS domains. Only entities belonging to the same domain can discover each other through matching topics, and consequently exchange data between publishers and subscribers.

CHAPTER

ONE

FAST DDS-GEN

Relying on interfaces implies the need for a generation tool that translates type descriptions into appropriate implementations that fill the gap between the interfaces and the middleware. This task is carried out by a dedicated generation tool, *Fast DDS-Gen*, a Java application that generates source code using the data types defined in an Interface Definition Language (IDL) file.

CHAPTER

TWO

RTPS WIRE PROTOCOL

The protocol used by *eProsima Fast DDS* to exchange messages over standard networks is the Real-Time Publish-Subscribe protocol (RTPS), an interoperability wire protocol for DDS defined and maintained by the OMG consortium. This protocol provides publisher-subscriber communications over transports such as TCP/UDP/IP, and guarantees compatibility among different DDS implementations.

Given its publish-subscribe roots and its specification designed for meeting the same requirements addressed by the DDS application domain, the RTPS protocol maps to many DDS concepts and is therefore a natural choice for DDS implementations. All the RTPS core entities are associated with an RTPS domain, which represents an isolated communication plane where endpoints match. The entities specified in the RTPS protocol are in one-to-one correspondence with the DDS entities, thus allowing the communication to occur.

CHAPTER

THREE

MAIN FEATURES

- **Two API Layers.** *eProsima Fast DDS* comprises a high-level DDS compliant layer focused on usability and a lower-level RTPS compliant layer that provides finer access to the RTPS protocol.
- **Real-Time behaviour.** *eProsima Fast DDS* can be configured to offer real-time features, guaranteeing responses within specified time constrains.
- **Built-in Discovery Server.** *eProsima Fast DDS* is based on the dynamical discovery of existing publishers and subscribers, and performs this task continuously without the need to contacting or setting any servers. However, a Client-Server discovery as well as other discovery paradigms can also be configured.
- **Sync and Async publication modes.** *eProsima Fast DDS* supports both synchronous and asynchronous data publication.
- **Best effort and reliable communication.** *eProsima Fast DDS* supports an optional reliable communication paradigm over *Best Effort* communications protocols such as UDP. Furthermore, another way of setting a reliable communication is to use our TCP transport.
- **Transport layers.** *eProsima Fast DDS* implements an architecture of pluggable transports. The current version implements five transports: UDPv4, UDPv6, TCPv4, TCPv6 and SHM (shared memory).
- **Security.** *eProsima Fast DDS* can be configured to provide secure communications. For this purpose, it implements pluggable security at three levels: authentication of remote participants, access control of entities and encryption of data.
- *Statistics Module.* *eProsima Fast DDS* can be configured to gather and provide information about the data being exchanged by the user application.
- **Flow controllers.** We support user-configurable flow controllers, that can be used to limit the amount of data to be sent under certain conditions.
- **Plug-and-play Connectivity.** New applications and services are automatically discovered, and can join and leave the network at any time without the need for reconfiguration.
- **Scalability and Flexibility.** DDS builds on the concept of a global data space. The middleware is in charge of propagating the information between publishers and subscribers. This guarantees that the distributed network is adaptable to reconfigurations and scalable to a large number of entities.
- **Application Portability.** The DDS specification includes a platform specific mapping to IDL, allowing an application using DDS to switch among DDS implementations with only a re-compile.
- **Extensibility.** *eProsima Fast DDS* allows the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- **Configurability and Modularity.** *eProsima Fast DDS* provides an intuitive way to be configured, either through code or XML profiles. Modularity allows simple devices to implement a subset of the protocol and still participate in the network.

- **High performance.** *eProsima Fast DDS* uses a static low-level serialization library, Fast CDR, a C++ library that serializes according to the standard CDR serialization mechanism defined in the RTPS Specification (see the Data Encapsulation chapter as a reference).
- **Easy to use.** The project comes with an out-of-the-box example, the *DDSHelloWorld* (see *Getting Started*) that puts into communication a publisher and a subscriber, showcasing how *eProsima Fast DDS* is deployed. Additionally, the interactive demo *ShapesDemo* is available for the user to dive into the DDS world. The DDS and the RTPS layers are thoroughly explained in the *DDS Layer* and *RTPS Layer* sections.
- **Low resources consumption.** *eProsima Fast DDS*:
  - Allows to preallocate resources, to minimize dynamic resource allocation.
  - Avoids the use of unbounded resources.
  - Minimizes the need to copy data.
- **Multi-platform.** The OS dependencies are treated as pluggable modules. Users may easily implement platform modules using the *eProsima Fast DDS* library on their target platforms. By default, the project can run over Linux, Windows and MacOS.
- **Free and Open Source.** The Fast DDS library, the underneath RTPS library, the generator tool, the internal dependencies (such as *eProsima Fast CDR*) and the external ones (such as the *foonathan* library) are free and open source.

CHAPTER

FOUR

CONTACTS AND COMMERCIAL SUPPORT

Find more about us at eProsima's webpage.

Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48

CHAPTER

FIVE

CONTRIBUTING TO THE DOCUMENTATION

Fast DDS-Docs is an open source project, and as such all contributions, both in the form of feedback and content generation, are most welcomed. To make such contributions, please refer to the Contribution Guidelines hosted in our GitHub repository.

CHAPTER

SIX

STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the sections below.

- *Installation Manual*
- *Fast DDS*
- *Fast DDS-Gen*
- *Release Notes*

eProsima Fast DDS is a C++ implementation of the DDS (Data Distribution Service) Specification, a protocol defined by the Object Management Group (OMG). The *eProsima Fast DDS* library provides both an Application Programming Interface (API) and a communication protocol that deploy a Data-Centric Publisher-Subscriber (DCPS) model, with the purpose of establishing efficient and reliable information distribution among Real-Time Systems. *eProsima Fast DDS* is predictable, scalable, flexible, and efficient in resource handling. For meeting these requirements, it makes use of typed interfaces and hinges on a many-to-many distributed network paradigm that neatly allows separation of the publisher and subscriber sides of the communication. *eProsima Fast DDS* comprises:

1. The *DDS API* implementation.
2. *Fast DDS-Gen*, a generation tool for bridging typed interfaces with the middleware implementation.
3. The underlying *RTPS* wire protocol implementation.

For all the above, *eProsima Fast DDS* has been chosen as the default middleware supported by the Robot Operating System 2 (ROS 2) in every long term (LTS) releases and most of the non-LTS releases.

6.1 DDS API

The communication model adopted by DDS is a many-to-many unidirectional data exchange where the applications that produce the data publish it to the local caches of subscribers belonging to applications that consume the data. The information flow is regulated by Quality of Service (QoS) policies established between the entities in charge of the data exchange.

As a data-centric model, DDS builds on the concept of a "global data space" accessible to all interested applications. Applications that want to contribute information declare their intent to become publishers, whereas applications that

want to access portions of the data space declare their intent to become subscribers. Each time a publisher posts new data into this space, the middleware propagates the information to all interested subscribers.

The communication happens across domains, i. e. isolated abstract planes that link all the distributed applications able to communicate with each other. Only entities belonging to a same domain can interact, and the matching between entities subscribing to data and entities publishing them is mediated by topics. Topics are unambiguous identifiers that associate a name, which is unique in the domain, to a data type and a set of attached data-specific QoS.

DDS entities are modeled either as classes or typed interfaces. The latter imply a more efficient resource handling as knowledge of the data type prior to the execution allows allocating memory in advance rather than dynamically.

Fig. 1: Conceptual diagram of how information flows within DDS domains. Only entities belonging to the same domain can discover each other through matching topics, and consequently exchange data between publishers and subscribers.

6.2 Fast DDS-Gen

Relying on interfaces implies the need for a generation tool that translates type descriptions into appropriate implementations that fill the gap between the interfaces and the middleware. This task is carried out by a dedicated generation tool, *Fast DDS-Gen*, a Java application that generates source code using the data types defined in an Interface Definition Language (IDL) file.

6.3 RTPS Wire Protocol

The protocol used by *eProsima Fast DDS* to exchange messages over standard networks is the Real-Time Publish-Subscribe protocol (RTPS), an interoperability wire protocol for DDS defined and maintained by the OMG consortium. This protocol provides publisher-subscriber communications over transports such as TCP/UDP/IP, and guarantees compatibility among different DDS implementations.

Given its publish-subscribe roots and its specification designed for meeting the same requirements addressed by the DDS application domain, the RTPS protocol maps to many DDS concepts and is therefore a natural choice for DDS implementations. All the RTPS core entities are associated with an RTPS domain, which represents an isolated communication plane where endpoints match. The entities specified in the RTPS protocol are in one-to-one correspondence with the DDS entities, thus allowing the communication to occur.

6.4 Main Features

- **Two API Layers.** *eProsima Fast DDS* comprises a high-level DDS compliant layer focused on usability and a lower-level RTPS compliant layer that provides finer access to the RTPS protocol.
- **Real-Time behaviour.** *eProsima Fast DDS* can be configured to offer real-time features, guaranteeing responses within specified time constrains.
- **Built-in Discovery Server.** *eProsima Fast DDS* is based on the dynamical discovery of existing publishers and subscribers, and performs this task continuously without the need to contacting or setting any servers. However, a Client-Server discovery as well as other discovery paradigms can also be configured.
- **Sync and Async publication modes.** *eProsima Fast DDS* supports both synchronous and asynchronous data publication.

- **Best effort and reliable communication.** *eProsima Fast DDS* supports an optional reliable communication paradigm over *Best Effort* communications protocols such as UDP. Furthermore, another way of setting a reliable communication is to use our TCP transport.
- **Transport layers.** *eProsima Fast DDS* implements an architecture of pluggable transports. The current version implements five transports: UDPv4, UDPv6, TCPv4, TCPv6 and SHM (shared memory).
- **Security.** *eProsima Fast DDS* can be configured to provide secure communications. For this purpose, it implements pluggable security at three levels: authentication of remote participants, access control of entities and encryption of data.
- *Statistics Module.* *eProsima Fast DDS* can be configured to gather and provide information about the data being exchanged by the user application.
- **Flow controllers.** We support user-configurable flow controllers, that can be used to limit the amount of data to be sent under certain conditions.
- **Plug-and-play Connectivity.** New applications and services are automatically discovered, and can join and leave the network at any time without the need for reconfiguration.
- **Scalability and Flexibility.** DDS builds on the concept of a global data space. The middleware is in charge of propagating the information between publishers and subscribers. This guarantees that the distributed network is adaptable to reconfigurations and scalable to a large number of entities.
- **Application Portability.** The DDS specification includes a platform specific mapping to IDL, allowing an application using DDS to switch among DDS implementations with only a re-compile.
- **Extensibility.** *eProsima Fast DDS* allows the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- **Configurability and Modularity.** *eProsima Fast DDS* provides an intuitive way to be configured, either through code or XML profiles. Modularity allows simple devices to implement a subset of the protocol and still participate in the network.
- **High performance.** *eProsima Fast DDS* uses a static low-level serialization library, Fast CDR, a C++ library that serializes according to the standard CDR serialization mechanism defined in the RTPS Specification (see the Data Encapsulation chapter as a reference).
- **Easy to use.** The project comes with an out-of-the-box example, the *DDSHelloWorld* (see *Getting Started*) that puts into communication a publisher and a subscriber, showcasing how *eProsima Fast DDS* is deployed. Additionally, the interactive demo *ShapesDemo* is available for the user to dive into the DDS world. The DDS and the RTPS layers are thoroughly explained in the *DDS Layer* and *RTPS Layer* sections.
- **Low resources consumption.** *eProsima Fast DDS*:
  - Allows to preallocate resources, to minimize dynamic resource allocation.
  - Avoids the use of unbounded resources.
  - Minimizes the need to copy data.
- **Multi-platform.** The OS dependencies are treated as pluggable modules. Users may easily implement platform modules using the *eProsima Fast DDS* library on their target platforms. By default, the project can run over Linux, Windows and MacOS.
- **Free and Open Source.** The Fast DDS library, the underneath RTPS library, the generator tool, the internal dependencies (such as *eProsima Fast CDR*) and the external ones (such as the *foonathan* library) are free and open source.

6.5 Contacts and Commercial support

Find more about us at eProsima's webpage.

Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48

6.6 Contributing to the documentation

Fast DDS-Docs is an open source project, and as such all contributions, both in the form of feedback and content generation, are most welcomed. To make such contributions, please refer to the Contribution Guidelines hosted in our GitHub repository.

6.7 Structure of the documentation

This documentation is organized into the sections below.

- *Installation Manual*
- *Fast DDS*
- *Fast DDS-Gen*
- *Release Notes*

6.8 Linux installation from binaries

The instructions for installing *eProsima Fast DDS* in a Linux environment from binaries are provided in this page.

- *Install*
  - *Contents*
  - *Run an application*
  - *Including Fast-DDS in a CMake project*
- *Uninstall*

6.8.1 Install

The latest release of *eProsima Fast DDS* for Linux is available at the eProsima website Downloads tab. Once downloaded, extract the contents in your preferred directory. Then, to install *eProsima Fast DDS* and all its dependencies in the system, execute the `install.sh` script with administrative privileges:

```
cd <extraction_directory>
sudo ./install.sh
```

Note: By default, *eProsima Fast DDS* does not compile tests. To activate them, please refer to the *Linux installation from sources* page.

To use the *xml* validation tool, please refer to the *Linux installation from sources* page.

Contents

The `src` folder contains the following packages:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocator library.
- `fastcdr`, a C++ library for data serialization according to the CDR standard (*Section 10.2.1.2 OMG CDR*).
- `fastrtps`, the core library of *eProsima Fast DDS* library.
- `fastddsgen`, a Java application that generates source code using the data types defined in an IDL file.

In case any of these components is unwanted, it can be simply renamed or removed from the `src` directory.

Run an application

When running an instance of an application using *eProsima Fast DDS*, it must be linked with the library where the packages have been installed, `/usr/local/lib/`. There are two possibilities:

- Prepare the environment locally by typing in the console used for running the *eProsima Fast DDS* instance the command:

```
export LD_LIBRARY_PATH=/usr/local/lib/
```

- Add it permanently to the `PATH` by executing:

```
echo 'export LD_LIBRARY_PATH=/usr/local/lib/' >> ~/.bashrc
```

Including Fast-DDS in a CMake project

The installer deploys *CMake config* files that simplify to incorporate **Fast-DDS** to any CMake project via the *find_package* CMake API.

By setting the CMake variable **BUILD_SHARED_LIBS** is possible to choose the desired linkage (dynamic or static library) in the CMake generator stage. If the variable is missing build process will default to static linking.

For example in order to build the examples dynamically linked to **Fast-DDS** do:

```
$ cmake -Bbuildexample -DBUILD_SHARED_LIBS=ON .
$ cmake --build buildexample --target install
```

6.8.2 Uninstall

To uninstall all installed components, execute the *uninstall.sh* script (with administrative privileges):

```
cd <extraction_directory>
sudo ./uninstall.sh
```

Warning: If any of the other components were already installed in some other way in the system, they will be removed as well. To avoid it, edit the script before executing it.

6.9 Windows installation from binaries

The instructions for installing *eProsima Fast DDS* in a Windows environment from binaries are provided in this page. It is organized as follows:

- *Requirements*
  - *Visual Studio*
- *Install*
  - *Contents*
  - *Environment variables*
  - *Including Fast-DDS in a CMake project*

First of all, the *Requirements* detailed below need to be met.

6.9.1 Requirements

The installation of *eProsima Fast DDS* in a Windows environment from binaries requires the following tools to be installed in the system:

- *Visual Studio*

Visual Studio

Visual Studio is required to have a C++ compiler in the system. For this purpose, make sure to check the `Desktop development with C++` option during the Visual Studio installation process.

If Visual Studio is already installed but the Visual C++ Redistributable packages are not, open Visual Studio and go to `Tools -> Get Tools and Features` and in the `Workloads` tab enable `Desktop development with C++`. Finally, click `Modify` at the bottom right.

6.9.2 Install

The latest release of *eProsima Fast DDS* for Windows is available at the company website downloads page. Once downloaded, execute the installer and follow the instructions, choosing the preferred Visual Studio version and architecture when prompted.

Note: By default, *eProsima Fast DDS* does not compile tests. To activate them, please refer to the *Windows installation from sources* page.

To use the *xml* validation tool, please refer to the *Windows installation from sources* page.

Contents

By default, the installation will download all the available packages, namely:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocator library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of *eProsima Fast DDS* library.
- `fastddsgen`, a Java application that generates source code using the data types defined in an IDL file.

Environment variables

eProsima Fast DDS requires the following environment variable setup in order to function properly:

- `FASTRTPSHOME`: Root folder where *eProsima Fast DDS* is installed.
- Additions to the `PATH`: The location of *eProsima Fast DDS* scripts and libraries should be appended to the `PATH`.

These variables are set automatically by checking the corresponding box during the installation process.

Including Fast-DDS in a CMake project

The installer deploys *CMake config* files that simplify to incorporate **Fast-DDS** to any CMake project via the *find_package* CMake API.

Shared and static libraries are provided by the installer. The user can select which one will be used in the CMake project using next mechanisms.

1. Through CMake package components when calling *find_package()*.

```
find_package(fastrtps shared) # Load shared library target
find_package(fastrtps static) # Load static library target
```

2. Through the custom CMake variable *fastrtps_SHARED_LIBS*.

```
cmake -Dfastrtps_SHARED_LIBS=ON .. # Load shared library target
cmake -Dfastrtps_SHARED_LIBS=OFF .. # Load static library target
```

3. Through the built-in CMake variable *BUILD_SHARED_LIBS*.

```
cmake -DBUILD_SHARED_LIBS=ON .. # Load shared library target
cmake -DBUILD_SHARED_LIBS=OFF .. # Load static library target
```

4. In case no previous mechanism is used, CMake will try to load static library target. If it fails then CMake will try to load shared library target.

For example in order to build the examples dynamically linked to **Fast-DDS** do:

```
> cmake -Bbuildexample -DBUILD_SHARED_LIBS=ON .
> cmake --build buildexample --target install
```

6.10 Linux installation from sources

The instructions for installing the *Fast DDS library*, the *Fast DDS Python bindings* and the *Fast DDS-Gen* generation tool from sources are provided in this page. It is organized as follows:

- *Fast DDS library installation*
  - *Requirements*
  - *Dependencies*
  - *Colcon installation*
  - *CMake installation*
- *Fast DDS Python bindings installation*
  - *Requirements*
  - *Dependencies*
  - *Colcon installation*
  - *CMake installation*
- *Fast DDS-Gen installation*
  - *Requirements*
  - *Compiling Fast DDS-Gen*

6.10.1 Fast DDS library installation

This section describes the instructions for installing *eProsima Fast DDS* in a Linux environment from sources. The following packages will be installed:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocator library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of *eProsima Fast DDS* library.

First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon* or the *CMake* installation instructions.

Requirements

The installation of *eProsima Fast DDS* in a Linux environment from sources requires the following tools to be installed in the system:

- *CMake, g++, pip3, wget and git*

CMake, g++, pip3, wget and git

These packages provide the tools required to install *eProsima Fast DDS* and its dependencies from command line. Install CMake, g++, pip3, wget and git using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install cmake g++ python3-pip wget git
```

Dependencies

eProsima Fast DDS has the following dependencies, when installed from sources in a Linux environment:

- *Asio and TinyXML2 libraries*
- *OpenSSL*
- *Libp11 and SoftHSM libraries*
- *Gtest* [optional]
- *XML validation tool* [optional]

Asio and TinyXML2 libraries

Asio is a cross-platform C++ library for network and low-level I/O programming, which provides a consistent asynchronous model. TinyXML2 is a simple, small and efficient C++ XML parser. Install these libraries using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libasio-dev libtinyxml2-dev
```

OpenSSL

OpenSSL is a robust toolkit for the TLS and SSL protocols and a general-purpose cryptography library. Install OpenSSL using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libssl-dev
```

Libp11 and SoftHSM libraries

Libp11 provides PKCS#11 support for OpenSSL. This is an optional dependency, that is needed only when *eprosima Fast DDS* is used with security and PKCS#11 URIs.

Install libp11 using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libp11-dev
```

SoftHSM is a software implementation of an HSM (Hardware Security Module). If *eProsima Fast DDS* tests are activated and *libp11* is installed on the system, SoftHSM is additionally required to run tests of PKCS#11 features.

Install SoftHSM using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install softhsm2
```

Note that the *softhsm2* package creates a new group called *softhsm*. In order to grant access to the HSM module a user must belong to this group.

```
sudo usermod -a -G softhsm <user>
```

OpenSSL access HSM and other hardware devices through its engine functionality. In order to set up a new engine the OpenSSL configuration files (usually */etc/ssl/openssl.cnf*) must be updated specifying the libp11 and hardware module (here SoftHSM) dynamic libraries location.

This configuration step can be avoided using p11kit which allows OpenSSL to find PKCS#11 devices on runtime without static configuration. This kit is often available through the Linux distribution package manager. On Ubuntu, for example:

```
sudo apt install libengine-pkcs11-openssl
```

Once installed, to check p11kit is able to find the SoftHSM module use:

```
p11-kit list-modules
```

In order to check if OpenSSL is able to access PKCS#11 engine use:

```
openssl engine pkcs11 -t
```

Gtest

GTest is a unit testing library for C++. By default, *eProsima Fast DDS* does not compile tests. It is possible to activate them with the opportune CMake configuration options when calling colcon or CMake. For more details, please refer to the *CMake options* section. Also add the Gtest repository into the workspace directory.

```
git clone --branch release-1.11.0 https://github.com/google/googletest src/googletest-
↪distribution
```

XML validation tool

XML validation is a new command introduced to validate the XML profiles against an XSD schema through Fast DDS CLI. That ensures the proper characterization of the entities using the xml profiles.

For more details, please refer to the *xml* section.

Install the xmlschema dependency to be able to use this optional tool.

Colcon installation

colcon is a command line tool based on CMake aimed at building sets of software packages. This section explains how to use it to compile *eProsima Fast DDS* and its dependencies.

1. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

```
pip3 install -U colcon-common-extensions vcstool
```

> **Note:** Mind that under non-root users, `pip3` may install python `colcon` and `vcs` executables in `$HOME/.local/bin`, for instance when running with `--user`. To be able to run these applications, make sure that `pip3` binary installation directory is in your `$PATH` (`$HOME/.local/bin` is normally introduced while login on an interactive non-root shell).

2. Create a `Fast-DDS` directory and download the repos file that will be used to install *eProsima Fast DDS* and its dependencies:

```
mkdir ~/Fast-DDS
cd ~/Fast-DDS
wget https://raw.githubusercontent.com/eProsima/Fast-DDS/master/fastrtps.repos
mkdir src
vcs import src < fastrtps.repos
```

3. Build the packages:

```
colcon build
```

Note: Being based on CMake, it is possible to pass CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual.

Run an application

When running an instance of an application using *eProsima Fast DDS*, the colcon overlay built in the dedicated `Fast-DDS` directory must be sourced. There are two possibilities:

- Every time a new shell is opened, prepare the environment locally by typing the command:

```
source ~/Fast-DDS/install/setup.bash
```

- Add the sourcing of the colcon overlay permanently to the `PATH`, by typing the following:

```
echo 'source ~/Fast-DDS/install/setup.bash' >> ~/.bashrc
```

CMake installation

This section explains how to compile *eProsima Fast DDS* with CMake, either *locally* or *globally*.

Local installation

1. Create a `Fast-DDS` directory where to download and build *eProsima Fast DDS* and its dependencies:

```
mkdir ~/Fast-DDS
```

2. Clone the following dependencies and compile them using CMake.
   - Foonathan memory

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/foonathan_memory_vendor.git
mkdir foonathan_memory_vendor/build
cd foonathan_memory_vendor/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/Fast-DDS/install -DBUILD_SHARED_LIBS=ON
cmake --build . --target install
```

   - Fast CDR

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/Fast-CDR.git
mkdir Fast-CDR/build
cd Fast-CDR/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/Fast-DDS/install
cmake --build . --target install
```

3. Once all dependencies are installed, install *eProsima Fast DDS*:

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/Fast-DDS.git
mkdir Fast-DDS/build
cd Fast-DDS/build
cmake ..  -DCMAKE_INSTALL_PREFIX=~/Fast-DDS/install
cmake --build . --target install
```

Note: By default, *eProsima Fast DDS* does not compile tests. However, they can be activated by downloading and installing Gtest.

Global installation

To install *eProsima Fast DDS* system-wide instead of locally, remove all the flags that appear in the configuration steps of `Fast-CDR` and `Fast-DDS`, and change the first in the configuration step of `foonathan_memory_vendor` to the following:

```
-DCMAKE_INSTALL_PREFIX=/usr/local/ -DBUILD_SHARED_LIBS=ON
```

Note: Installation on system directories may need of permissions. Maybe permissions have to be granted through sudo.

```
sudo cmake --build . --target install
```

Run an application

When running an instance of an application using *eProsima Fast DDS*, it must be linked with the library where the packages have been installed, which in the case of system-wide installation is: `/usr/local/lib/` (if local installation is used, adjust for the correct directory). There are two possibilities:

- Prepare the environment locally by typing the command:

```
export LD_LIBRARY_PATH=/usr/local/lib/
```

- Add it permanently it to the `PATH`, by typing:

```
echo 'export LD_LIBRARY_PATH=/usr/local/lib/' >> ~/.bashrc
```

6.10.2 Fast DDS Python bindings installation

This section provides the instructions for installing *Fast DDS Python bindings* in a Linux environment from sources. *Fast DDS Python bindings* is an extension of *Fast DDS* which provides access to the Fast DDS API through Python. Therefore, its installation is an extension of the installation of *Fast DDS*.

Fast DDS Python bindings source code consists on several *.i* files which will be processed by SWIG. Then C++ files (for connecting C++ and Python) and Python files (Python module for Fast DDS) will be generated.

First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon* or the *CMake* installation instructions.

Requirements

The installation of *Fast DDS Python bindings* in a Linux environment from sources requires the following tools to be installed in the system:

- *Fast DDS requirements*
- *SWIG*
- *Header files and static library for Python*

SWIG

SWIG is a development tool that allows connecting programs written in C/C++ with a variety of other programming languages, among them Python. SWIG 4.0 is required in order to build Fast DDS Python bindings.

Note: More recent SWIG releases are not yet supported. Please, ensure to be using SWIG 4.0.

SWIG can be installed directly from the package manager of the appropriate Linux distribution. For Ubuntu, please run:

```
sudo apt install swig
```

Header files and static library for Python

Python static libraries and header files are needed to compile C++ source code generated by SWIG. They can be installed directly from the package manager of the appropriate Linux distribution. For Ubuntu, please run:

```
sudo apt install libpython3-dev
```

Dependencies

Fast DDS Python bindings has the following dependencies, when installed from sources in a Linux environment:

- *Fast DDS dependencies*

Colcon installation

colcon is a command line tool based on CMake aimed at building sets of software packages. This section explains how to use it to compile *Fast DDS Python bindings* and its dependencies.

1. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

   ```
   pip3 install -U colcon-common-extensions vcstool
   ```

   Note: If this fails due to an Environment Error, add the `--user` flag to the `pip3` installation command.

2. Create a `Fast-DDS-python` directory and download the repos file that will be used to install *Fast DDS Python bindings* and its dependencies:

   ```
   mkdir ~/Fast-DDS-python
   cd ~/Fast-DDS-python
   wget https://raw.githubusercontent.com/eProsima/Fast-DDS-python/main/fastdds_python.
   ↪repos
   mkdir src
   vcs import src < fastdds_python.repos
   ```

3. Build the packages:

   ```
   colcon build
   ```

Note: Being based on CMake, it is possible to pass CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual.

Run an application

When running an instance of an application using *Fast DDS Python bindings*, the colcon overlay built in the dedicated `Fast-DDS-python` directory must be sourced. There are two possibilities:

- Every time a new shell is opened, prepare the environment locally by typing the command:

```
source ~/Fast-DDS-python/install/setup.bash
```

- Add the sourcing of the colcon overlay permanently to the `PATH`, by typing the following:

```
echo 'source ~/Fast-DDS-python/install/setup.bash' >> ~/.bashrc
```

CMake installation

This section explains how to compile *Fast DDS Python bindings* with CMake, either *locally* or *globally*.

Local installation

1. Create a `Fast-DDS-python` directory where to download and build *Fast DDS Python bindings* and its dependencies:

```
mkdir ~/Fast-DDS-python
```

2. Clone the following dependencies and compile them using CMake.

   - Foonathan memory

```
cd ~/Fast-DDS-python
git clone https://github.com/eProsima/foonathan_memory_vendor.git
mkdir foonathan_memory_vendor/build
cd foonathan_memory_vendor/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/Fast-DDS-python/install -DBUILD_SHARED_LIBS=ON
cmake --build . --target install
```

   - Fast CDR

```
cd ~/Fast-DDS-python
git clone https://github.com/eProsima/Fast-CDR.git
mkdir Fast-CDR/build
cd Fast-CDR/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/Fast-DDS-python/install
cmake --build . --target install
```

   - Fast DDS

```
cd ~/Fast-DDS-python
git clone https://github.com/eProsima/Fast-DDS.git
mkdir Fast-DDS/build
cd Fast-DDS/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/Fast-DDS-python/install
cmake --build . --target install
```

3. Once all dependencies are installed, install *Fast DDS Python bindings*:

```
cd ~/Fast-DDS-python
git clone https://github.com/eProsima/Fast-DDS-python.git
mkdir -p Fast-DDS-python/fastdds_python/build
cd Fast-DDS-python/fastdds_python/build
cmake ..  -DCMAKE_INSTALL_PREFIX=~/Fast-DDS-python/install
cmake --build . --target install
```

Global installation

To install *Fast DDS Python bindings* system-wide instead of locally, remove all the flags that appear in the configuration steps of `Fast-CDR`, `Fast-DDS` and `Fast-DDS-python`, and change the first in the configuration step of `foonathan_memory_vendor` to the following:

```
-DCMAKE_INSTALL_PREFIX=/usr/local/ -DBUILD_SHARED_LIBS=ON
```

Note: Installation on system directories may need of permissions. Maybe permissions have to be granted through sudo.

```
sudo cmake --build . --target install
```

Run an application

When running an instance of an application using *Fast DDS Python bindings*, it must be linked with the library where the packages have been installed, which in the case of system-wide installation is: `/usr/local/lib/` (if local installation is used, adjust for the correct directory). There are two possibilities:

- Prepare the environment locally by typing the command:

```
export LD_LIBRARY_PATH=/usr/local/lib/
```

- Add it permanently it to the `PATH`, by typing:

```
echo 'export LD_LIBRARY_PATH=/usr/local/lib/' >> ~/.bashrc
```

6.10.3 Fast DDS-Gen installation

This section provides the instructions for installing *Fast DDS-Gen* in a Linux environment from sources. *Fast DDS-Gen* is a Java application that generates source code using the data types defined in an IDL file. Please refer to *Introduction* for more information.

Requirements

Fast DDS-Gen is built using Gradle. Gradle is an open-source build automation tool which requires a Java version to be executed (see Gradle-Java compatibility matrix).

Important: Even though earlier versions of Gradle support Java 8, *Fast DDS-Gen* stopped supporting Java versions previous to Java 11 since release v2.4.0.

Important: *Fast DDS-Gen* introduced support for Gradle 7 in release v2.2.0. Gradle 8 is not yet supported.

Java JDK

The JDK is a development environment for building applications and components using the Java language. There are several versions of Java available. For instance, to install Java 11 JDK, run the following command:

```
sudo apt install openjdk-11-jdk
```

Note: *Fast DDS-Gen* supports Java versions from 11 to 19.

Compiling Fast DDS-Gen

In order to compile *Fast DDS-Gen*, an executable script is included in the repository which will download Gradle temporarily for the compilation step. Please, follow the steps below to build *Fast DDS-Gen*:

Note: If Fast DDS has already been installed following *Colcon installation*, skip cloning *Fast DDS-Gen*'s repository, as it can already be found under the `src` directory within the colcon workspace.

```
cd ~
git clone --recursive https://github.com/eProsima/Fast-DDS-Gen.git
cd Fast-DDS-Gen
./gradlew assemble
```

Note: In case that a supported Gradle version is already installed in the system, *Fast DDS-Gen* can also be built running directly:

```
gradle assemble
```

Contents

The `Fast-DDS-Gen` folder contains the following packages:

- `share/fastddsgen`, where the generated Java application is.
- `scripts`, containing some user friendly scripts.

> **Note:** To make these scripts accessible from any shell session and directory, add the `scripts` folder path to the `PATH` environment variable.

6.11 Windows installation from sources

The instructions for installing both the *Fast DDS library* and the *Fast DDS-Gen* generation tool from sources are provided in this page. It is organized as follows:

- *Fast DDS library installation*
  - *Requirements*
  - *Dependencies*
  - *Colcon installation*
  - *CMake installation*
- *Fast DDS Python bindings installation*
  - *Requirements*
  - *Dependencies*
  - *Colcon installation*
  - *CMake installation*
- *Fast DDS-Gen installation*
  - *Requirements*
  - *Compiling Fast DDS-Gen*

6.11.1 Fast DDS library installation

This section provides the instructions for installing *eProsima Fast DDS* in a Windows environment from sources. The following packages will be installed:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocator library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of *eProsima Fast DDS* library.

First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon* or the *CMake* installation instructions.

Requirements

The installation of *eProsima Fast DDS* in a Windows environment from sources requires the following tools to be installed in the system:

- *Visual Studio*
- *Chocolatey*
- *CMake, pip3, wget and git*
- *Gtest* [optional]
- *XML validation tool* [optional]

Visual Studio

Visual Studio is required to have a C++ compiler in the system. For this purpose, make sure to check the `Desktop development with C++` option during the Visual Studio installation process.

If Visual Studio is already installed but the Visual C++ Redistributable packages are not, open Visual Studio and go to `Tools -> Get Tools and Features` and in the `Workloads` tab enable `Desktop development with C++`. Finally, click `Modify` at the bottom right.

Chocolatey

Chocolatey is a Windows package manager. It is needed to install some of *eProsima Fast DDS*'s dependencies. Download and install it directly from the website.

CMake, pip3, wget and git

These packages provide the tools required to install *eProsima Fast DDS* and its dependencies from command line. Download and install CMake, pip3, wget and git by following the instructions detailed in the respective websites. Once installed, add the path to the executables to the `PATH` from the *Edit the system environment variables* control panel.

Gtest

GTest is a unit testing library for C++. By default, *eProsima Fast DDS* does not compile tests. It is possible to activate them with the opportune CMake configuration options when calling colcon or CMake. For more details, please refer to the *CMake options* section. Also add the Gtest repository into the workspace directory.

```
git clone --branch release-1.11.0 https://github.com/google/googletest src/googletest-
↪distribution
```

and add next argument to the *colcon* call

```
colcon build --cmake-args -Dgtest_force_shared_crt=ON
```

XML validation tool

XML validation is a new command introduced to validate the XML profiles against an XSD schema through Fast DDS CLI. That ensures the proper characterization of the entities using the xml profiles.

For more details, please refer to the *xml* section.

Install the xmlschema dependency to be able to use this optional tool.

Dependencies

eProsima Fast RTPS has the following dependencies, when installed from sources in a Windows environment:

- *Asio and TinyXML2 libraries*
- *OpenSSL*

Asio and TinyXML2 libraries

Asio is a cross-platform C++ library for network and low-level I/O programming, which provides a consistent asynchronous model. TinyXML2 is a simple, small and efficient C++ XML parser. They can be downloaded directly from the links below:

- Asio
- TinyXML2

After downloading these packages, open an administrative shell with *PowerShell* and execute the following command:

```
choco install -y -s <PATH_TO_DOWNLOADS> asio tinyxml2
```

where <PATH_TO_DOWNLOADS> is the folder into which the packages have been downloaded.

OpenSSL

OpenSSL is a robust toolkit for the TLS and SSL protocols and a general-purpose cryptography library. Install it by running the following command inside an administrative shell with *PowerShell*:

```
choco install -y openssl
```

Libp11 and SoftHSM libraries

Libp11 provides PKCS#11 support for OpenSSL. This is an optional dependency, that is needed only when *eprosima Fast DDS* is used with security and PKCS#11 URIs.

Download the latest libp11 version for Windows from this repository and follow the installation instructions

SoftHSM is a software implementation of an HSM (Hardware Security Module). If *eProsima Fast DDS* tests are activated and *libp11* is installed on the system, SoftHSM is additionally required to run tests of PKCS#11 features.

Download the SoftHSM for Windows installer from this repository. Execute the installer and follow the installation instructions.

OpenSSL access HSM and other hardware devices through its engine functionality. In order to set up a new engine the OpenSSL configuration files must be updated specifying the libp11 and hardware module (here SoftHSM) dynamic libraries location.

OpenSSL on Windows references its default configuration file through the *OPENSSL_CONF* environment variable. By default OpenSSL installs two identical default configuration files:

- *C:\Program Files\OpenSSL-Win64\bin\cnf\openssl.cnf* mimics the Linux distributions one.
- *C:\Program Files\OpenSSL-Win64\bin\openssl.cfg* kept for backward compatibility.

Neither of them are loaded by default. In order to direct OpenSSL to load one of them or any other we must set the variable:

```
cmd> set OPENSSL_CONF=C:\Program Files\OpenSSL-Win64\bin\cnf\openssl.cnf
powershell> $Env:OPENSSL_CONF="C:\Program Files\OpenSSL-Win64\bin\cnf\openssl.cnf"
```

Once we have hinted OpenSSL the configuration file to use we must modify it to set up the new PKCS#11 engine following the OpenSSL guidelines replacing the binaries path with the proper ones. For example, before any section in the configuration file we introduce:

```
openssl_conf = openssl_init
```

at the end of the file we include the engine devoted sections. Note to use POSIX path separator instead of the windows one.

```
[openssl_init]
    engines = engine_section

[engine_section]
    pkcs11 = pkcs11_section

    [pkcs11_section]
    engine_id = pkcs11
    dynamic_path = C:/Program Files/libp11/src/pkcs11.dll
    MODULE_PATH = C:/Program Files (x86)/SoftHSM2/lib/softhsm2-x64.dll
    init = 0
```

A proper set up can be verified using OpenSSL command line tool:

```
openssl engine pkcs11 -t
```

Colcon installation

colcon is a command line tool based on CMake aimed at building sets of software packages. This section explains how to use it to compile *eProsima Fast DDS* and its dependencies.

Important: Run colcon within a Visual Studio prompt. To do so, launch a *Developer Command Prompt* from the search engine.

---

1. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

   ```
   pip3 install -U colcon-common-extensions vcstool
   ```

and add the path to the `vcs` executable to the `PATH` from the *Edit the system environment variables* control panel.

Note: If this fails due to an Environment Error, add the `--user` flag to the `pip3` installation command.

2. Create a `Fast-DDS` directory and download the repos file that will be used to install *eProsima Fast DDS* and its dependencies:

```
mkdir ~\Fast-DDS
cd ~\Fast-DDS
wget https://raw.githubusercontent.com/eProsima/Fast-DDS/master/fastrtps.repos -
↪output fastrtps.repos
mkdir src
vcs import src --input fastrtps.repos
```

Finally, use colcon to compile all software:

```
colcon build
```

Note: Being based on CMake, it is possible to pass the CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual.

Run an application

When running an instance of an application using *eProsima Fast DDS*, the colcon overlay built in the dedicated `Fast-DDS` directory must be sourced. There are two possibilities:

- Every time a new shell is opened, prepare the environment locally by typing the command:

```
setup.bat
```

- Add the sourcing of the colcon overlay permanently, by opening the *Edit the system environment variables* control panel, and adding `~/Fast-DDS/install/setup.bat` to the `PATH`.

CMake installation

This section explains how to compile *eProsima Fast DDS* with CMake, either *locally* or *globally*.

Local installation

1. Open a command prompt, and create a `Fast-DDS` directory where to download and build *eProsima Fast DDS* and its dependencies:

```
mkdir %USERPROFILE%\Fast-DDS
```

2. Clone the following dependencies and compile them using CMake.

   - Fast DDS depends on Foonathan memory. To ease the dependency management, *eProsima* provides a vendor package Foonathan memory vendor, which downloads and builds a specific revision of *Foonathan memory* if the library is not found in the system.

```
cd %USERPROFILE%\Fast-DDS
git clone https://github.com/eProsima/foonathan_memory_vendor.git
cd foonathan_memory_vendor
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=%USERPROFILE%/Fast-DDS/install ..
cmake --build . --target install
```

- Fast CDR

```
cd %USERPROFILE%\Fast-DDS
git clone https://github.com/eProsima/Fast-CDR.git
cd Fast-CDR
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=%USERPROFILE%/Fast-DDS/install ..
cmake --build . --target install
```

3. Once all dependencies are installed, install *eProsima Fast DDS*:

```
cd %USERPROFILE%\Fast-DDS
git clone https://github.com/eProsima/Fast-DDS.git
cd Fast-DDS
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=%USERPROFILE%/Fast-DDS/install ..
cmake --build . --target install
```

Global installation

To install *eProsima Fast DDS* system-wide instead of locally, remove the `CMAKE_INSTALL_PREFIX` flags that appear in the configuration steps of `Fast-CDR` and `Fast-DDS`.

Note: By default, *eProsima Fast DDS* does not compile tests. However, they can be activated by downloading and installing Gtest.

Run an application

When running an instance of an application using *eProsima Fast DDS*, it must be linked with the library where the packages have been installed. This can be done by opening the *Edit system environment variables* control panel and adding to the `PATH` the *Fast DDS* and *Fast CDR* installation directories:

- *Fast DDS*: C:\Program Files\fastrtps
- *Fast CDR*: C:\Program Files\fastcdr

6.11.2 Fast DDS Python bindings installation

This section provides the instructions for installing *Fast DDS Python bindings* in a Windows environment from sources. *Fast DDS Python bindings* is an extension of *Fast DDS* which provides access to the Fast DDS API through Python. Therefore, its installation is an extension of the installation of *Fast DDS*.

Fast DDS Python bindings source code consists on several *.i* files which will be processed by *SWIG*. Then C++ files (for connecting C++ and Python) and Python files (Python module for Fast DDS) will be generated.

First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon* or the *CMake* installation instructions.

Requirements

The installation of *Fast DDS Python bindings* in a Windows environment from sources requires the following tools to be installed in the system:

- *Fast DDS requirements*
- *SWIG*

SWIG

SWIG is a development tool that allows connecting programs written in C/C++ with a variety of other programming languages, among them Python. SWIG 4.0 is required in order to build Fast DDS Python bindings.

Note: More recent *SWIG* releases are not yet supported. Please, ensure to be using SWIG 4.0.

Dependencies

Fast DDS Python bindings has the following dependencies, when installed from sources in a Windows environment:

- *Fast DDS dependencies*

Colcon installation

colcon is a command line tool based on CMake aimed at building sets of software packages. This section explains how to use it to compile *Fast DDS Python bindings* and its dependencies.

Important: Run colcon within a Visual Studio prompt. To do so, launch a *Developer Command Prompt* from the search engine.

1. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

   ```
   pip3 install -U colcon-common-extensions vcstool
   ```

   and add the path to the `vcs` executable to the `PATH` from the *Edit the system environment variables* control panel.

   Note: If this fails due to an Environment Error, add the `--user` flag to the `pip3` installation command.

2. Create a `Fast-DDS-python` directory and download the repos file that will be used to install *Fast DDS Python bindings* and its dependencies:

```
mkdir ~\Fast-DDS-python
cd ~\Fast-DDS-python
wget https://raw.githubusercontent.com/eProsima/Fast-DDS-python/main/fastdds_python.
↪repos
mkdir src
vcs import src --input fastdds_python.repos
```

3. Build the packages:

```
colcon build
```

Note: Being based on CMake, it is possible to pass CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual.

Run an application

When running an instance of an application using *Fast DDS Python bindings*, the colcon overlay built in the dedicated `Fast-DDS-python` directory must be sourced. There are two possibilities:

- Every time a new shell is opened, prepare the environment locally by typing the command:

```
setup.bat
```

- Add the sourcing of the colcon overlay permanently, by opening the *Edit the system environment variables* control panel, and adding `~/Fast-DDS/install/setup.bat` to the `PATH`.

CMake installation

This section explains how to compile *Fast DDS Python bindings* with CMake, either *locally* or *globally*.

Local installation

1. Open a command prompt, and create a `Fast-DDS-python` directory where to download and build *Fast DDS Python bindings* and its dependencies:

```
mkdir %USERPROFILE%\Fast-DDS-python
```

2. Clone the following dependencies and compile them using CMake.

   - Fast DDS depends on Foonathan memory. To ease the dependency management, *eProsima* provides a vendor package Foonathan memory vendor, which downloads and builds a specific revision of *Foonathan memory* if the library is not found in the system.

```
cd %USERPROFILE%\Fast-DDS-python
git clone https://github.com/eProsima/foonathan_memory_vendor.git
cd foonathan_memory_vendor
mkdir build && cd build
```

(continues on next page)

(continued from previous page)

```
cmake -DCMAKE_INSTALL_PREFIX=%USERPROFILE%/Fast-DDS-python/install ..
cmake --build . --target install
```

- Fast CDR

```
cd %USERPROFILE%\Fast-DDS-python
git clone https://github.com/eProsima/Fast-CDR.git
cd Fast-CDR
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=%USERPROFILE%/Fast-DDS-python/install ..
cmake --build . --target install
```

- Fast DDS

```
cd %USERPROFILE%\Fast-DDS-python
git clone https://github.com/eProsima/Fast-DDS.git
cd Fast-DDS
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=%USERPROFILE%/Fast-DDS-python/install ..
cmake --build . --target install
```

3. Once all dependencies are installed, install *Fast DDS Python bindings*:

```
cd ~/Fast-DDS-python
git clone https://github.com/eProsima/Fast-DDS-python.git
cd Fast-DDS-python
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=%USERPROFILE%/Fast-DDS-python/install ..
cmake --build . --target install
```

Global installation

To install *Fast DDS Python bindings* system-wide instead of locally, remove all the flags that appear in the configuration steps of `Fast-CDR`, `Fast-DDS` and `Fast-DDS-python`, and change the first in the configuration step of `foonathan_memory_vendor` to the following:

```
-DCMAKE_INSTALL_PREFIX=/usr/local/ -DBUILD_SHARED_LIBS=ON
```

Note: Installation on system directories may need of permissions. Maybe permissions have to be granted through sudo.

```
sudo cmake --build . --target install
```

Run an application

When running an instance of an application using *Fast DDS Python bindings*, it must be linked with the library where the packages have been installed. This can be done by opening the *Edit system environment variables* control panel and adding to the `PATH` the *Fast DDS python*, *Fast CDR* and *Fast DDS* installation directories:

- *Fast DDS python*: C:\Program Files\fastdds_python
- *Fast DDS*: C:\Program Files\fastrtps
- *Fast CDR*: C:\Program Files\fastcdr

6.11.3 Fast DDS-Gen installation

This section outlines the instructions for installing *Fast DDS-Gen* in a Windows environment from sources. *Fast DDS-Gen* is a Java application that generates source code using the data types defined in an IDL file. Please refer to *Introduction* for more information.

Requirements

Fast DDS-Gen is built using Gradle. Gradle is an open-source build automation tool which requires a Java version to be executed (see Gradle-Java compatibility matrix).

Important: Even though earlier versions of Gradle support Java 8, *Fast DDS-Gen* stopped supporting Java versions previous to Java 11 since release v2.4.0.

Important: *Fast DDS-Gen* introduced support for Gradle 7 in release v2.2.0. Gradle 8 is not yet supported.

Java JDK

The JDK is a development environment for building applications and components using the Java language. Download and install it following the steps given in the Oracle website.

Note: *Fast DDS-Gen* supports Java versions from 11 to 19.

Compiling Fast DDS-Gen

In order to compile *Fast DDS-Gen*, an executable script is included in the repository which will download Gradle temporarily for the compilation step. Please, follow the steps below to build *Fast DDS-Gen*:

Note: If Fast DDS has already been installed following *Colcon installation*, skip cloning *Fast DDS-Gen*'s repository, as it can already be found under the `src` directory within the colcon workspace.

```
cd ~
git clone --recursive https://github.com/eProsima/Fast-DDS-Gen.git
cd Fast-DDS-Gen
gradlew.bat assemble
```

Note: In case that a supported Gradle version is already installed in the system, *Fast DDS-Gen* can also be built running directly:

```
gradle assemble
```

Contents

The `Fast-DDS-Gen` folder contains the following packages:

- `share/fastddsgen`, where the generated Java application is.
- `scripts`, containing some user friendly scripts.

  Note: To make these scripts accessible from any directory, add the `scripts` folder path to the `PATH` environment variable.

6.12 Mac OS installation from sources

The instructions for installing both the *Fast DDS library* and the *Fast DDS-Gen* generation tool from sources are provided in this page. It is organized as follows:

- *Fast DDS library installation*
  - *Requirements*
  - *Dependencies*
  - *Colcon installation*
  - *CMake installation*
- *Fast DDS-Gen installation*
  - *Requirements*
  - *Compiling Fast DDS-Gen*

6.12.1 Fast DDS library installation

This section describes the instructions for installing *eProsima Fast DDS* in a Mac OS environment from sources. The following packages will be installed:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocator library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of *eProsima Fast DDS* library.

First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon*) or the *CMake*) installation instructions.

Requirements

The installation of *eProsima Fast DDS* in a MacOS environment from sources requires the following tools to be installed in the system:

- *Homebrew*
- *Xcode Command Line Tools*
- *CMake, g++, pip3, wget and git*
- *Gtest* [optional]
- *XML validation tool* [optional]

Homebrew

Homebrew is a macOS package manager, it is needed to install some of *eProsima Fast DDS*'s dependencies. To install it open a terminal window and run the following command.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install.sh)"
```

Xcode Command Line Tools

The Xcode command line tools package is separate from Xcode and allows for command line development in mac. The previous step should have installed Xcode CLI, to check the correct installation run the following command:

```
gcc --version
```

CMake, g++, pip3, wget and git

These packages provide the tools required to install *eProsima Fast DDS* and its dependencies from command line. Install CMake, pip3 and wget using the Homebrew package manager:

```
brew install cmake python3 wget
```

Gtest

GTest is a unit testing library for C++. By default, *eProsima Fast DDS* does not compile tests. It is possible to activate them with the opportune CMake configuration options when calling colcon or CMake. For more details, please refer to the *CMake options* section. Also add the Gtest repository into the workspace directory.

```
git clone --branch release-1.11.0 https://github.com/google/googletest src/googletest-
↪distribution
```

XML validation tool

XML validation is a new command introduced to validate the XML profiles against an XSD schema through Fast DDS CLI. That ensures the proper characterization of the entities using the xml profiles.

For more details, please refer to the *xml* section.

Install the xmlschema dependency to be able to use this optional tool.

Dependencies

eProsima Fast DDS has the following dependencies, when installed from binaries in a Linux environment:

- *Asio and TinyXML2 libraries*
- *OpenSSL*

Asio and TinyXML2 libraries

Asio is a cross-platform C++ library for network and low-level I/O programming, which provides a consistent asynchronous model. TinyXML2 is a simple, small and efficient C++ XML parser. Install these libraries using Homebrew:

```
brew install asio tinyxml2
```

OpenSSL

OpenSSL is a robust toolkit for the TLS and SSL protocols and a general-purpose cryptography library. Install OpenSSL using Homebrew:

```
brew install openssl@1.1
```

Colcon installation

colcon is a command line tool based on CMake aimed at building sets of software packages. This section explains how to use it to compile *eProsima Fast DDS* and its dependencies.

1. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

   ```
   pip3 install -U colcon-common-extensions vcstool
   ```

2. Create a `Fast-DDS` directory and download the repos file that will be used to install *eProsima Fast DDS* and its dependencies:

```
mkdir ~/Fast-DDS
cd ~/Fast-DDS
wget https://raw.githubusercontent.com/eProsima/Fast-DDS/master/fastrtps.repos
mkdir src
vcs import src < fastrtps.repos
```

3. Build the packages:

```
colcon build
```

Note: The `--cmake-args` option allows to pass the CMake configuration options to the `colcon build` command. In Mac OS the location of OpenSSL is not found automatically and therefore has to be passed explicitly: `--cmake-args -DOPENSSL_ROOT_DIR=/usr/local/opt/openssl -DOPENSSL_LIBRARIES=/usr/local/opt/openssl/lib`. This is only required when building with *Security*. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual.

Run an application

When running an instance of an application using *eProsima Fast DDS*, the colcon overlay built in the dedicated `Fast-DDS` directory must be sourced. There are two possibilities:

- Every time a new shell is opened, prepare the environment locally by typing the command:

```
source ~/Fast-DDS/install/setup.bash
```

- Add the sourcing of the colcon overlay permanently to the `PATH`, by typing the following:

```
touch ~/.bash_profile
echo 'source ~/Fast-DDS/install/setup.bash' >> ~/.bash_profile
```

CMake installation

This section explains how to compile *eProsima Fast DDS* with CMake, either *locally* or *globally*.

Local installation

1. Create a `Fast-DDS` directory where to download and build *eProsima Fast DDS* and its dependencies:

```
mkdir ~/Fast-DDS
```

2. Clone the following dependencies and compile them using CMake.

   - Foonathan memory

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/foonathan_memory_vendor.git
mkdir foonathan_memory_vendor/build
cd foonathan_memory_vendor/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/Fast-DDS/install -DBUILD_SHARED_LIBS=ON
sudo cmake --build . --target install
```

- Fast CDR

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/Fast-CDR.git
mkdir Fast-CDR/build
cd Fast-CDR/build
cmake ..  -DCMAKE_INSTALL_PREFIX=~/Fast-DDS/install
sudo cmake --build . --target install
```

3. Once all dependencies are installed, install *eProsima Fast DDS*:

```
cd ~/Fast-DDS
git clone https://github.com/eProsima/Fast-DDS.git
mkdir Fast-DDS/build
cd Fast-DDS/build
cmake ..  -DCMAKE_INSTALL_PREFIX=~/Fast-DDS/install -DCMAKE_PREFIX_PATH=~/Fast-DDS/
↪install
sudo cmake --build . --target install
```

Note: By default, *eProsima Fast DDS* does not compile tests. However, they can be activated by downloading and installing Gtest.

Global installation

To install *eProsima Fast DDS* system-wide instead of locally, remove all the flags that appear in the configuration steps of `Fast-CDR` and `Fast-DDS`, and change the first in the configuration step of `foonathan_memory_vendor` to the following:

```
-DCMAKE_INSTALL_PREFIX=/usr/local/ -DBUILD_SHARED_LIBS=ON
```

Run an application

When running an instance of an application using *eProsima Fast DDS*, it must be linked with the library where the packages have been installed, which in the case of system-wide installation is: `/usr/local/lib/` (if local installation is used, adjust for the correct directory). There are two possibilities:

- Prepare the environment locally by typing the command:

```
export LD_LIBRARY_PATH=/usr/local/lib/
```

- Add it permanently it to the `PATH`, by typing:

```
touch ~/.bash_profile
echo 'export LD_LIBRARY_PATH=/usr/local/lib/' >> ~/.bash_profile
```

6.12.2 Fast DDS-Gen installation

This section provides the instructions for installing *Fast DDS-Gen* in a Mac OS environment from sources. *Fast DDS-Gen* is a Java application that generates source code using the data types defined in an IDL file. Please refer to *Introduction* for more information.

Requirements

Fast DDS-Gen is built using Gradle. Gradle is an open-source build automation tool which requires a Java version to be executed (see Gradle-Java compatibility matrix).

Important: Even though earlier versions of Gradle support Java 8, *Fast DDS-Gen* stopped supporting Java versions previous to Java 11 since release v2.4.0.

Important: *Fast DDS-Gen* introduced support for Gradle 7 in release v2.2.0. Gradle 8 is not yet supported.

Java JDK

The JDK is a development environment for building applications and components using the Java language. Download and install it following the steps given in the Oracle website.

Note: *Fast DDS-Gen* supports Java versions from 11 to 19.

Compiling Fast DDS-Gen

In order to compile *Fast DDS-Gen*, an executable script is included in the repository which will download Gradle temporarily for the compilation step. Please, follow the steps below to build *Fast DDS-Gen*:

Note: If Fast DDS has already been installed following *Colcon installation*, skip cloning *Fast DDS-Gen*'s repository, as it can already be found under the `src` directory within the colcon workspace.

```
cd ~
git clone --recursive https://github.com/eProsima/Fast-DDS-Gen.git
cd Fast-DDS-Gen
./gradlew assemble
```

Note: In case that a supported Gradle version is already installed in the system, *Fast DDS-Gen* can also be built running directly:

```
gradle assemble
```

Contents

The `Fast-DDS-Gen` folder contains the following packages:

- `share/fastddsgen`, where the generated Java application is.
- `scripts`, containing some user friendly scripts.

> **Note:** To make these scripts accessible from any shell session and directory, add the `scripts` folder path to the `PATH` environment variable using the method described above.

6.13 QNX 7.1 installation from sources

The instructions for installing *Fast DDS library* and running examples and tests on QNX 7.1 are provided in this page. It is organized as follows:

- *Fast DDS library installation*
  - *Requirements*
  - *Build and Installation*
- *Run Examples and Tests on a QNX 7.1 Target*
  - *Move Libraries, Examples, and Tests to the QNX Target*
  - *Run DDSHelloWorldExample*
  - *Run a Test*

6.13.1 Fast DDS library installation

This section provides the instructions for installing *eProsima Fast DDS* for QNX 7.1 in a Ubuntu environment from sources. The following packages will be installed:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocator library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of *eProsima Fast DDS* library.

The *Requirements* detailed below needs to be met first.

Requirements

Users must be in a Ubuntu environment to cross-compile for QNX 7.1. It is recommended that users use Ubuntu 20.04. The installation of *eProsima Fast DDS* in a Ubuntu environment from sources requires the following tools to be installed in the system:

- *QNX SDP 7.1*
- *CMake, pip3, git, dos2unix, and automake*

QNX SDP 7.1

QNX SDP 7.1 is required to be installed in the user's Ubuntu environment. QNX SDP is QNX's Software Development Platform which contains tools and files which are needed to cross-compile for QNX.

QNX SDP Installation Guide

For the purpose of these instructions, QNX SDP 7.1 is assumed to be installed at ~/qnx710. If this is not the case, please adjust the paths accordingly.

CMake, pip3, git, dos2unix, and automake

These packages provide the tools required to install *eProsima Fast DDS* and its dependencies from command line.

```
sudo apt install cmake python3-pip git dos2unix automake
```

Build and Installation

1. Open a terminal and clone *eProsima Fast DDS*:

   ```
   git clone https://github.com/eProsima/Fast-DDS.git && cd Fast-DDS
   WORKSPACE=$PWD
   ```

2. Initialize git submodules for Asio, Fast-CDR and TinyXML2 and apply QNX patches to them:

   Note: OpenSSL is already installed in the QNX SDP 7.1.

   ```
   cd $WORKSPACE

   # Initialize git submodules
   git submodule update --init $WORKSPACE/thirdparty/asio/ $WORKSPACE/thirdparty/
   ↪fastcdr $WORKSPACE/thirdparty/tinyxml2/

   # Apply QNX patch to Asio.
   cd $WORKSPACE/thirdparty/asio
   git apply $WORKSPACE/build_qnx/qnx_patches/asio_qnx.patch

   # Apply QNX patch to Fast-CDR.
   cd $WORKSPACE/thirdparty/fastcdr
   git apply $WORKSPACE/build_qnx/qnx_patches/fastcdr_qnx.patch

   # Apply QNX patch to TinyXML2.
   # TinyXML2's CMakeLists.txt has CRLF, so use unix2dos to convert the patch to CRLF.
   cd $WORKSPACE/thirdparty/tinyxml2
   unix2dos $WORKSPACE/build_qnx/qnx_patches/tinyxml2_qnx.patch
   git apply $WORKSPACE/build_qnx/qnx_patches/tinyxml2_qnx.patch
   ```

3. Get foonathan_memory_vendor:

   ```
   cd $WORKSPACE
   git clone https://github.com/eProsima/foonathan_memory_vendor.git
   ```

4. Optional: clone GoogleTest and apply QNX patch to it:

   Note: GoogleTest is required for building Fast-DDS tests.

```
cd $WORKSPACE
git clone https://github.com/google/googletest.git && cd googletest
git checkout v1.13.0
git apply $WORKSPACE/build_qnx/qnx_patches/googletest_qnx.patch
```

5. Source the QNX environment script:

```
source ~/qnx710/qnxsdp-env.sh
```

6. Build and install Fast-DDS and its dependencies:

   Note:

   To build examples, set COMPILE_EXAMPLES to ON in $WORKSPACE/build_qnx/common.mk.
   To build tests, set EPROSIMA_BUILD_TESTS to ON in $WORKSPACE/build_qnx/common.mk.

   Note:

   All libraries will be installed to $(QNX_TARGET)/$(CPUVARDIR)/usr/lib.
   All examples will be installed to $(QNX_TARGET)/$(CPUVARDIR)/usr/examples.
   All tests will be installed to $(QNX_TARGET)/$(CPUVARDIR)/usr/bin/Fast-DDS_test.

   QNX_TARGET is where the QNX SDP 7.1 installation's target folder is.
   If QNX SDP 7.1 is installed at ~/qnx710, the QNX_TARGET will be at ~/qnx710/target/qnx7.
   CPUVARDIR is a directory for a specific target architecture e.g. aarch64le and x86_64.

   For example, libraries compiled for an aarch64 target will be at
   ~/qnx710/target/qnx7/aarch64le/usr/lib assuming QNX SDP 7.1 is installed at ~/qnx710.

```
cd $WORKSPACE/build_qnx
make install -j 4
```

6.13.2 Run Examples and Tests on a QNX 7.1 Target

Because examples and tests are compiled for QNX, they can only be run on a QNX target, not Ubuntu.

Move Libraries, Examples, and Tests to the QNX Target

1. Move the built libraries to the QNX target:

   The following steps assume that $(QNX_TARGET) is ~/qnx710/target/qnx7 and that $(CPU-VARDIR) is aarch64le. Adjust the values if this is not the case.

```
# Move Fast-CDR library to the QNX target
scp ~/qnx710/target/qnx7/aarch64le/usr/lib/libfastcdr.so* root@<target-ip-address>:/
↪usr/lib

# Move Fast-DDS library to the QNX target
scp ~/qnx710/target/qnx7/aarch64le/usr/lib/libfastrtps.so* root@<target-ip-address>
↪:/usr/lib

# Move Foonathan Memory library to the QNX target
scp ~/qnx710/target/qnx7/aarch64le/usr/lib/libfoonathan_memory* root@<target-ip-
↪address>:/usr/lib

# Move TinyXML2 library to the QNX target
scp ~/qnx710/target/qnx7/aarch64le/usr/lib/libtinyxml2.so* root@<target-ip-address>
↪:/usr/lib

# Move GoogleTest library to the QNX target
scp ~/qnx710/target/qnx7/aarch64le/usr/lib/libgtest* root@<target-ip-address>:/usr/
↪lib
scp ~/qnx710/target/qnx7/aarch64le/usr/lib/libgmock* root@<target-ip-address>:/usr/
↪lib
```

2. Move examples and tests to the QNX target:

```
# Move Fast-CDR library to the QNX target
scp -r ~/qnx710/target/qnx7/aarch64le/usr/examples root@<target-ip-address>
↪:/var

# Move Fast-DDS library to the QNX target
scp -r ~/qnx710/target/qnx7/aarch64le/usr/bin/Fast-DDS_test root@<target-ip-
↪address>:/var
```

Run DDSHelloWorldExample

1. Open a terminal and run a subscriber:

```
# ssh into the QNX target
ssh root@<target-ip-address>

# Run a subscriber
/var/examples/cpp/dds/HelloWorldExample/bin/DDSHelloWorldExample subscriber
```

2. Open another terminal and run a publisher:

```
# ssh into the QNX target
ssh root@<target-ip-address>

# Run a publisher
/var/examples/cpp/dds/HelloWorldExample/bin/DDSHelloWorldExample publisher
```

The following output will be shown in the subscriber terminal:

```
Starting
Subscriber running. Please press enter to stop the Subscriber
Subscriber matched.
Message HelloWorld 1 RECEIVED
Message HelloWorld 2 RECEIVED
Message HelloWorld 3 RECEIVED
Message HelloWorld 4 RECEIVED
Message HelloWorld 5 RECEIVED
Message HelloWorld 6 RECEIVED
Message HelloWorld 7 RECEIVED
Message HelloWorld 8 RECEIVED
Message HelloWorld 9 RECEIVED
Message HelloWorld 10 RECEIVED
Subscriber unmatched.
```

The following output will be shown for the publisher:

```
Starting
Publisher running 10 samples.
Publisher matched.
Message: HelloWorld with index: 1 SENT
Message: HelloWorld with index: 2 SENT
Message: HelloWorld with index: 3 SENT
Message: HelloWorld with index: 4 SENT
Message: HelloWorld with index: 5 SENT
Message: HelloWorld with index: 6 SENT
Message: HelloWorld with index: 7 SENT
Message: HelloWorld with index: 8 SENT
Message: HelloWorld with index: 9 SENT
Message: HelloWorld with index: 10 SENT
```

Run a Test

Because test binaries compiled for QNX cannot be run on Ubuntu, test binaries must be run on a target which is running QNX.

```
# ssh into the QNX target
ssh root@<target-ip-address>

# Run a test
cd /var/Fast-DDS_test/unittest/dds/core/entity
./EntityTests
```

The following test output for EntityTests will be shown:

```
[==========] Running 5 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 5 tests from EntityTests
[ RUN      ] EntityTests.entity_constructor
[       OK ] EntityTests.entity_constructor (0 ms)
[ RUN      ] EntityTests.entity_enable
[       OK ] EntityTests.entity_enable (0 ms)
[ RUN      ] EntityTests.entity_get_instance_handle
[       OK ] EntityTests.entity_get_instance_handle (0 ms)
[ RUN      ] EntityTests.entity_equal_operator
[       OK ] EntityTests.entity_equal_operator (0 ms)
[ RUN      ] EntityTests.get_statuscondition
[       OK ] EntityTests.get_statuscondition (0 ms)
[----------] 5 tests from EntityTests (0 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 5 tests.
```

6.14 CMake options

eProsima Fast DDS provides numerous CMake options for changing the behavior and configuration of *Fast DDS*. These options allow the user to enable/disable certain *Fast DDS* settings by defining these options to ON/OFF at the CMake execution. This section is structured as follows: first, the CMake options for the general configuration of *Fast DDS* are described; then, the options related to the third party libraries are presented; finally, the possible options for the building of *Fast DDS* tests are defined.

6.14.1 General options

The *Fast DDS* CMake options for configuring general settings are shown below, together with their description and dependency on other options.

Option	Description	Possible values	Default
EPROSIMA_INSTALLER	Creates a build for Windows binary installers. Specifically it adds to the list of components to install (CPACK_COMPONENTS_ALL) the libraries corresponding to the Microsoft Visual C++ compiler (MSVC). Setting EPROSIMA_INSTALLER to ON has the following effects on other options: • EPROSIMA_BUILD is set to ON. • BUILD_DOCUMENTATION is set to ON. • INSTALL_EXAMPLES is set to ON.	ON OFF	OFF
EPROSIMA_BUILD	Activates internal *Fast DDS* builds. It is set to ON if EPROSIMA_INSTALLER is ON. Setting EPROSIMA_BUILD to ON has the following effects on other options: • INTERNAL_DEBUG is set to ON. • COMPILE_EXAMPLES is set to ON if EPROSIMA_INSTALLER is OFF. • THIRDPARTY_fastcdr is set to ON if it was not set to FORCE. • THIRDPARTY_Asio is set to ON if it was not set to FORCE. • THIRDPARTY_TinyXML2 is set to ON if it was not set to FORCE. • THIRDPARTY_android-ifaddrs is set to ON if it was not set to FORCE. • EPROSIMA_BUILD_TESTS is set to ON if EPROSIMA_INSTALLER is OFF.	ON OFF	OFF
BUILD_SHARED_LIBS	Builds internal libraries as	ON OFF	ON

6.14.2 Log options

Fast DDS uses its own configurable **Log module** with different verbosity levels. Please, refer to *Logging* section for more information.

This module can be configured using *Fast DDS* CMake arguments regarding the following options.

Option	Description	Possible values	Default
`LOG_CONSUMER_DEFAULT`	Selects default log consumer for the logging module. `AUTO` has the same behavior as `STDOUT`. For more information, please refer to *Log consumers*.	`AUTO` `STDOUT` `STDOUTERR`	`AUTO`
`LOG_NO_INFO`	Deactivates Info Log level. If *Fast DDS* is built in debug mode for Single-Config generators, the default value will be `OFF`.	`ON` `OFF`	`ON`
`FASTDDS_ENFORCE_LOG_INFO`	Enables Info log level even on non `Debug` configurations. This option only takes action if `LOG_NO_INFO` is set to `OFF` (see *Disable Logging Module*). Mind that this may entail a significant performance hit.	`ON` `OFF`	`OFF`
`LOG_NO_WARNING`	Deactivates Warning Log level.	`ON` `OFF`	`OFF`
`LOG_NO_ERROR`	Deactivates Error Log level.	`ON` `OFF`	`OFF`
`INTERNAL_DEBUG`	Activates compilation of log messages (See *Disable Logging Module*). Moreover, `INTERNAL_DEBUG` is set to `ON` if `EPROSIMA_BUILD` is `ON`.	`ON` `OFF`	`OFF`
`ENABLE_OLD_LOG_MACROS`	Enables old log macros (See *Old Log macros disable*).	`ON` `OFF`	`ON`

6.14.3 Third-party libraries options

Fast DDS relies on the eProsima FastCDR library for serialization mechanisms. Moreover, *Fast DDS* requires two external dependencies for its proper operation: Asio and TinyXML2. Asio is a cross-platform C++ library for network and low-level I/O programming, while TinyXML2 parses the XML profile files, so *Fast DDS* can use them (see *XML profiles*). These three libraries (eProsima FastCDR, Asio and TinyXML2) can be installed by the user, or downloaded on the *Fast DDS* build. In the latter case, they are referred to as *Fast DDS* internal third-party libraries. This can be done by setting either `THIRDPARTY` or `EPROSIMA_BUILD` to `ON`.

These libraries can also be configured using *Fast DDS* CMake options.

Option	Description	Possible values	Default
THIRDPARTY_fastcdr	Activates the use of the internal Fast CDR third-party library if it is not found elsewhere in the system. FORCE activates the use of the internal Fast CDR third-party library regardless of whether it can be found elsewhere in the system. OFF deactivates the use of the internal Fast CDR third-party library. If it is not set to FORCE, it is set to ON if EPROSIMA_BUILD is ON.	ON OFF FORCE	OFF
THIRDPARTY_Asio	Activates the use of the internal Asio third-party library if it is not found elsewhere in the system. FORCE activates the use of the internal Asio third-party library regardless of whether it can be found elsewhere in the system. OFF deactivates the use of the internal Asio third-party library. If it is not set to FORCE, it is set to ON if EPROSIMA_BUILD is ON.	ON OFF FORCE	OFF
THIRDPARTY_TinyXML2	Activates the use of the internal TinyXML2 third-party library if it is not found elsewhere in the system. FORCE activates the use of the internal TinyXML2 third-party library regardless of whether it can be found elsewhere in the system. OFF deactivates the use of the internal TinyXML2 third-party library. If it is not set to FORCE, it is set to ON if EPROSIMA_BUILD is ON.	ON OFF FORCE	OFF
THIRDPARTY_android-ifaddrs	Activates the use of the internal implementation of getifaddrs() for Android. Only used if ANDROID is 1. ON activates the use of the internal android-ifaddrs third-party library if it is not found elsewhere in the system. FORCE activates the use of the internal android-ifaddrs third-party library regardless of whether it can be found elsewhere in the system. OFF deactivates the use of the internal android-ifaddrs third-party library. If it is not set to FORCE, it is set to ON if EPROSIMA_BUILD is ON.	ON OFF FORCE	OFF
THIRDPARTY	Unless they are otherwise specified, sets value of all third-party git submodules THIRDPARTY_fastcdr, THIRDPARTY_Asio, THIRDPARTY_TinyXML2, and THIRDPARTY_android-ifaddrs.	ON OFF FORCE	OFF
THIRDPARTY_UPDATE	Activates the update of all third-party git submodules.	ON OFF	ON

Note: ANDROID is a CMake environment variable that is set to 1 if the target system (CMAKE_SYSTEM_NAME) is Android.

6.14.4 Test options

eProsima Fast DDS comes with a full set of tests for continuous integration. The types of tests are: unit tests, black-box tests, performance tests, profiling tests, and XTypes tests. The building and execution of these tests is specified by the *Fast DDS* CMake options shown in the table below.

Option	Description	Possible values	Default
GTEST_INDIVIDUAL	Activates the individual building of GoogleTest tests, since *Fast DDS* tests are implemented using the GoogleTest framework. However, the test are compiled if EPROSIMA_BUILD is set to ON. Therefore, if GTEST_INDIVIDUAL is OFF and EPROSIMA_BUILD is ON, the tests are processed as a single major test.	ON OFF	OFF
FASTRTPS_API_TESTS	Activates the building of black-box tests for the verification of RTPS communications using the *Fast DDS* RTPS-layer API.	ON OFF	OFF
FASTDDS_API_TESTS	Activates the building of black-box tests for the verification of DDS communications using the *Fast DDS* DDS-layer API.	ON OFF	OFF
PERFORMANCE_TESTS	Activates the building of performance tests, except for the video test, which requires both PERFORMANCE_TESTS and VIDEO_TESTS to be set to ON.	ON OFF	OFF
PROFILING_TESTS	Activates the building of profiling tests using Valgrind.	ON OFF	OFF
EPROSIMA_BUILD_TESTS	Activates the building of black-box, unit, xtypes, RTPS communication and DDS communication tests. It is set to ON if EPROSIMA_BUILD is ON and EPROSIMA_INSTALLER is OFF.	ON OFF	OFF
VIDEO_TESTS	If PERFORMANCE_TESTS is ON, it will activate the building of video performance tests.	ON OFF	OFF
DISABLE_IPV6_TESTS	Disables IPv6 tests.	ON OFF	OFF
INSTALL_ANDROID_TESTS	Android cross-compilation only. Marks the tests for installation on the connected device/emulator.	ON OFF	OFF
ANDROID_TESTING_ROOT	Android cross-compilation only. Path on the Android device/emulator to use for installing and running the tests.	Valid Unix filesystem path string	""

6.15 Getting Started

This section defines the concepts of DDS and RTPS. It also provides a step-by-step tutorial on how to write a simple Fast DDS (formerly Fast RTPS) publish/subscribe application.

6.15.1 What is DDS?

The Data Distribution Service (DDS) is a data-centric communication protocol used for distributed software application communications. It describes the communications Application Programming Interfaces (APIs) and Communication Semantics that enable communication between data providers and data consumers.

Since it is a Data-Centric Publish Subscribe (DCPS) model, three key application entities are defined in its implementation: publication entities, which define the information-generating objects and their properties; subscription entities, which define the information-consuming objects and their properties; and configuration entities that define the types of information that are transmitted as topics, and create the publisher and subscriber with its Quality of Service (QoS) properties, ensuring the correct performance of the above entities.

DDS uses QoS to define the behavioral characteristics of DDS Entities. QoS are comprised of individual QoS policies (objects of type deriving from QoSPolicy). These are described in *Policy*.

The DCPS conceptual model

In the DCPS model, four basic elements are defined for the development of a system of communicating applications.

- **Publisher**. It is the DCPS entity in charge of the creation and configuration of the **DataWriters** it implements. The **DataWriter** is the entity in charge of the actual publication of the messages. Each one will have an assigned **Topic** under which the messages are published. See *Publisher* for further details.
- **Subscriber**. It is the DCPS Entity in charge of receiving the data published under the topics to which it subscribes. It serves one or more **DataReader** objects, which are responsible for communicating the availability of new data to the application. See *Subscriber* for further details.
- **Topic**. It is the entity that binds publications and subscriptions. It is unique within a DDS domain. Through the **TopicDescription**, it allows the uniformity of data types of publications and subscriptions. See *Topic* for further details.
- **Domain**. This is the concept used to link all publishers and subscribers, belonging to one or more applications, which exchange data under different topics. These individual applications that participate in a domain are called **DomainParticipant**. The DDS Domain is identified by a domain ID. The DomainParticipant defines the domain ID to specify the DDS domain to which it belongs. Two DomainParticipants with different IDs are not aware of each other's presence in the network. Hence, several communication channels can be created. This is applied in scenarios where several DDS applications are involved, with their respective DomainParticipants communicating with each other, but these applications must not interfere. The **DomainParticipant** acts as a container for other DCPS Entities, acts as a factory for **Publisher**, **Subscriber** and **Topic** Entities, and provides administrative services in the domain. See *Domain* for further details.

These elements are shown in the figure below.

Fig. 2: DCPS model entities in the DDS Domain.

6.15.2 What is RTPS?

The Real-Time Publish Subscribe (RTPS) protocol, developed to support DDS applications, is a publication-subscription communication middleware over best-effort transports such as UDP/IP. Furthermore, Fast DDS provides support for TCP and Shared Memory (SHM) transports.

It is designed to support both unicast and multicast communications.

At the top of RTPS, inherited from DDS, the **Domain** can be found, which defines a separate plane of communication. Several domains can coexist at the same time independently. A domain contains any number of **RTPSParticipants**, that is, elements capable of sending and receiving data. To do this, the RTPSParticipants use their **Endpoints**:

- **RTPSWriter**: Endpoint able to send data.
- **RTPSReader**: Endpoint able to receive data.

A RTPSParticipant can have any number of writer and reader endpoints.

Fig. 3: RTPS high-level architecture

Communication revolves around **Topics**, which define and label the data being exchanged. The topics do not belong to a specific participant. The participant, through the RTPSWriters, makes changes in the data published under a topic, and through the RTPSReaders receives the data associated with the topics to which it subscribes. The communication unit is called **Change**, which represents an update in the data that is written under a Topic. **RTPSReaders/RTPSWriters** register these changes on their **History**, a data structure that serves as a cache for recent changes.

In the default configuration of *eProsima Fast DDS*, when you publish a *change* through a RTPSWriter endpoint, the following steps happen behind the scenes:

1. The *change* is added to the RTPSWriter's history cache.
2. The RTPSWriter sends the change to any RTPSReaders it knows about.
3. After receiving data, RTPSReaders update their history cache with the new change.

However, Fast DDS supports numerous configurations that allow you to change the behavior of RTPSWriters/RTPSReaders. A modification in the default configuration of the RTPS entities implies a change in the data exchange flow between RTPSWriters and RTPSReaders. Moreover, by choosing Quality of Service (QoS) policies, you can affect how these history caches are managed in several ways, but the communication loop remains the same. You can continue reading section *RTPS Layer* to learn more about the implementation of the RTPS protocol in Fast DDS.

6.15.3 Writing a simple C++ publisher and subscriber application

This section details how to create a simple Fast DDS application with a publisher and a subscriber using C++ API step by step. It is also possible to self-generate a similar example to the one implemented in this section by using the *eProsima Fast DDS-Gen* tool. This additional approach is explained in *Building a publish/subscribe application*.

- *Background*
- *Prerequisites*
- *Create the application workspace*
- *Import linked libraries and its dependencies*
  - *Installation from binaries and manual installation*
  - *Colcon installation*
- *Configure the CMake project*
- *Build the topic data type*
  - *CMakeLists.txt*
- *Write the Fast DDS publisher*
  - *Examining the code*
  - *CMakeLists.txt*
- *Write the Fast DDS subscriber*
  - *Examining the code*
  - *CMakeLists.txt*
- *Putting all together*
- *Summary*
- *Next steps*

Background

DDS is a data-centric communications middleware that implements the DCPS model. This model is based on the development of a publisher, a data generating element; and a subscriber, a data consuming element. These entities communicate by means of the topic, an element that binds both DDS entities. Publishers generate information under a topic and subscribers subscribe to this same topic to receive information.

Prerequisites

First of all, you need to follow the steps outlined in the Installation Manual for the installation of *eProsima Fast DDS* and all its dependencies. You also need to have completed the steps outlined in the Installation Manual for the installation of the *eProsima Fast DDS-Gen* tool. Moreover, all the commands provided in this tutorial are outlined for a Linux environment.

Create the application workspace

The application workspace will have the following structure at the end of the project. Files `build/DDSHelloWorldPublisher` and `build/DDSHelloWorldSubscriber` are the Publisher application and Subscriber application respectively.

```
.
└── workspace_DDSHelloWorld
    ├── build
    │   ├── CMakeCache.txt
    │   ├── CMakeFiles
    │   ├── cmake_install.cmake
    │   ├── DDSHelloWorldPublisher
    │   ├── DDSHelloWorldSubscriber
    │   └── Makefile
    ├── CMakeLists.txt
    └── src
        ├── HelloWorld.cxx
        ├── HelloWorld.h
        ├── HelloWorld.idl
        ├── HelloWorldCdrAux.hpp
        ├── HelloWorldCdrAux.ipp
        ├── HelloWorldPublisher.cpp
        ├── HelloWorldPubSubTypes.cxx
        ├── HelloWorldPubSubTypes.h
        └── HelloWorldSubscriber.cpp
```

Let's create the directory tree first.

```
mkdir workspace_DDSHelloWorld && cd workspace_DDSHelloWorld
mkdir src build
```

Import linked libraries and its dependencies

The DDS application requires the Fast DDS and Fast CDR libraries. Depending on the installation procedure followed the process of making these libraries available for our DDS application will be slightly different.

Installation from binaries and manual installation

If we have followed the installation from binaries or the manual installation, these libraries are already accessible from the workspace. On Linux, the header files can be found in directories */usr/include/fastrtps/* and */usr/include/fastcdr/* for Fast DDS and Fast CDR respectively. The compiled libraries of both can be found in the directory */usr/lib/*.

Colcon installation

From a Colcon installation there are several ways to import the libraries. If the libraries need to be available just for the current session, run the following command.

```
source <path/to/Fast-DDS/workspace>/install/setup.bash
```

They can be made accessible from any session by adding the Fast DDS installation directory to your `$PATH` variable in the shell configuration files for the current user running the following command.

```
echo 'source <path/to/Fast-DDS/workspace>/install/setup.bash' >> ~/.bashrc
```

This will set up the environment after each of this user's logins.

Configure the CMake project

We will use the CMake tool to manage the building of the project. With your preferred text editor, create a new file called CMakeLists.txt and copy and paste the following code snippet. Save this file in the root directory of your workspace. If you have followed these steps, it should be *workspace_DDSHelloWorld*.

```
cmake_minimum_required(VERSION 3.20)




project(DDSHelloWorld)

# Find requirements
if(NOT fastcdr_FOUND)
    find_package(fastcdr 2 REQUIRED)
endif()

if(NOT fastrtps_FOUND)
    find_package(fastrtps 2.12 REQUIRED)
endif()

# Set C++11
include(CheckCXXCompilerFlag)
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_COMPILER_IS_CLANG OR
```

(continues on next page)

(continued from previous page)

```
        CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    check_cxx_compiler_flag(-std=c++11 SUPPORTS_CXX11)
    if(SUPPORTS_CXX11)
        add_compile_options(-std=c++11)
    else()
        message(FATAL_ERROR "Compiler doesn't support C++11")
    endif()
endif()
```

In each section we will complete this file to include the specific generated files.

Build the topic data type

eProsima Fast DDS-Gen is a Java application that generates source code using the data types defined in an Interface Description Language (IDL) file. This application can do two different things:

1. Generate C++ definitions for your custom topic.
2. Generate a functional example that uses your topic data.

It will be the former that will be followed in this tutorial. To see an example of application of the latter you can check this other *example*. See *Introduction* for further details. For this project, we will use the Fast DDS-Gen application to define the data type of the messages that will be sent by the publishers and received by the subscribers.

In the workspace directory, execute the following commands:

```
cd src && touch HelloWorld.idl
```

This creates the HelloWorld.idl file in the *src* directory. Open the file in a text editor and copy and paste the following snippet of code.

```
struct HelloWorld
{
    unsigned long index;
    string message;
};
```

By doing this we have defined the `HelloWorld` data type, which has two elements: an *index* of type `uint32_t` and a *message* of type `std::string`. All that remains is to generate the source code that implements this data type in C++11. To do this, run the following command from the `src` directory.

```
<path/to/Fast DDS-Gen>/scripts/fastddsgen HelloWorld.idl
```

This must have generated the following files:

- HelloWorld.cxx: HelloWorld type definition.
- HelloWorld.h: Header file for HelloWorld.cxx.
- HelloWorldPubSubTypes.cxx: Interface used by Fast DDS to support HelloWorld type.
- HelloWorldPubSubTypes.h: Header file for HelloWorldPubSubTypes.cxx.
- HelloWorldCdrAux.ipp: Serialization and Deserialization code for the HelloWorld type.
- HelloWorldCdrAux.hpp: Header file for HelloWorldCdrAux.ipp.

CMakeLists.txt

Include the following code snippet at the end of the CMakeList.txt file you created earlier. This includes the files we have just created.

```
message(STATUS "Configuring HelloWorld publisher/subscriber example...")
file(GLOB DDS_HELLOWORLD_SOURCES_CXX "src/*.cxx")
```

Write the Fast DDS publisher

From the *src* directory in the workspace, run the following command to download the HelloWorldPublisher.cpp file.

```
wget -O HelloWorldPublisher.cpp \
    https://raw.githubusercontent.com/eProsima/Fast-RTPS-docs/master/code/Examples/C++/
↪DDSHelloWorld/src/HelloWorldPublisher.cpp
```

This is the C++ source code for the publisher application. It is going to send 10 publications under the topic *HelloWorldTopic*.

```
1  // Copyright 2016 Proyectos y Sistemas de Mantenimiento SL (eProsima).
2  //
3  // Licensed under the Apache License, Version 2.0 (the "License");
4  // you may not use this file except in compliance with the License.
5  // You may obtain a copy of the License at
6  //
7  //     http://www.apache.org/licenses/LICENSE-2.0
8  //
9  // Unless required by applicable law or agreed to in writing, software
10 // distributed under the License is distributed on an "AS IS" BASIS,
11 // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
12 // See the License for the specific language governing permissions and
13 // limitations under the License.
14
15 /**
16  * @file HelloWorldPublisher.cpp
17  *
18  */
19
20 #include "HelloWorldPubSubTypes.h"
21
22 #include <fastdds/dds/domain/DomainParticipantFactory.hpp>
23 #include <fastdds/dds/domain/DomainParticipant.hpp>
24 #include <fastdds/dds/topic/TypeSupport.hpp>
25 #include <fastdds/dds/publisher/Publisher.hpp>
26 #include <fastdds/dds/publisher/DataWriter.hpp>
27 #include <fastdds/dds/publisher/DataWriterListener.hpp>
28
29 using namespace eprosima::fastdds::dds;
30
31 class HelloWorldPublisher
32 {
33 private:
```

(continues on next page)

(continued from previous page)

```
    HelloWorld hello_;

    DomainParticipant* participant_;

    Publisher* publisher_;

    Topic* topic_;

    DataWriter* writer_;

    TypeSupport type_;

    class PubListener : public DataWriterListener
    {
    public:

        PubListener()
            : matched_(0)
        {
        }

        ~PubListener() override
        {
        }

        void on_publication_matched(
                DataWriter*,
                const PublicationMatchedStatus& info) override
        {
            if (info.current_count_change == 1)
            {
                matched_ = info.total_count;
                std::cout << "Publisher matched." << std::endl;
            }
            else if (info.current_count_change == -1)
            {
                matched_ = info.total_count;
                std::cout << "Publisher unmatched." << std::endl;
            }
            else
            {
                std::cout << info.current_count_change
                        << " is not a valid value for PublicationMatchedStatus current count change." << std::endl;
            }
        }

        std::atomic_int matched_;

    } listener_;

```

(continues on next page)

(continued from previous page)

```
public:

    HelloWorldPublisher()
        : participant_(nullptr)
        , publisher_(nullptr)
        , topic_(nullptr)
        , writer_(nullptr)
        , type_(new HelloWorldPubSubType())
    {
    }

    virtual ~HelloWorldPublisher()
    {
        if (writer_ != nullptr)
        {
            publisher_->delete_datawriter(writer_);
        }
        if (publisher_ != nullptr)
        {
            participant_->delete_publisher(publisher_);
        }
        if (topic_ != nullptr)
        {
            participant_->delete_topic(topic_);
        }
        DomainParticipantFactory::get_instance()->delete_participant(participant_);
    }

    //!Initialize the publisher
    bool init()
    {
        hello_.index(0);
        hello_.message("HelloWorld");

        DomainParticipantQos participantQos;
        participantQos.name("Participant_publisher");
        participant_ = DomainParticipantFactory::get_instance()->create_participant(0,␣
↪participantQos);

        if (participant_ == nullptr)
        {
            return false;
        }

        // Register the Type
        type_.register_type(participant_);

        // Create the publications Topic
        topic_ = participant_->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_
↪DEFAULT);

        if (topic_ == nullptr)
```

(continues on next page)

(continued from previous page)

```
        {
            return false;
        }

        // Create the Publisher
        publisher_ = participant_->create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);

        if (publisher_ == nullptr)
        {
            return false;
        }

        // Create the DataWriter
        writer_ = publisher_->create_datawriter(topic_, DATAWRITER_QOS_DEFAULT, &
↪listener_);

        if (writer_ == nullptr)
        {
            return false;
        }
        return true;
    }

    //!Send a publication
    bool publish()
    {
        if (listener_.matched_ > 0)
        {
            hello_.index(hello_.index() + 1);
            writer_->write(&hello_);
            return true;
        }
        return false;
    }

    //!Run the Publisher
    void run(
            uint32_t samples)
    {
        uint32_t samples_sent = 0;
        while (samples_sent < samples)
        {
            if (publish())
            {
                samples_sent++;
                std::cout << "Message: " << hello_.message() << " with index: " << hello_
↪.index()
                          << " SENT" << std::endl;
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        }
    }
```

(continues on next page)

(continued from previous page)

```
};

int main(
        int argc,
        char** argv)
{
    std::cout << "Starting publisher." << std::endl;
    uint32_t samples = 10;

    HelloWorldPublisher* mypub = new HelloWorldPublisher();
    if(mypub->init())
    {
        mypub->run(samples);
    }

    delete mypub;
    return 0;
}
```

Examining the code

At the beginning of the file we have a Doxygen style comment block with the `@file` field that tells us the name of the file.

```
/**
 * @file HelloWorldPublisher.cpp
 *
 */
```

Below are the includes of the C++ headers. The first one includes the HelloWorldPubSubTypes.h file with the serialization and deserialization functions of the data type that we have defined in the previous section.

```
#include "HelloWorldPubSubTypes.h"
```

The next block includes the C++ header files that allow the use of the Fast DDS API.

- *DomainParticipantFactory*. Allows for the creation and destruction of DomainParticipant objects.
- *DomainParticipant*. Acts as a container for all other Entity objects and as a factory for the Publisher, Subscriber, and Topic objects.
- *TypeSupport*. Provides the participant with the functions to serialize, deserialize and get the key of a specific data type.
- *Publisher*. It is the object responsible for the creation of DataWriters.
- *DataWriter*. Allows the application to set the value of the data to be published under a given Topic.
- *DataWriterListener*. Allows the redefinition of the functions of the DataWriterListener.

```
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include <fastdds/dds/publisher/Publisher.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <fastdds/dds/publisher/DataWriter.hpp>
#include <fastdds/dds/publisher/DataWriterListener.hpp>
```

Next, we define the namespace that contains the eProsima Fast DDS classes and functions that we are going to use in our application.

```
using namespace eprosima::fastdds::dds;
```

The next line creates the `HelloWorldPublisher` class that implements a publisher.

```
class HelloWorldPublisher
```

Continuing with the private data members of the class, the `hello_` data member is defined as an object of the `HelloWorld` class that defines the data type we created with the IDL file. Next, the private data members corresponding to the participant, publisher, topic, DataWriter and data type are defined. The `type_` object of the *TypeSupport* class is the object that will be used to register the topic data type in the DomainParticipant.

```
private:

    HelloWorld hello_;

    DomainParticipant* participant_;

    Publisher* publisher_;

    Topic* topic_;

    DataWriter* writer_;

    TypeSupport type_;
```

Then, the `PubListener` class is defined by inheriting from the *DataWriterListener* class. This class overrides the default DataWriter listener callbacks, which allows the execution of routines in case of an event. The overridden callback *on_publication_matched()* allows the definition of a series of actions when a new DataReader is detected listening to the topic under which the DataWriter is publishing. The `info.current_count_change()` detects these changes of DataReaders that are matched to the DataWriter. This is a member in the *MatchedStatus* structure that allows tracking changes in the status of subscriptions. Finally, the `listener_` object of the class is defined as an instance of `PubListener`.

```
class PubListener : public DataWriterListener
{
public:

    PubListener()
        : matched_(0)
    {
    }

    ~PubListener() override
    {
    }

    void on_publication_matched(
```

(continues on next page)

(continued from previous page)

```
            DataWriter*,
            const PublicationMatchedStatus& info) override
    {
        if (info.current_count_change == 1)
        {
            matched_ = info.total_count;
            std::cout << "Publisher matched." << std::endl;
        }
        else if (info.current_count_change == -1)
        {
            matched_ = info.total_count;
            std::cout << "Publisher unmatched." << std::endl;
        }
        else
        {
            std::cout << info.current_count_change
                      << " is not a valid value for PublicationMatchedStatus current count␣
↪change." << std::endl;
        }
    }

    std::atomic_int matched_;

} listener_;
```

The public constructor and destructor of the `HelloWorldPublisher` class are defined below. The constructor initializes the private data members of the class to `nullptr`, with the exception of the TypeSupport object, that is initialized as an instance of the `HelloWorldPubSubType` class. The class destructor removes these data members and thus cleans the system memory.

```
HelloWorldPublisher()
    : participant_(nullptr)
    , publisher_(nullptr)
    , topic_(nullptr)
    , writer_(nullptr)
    , type_(new HelloWorldPubSubType())
{
}

virtual ~HelloWorldPublisher()
{
    if (writer_ != nullptr)
    {
        publisher_->delete_datawriter(writer_);
    }
    if (publisher_ != nullptr)
    {
        participant_->delete_publisher(publisher_);
    }
    if (topic_ != nullptr)
    {
        participant_->delete_topic(topic_);
```

(continues on next page)

(continued from previous page)

```
    }
    DomainParticipantFactory::get_instance()->delete_participant(participant_);
}
```

Continuing with the public member functions of the `HelloWorldPublisher` class, the next snippet of code defines the public publisher's initialization member function. This function performs several actions:

1. Initializes the content of the HelloWorld type `hello_` structure members.
2. Assigns a name to the participant through the QoS of the DomainParticipant.
3. Uses the *DomainParticipantFactory* to create the participant.
4. Registers the data type defined in the IDL.
5. Creates the topic for the publications.
6. Creates the publisher.
7. Creates the DataWriter with the listener previously created.

As you can see, the QoS configuration for all entities, except for the participant's name, is the default configuration (*PARTICIPANT_QOS_DEFAULT*, *PUBLISHER_QOS_DEFAULT*, *TOPIC_QOS_DEFAULT*, *DATAWRITER_QOS_DEFAULT*). The default value of the QoS of each DDS Entity can be checked in the DDS standard.

```
//!Initialize the publisher
bool init()
{
    hello_.index(0);
    hello_.message("HelloWorld");

    DomainParticipantQos participantQos;
    participantQos.name("Participant_publisher");
    participant_ = DomainParticipantFactory::get_instance()->create_participant(0,␣
↪participantQos);

    if (participant_ == nullptr)
    {
        return false;
    }

    // Register the Type
    type_.register_type(participant_);

    // Create the publications Topic
    topic_ = participant_->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_
↪DEFAULT);

    if (topic_ == nullptr)
    {
        return false;
    }

    // Create the Publisher
    publisher_ = participant_->create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);
```

(continues on next page)

(continued from previous page)

```
    if (publisher_ == nullptr)
    {
        return false;
    }

    // Create the DataWriter
    writer_ = publisher_->create_datawriter(topic_, DATAWRITER_QOS_DEFAULT, &listener_);

    if (writer_ == nullptr)
    {
        return false;
    }
    return true;
}
```

To make the publication, the public member function `publish()` is implemented. In the DataWriter's listener callback which states that the DataWriter has matched with a DataReader that listens to the publication topic, the data member `matched_` is updated. It contains the number of DataReaders discovered. Therefore, when the first DataReader has been discovered, the application starts to publish. This is simply the *writing* of a change by the DataWriter object.

```
//!Send a publication
bool publish()
{
    if (listener_.matched_ > 0)
    {
        hello_.index(hello_.index() + 1);
        writer_->write(&hello_);
        return true;
    }
    return false;
}
```

The public *run* function executes the action of publishing a given number of times, waiting for 1 second between publications.

```
//!Run the Publisher
void run(
        uint32_t samples)
{
    uint32_t samples_sent = 0;
    while (samples_sent < samples)
    {
        if (publish())
        {
            samples_sent++;
            std::cout << "Message: " << hello_.message() << " with index: " << hello_.
↪index()
                      << " SENT" << std::endl;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
}
```

Finally, the HelloWorldPublisher is initialized and run in main.

```
int main(
        int argc,
        char** argv)
{
    std::cout << "Starting publisher." << std::endl;
    uint32_t samples = 10;

    HelloWorldPublisher* mypub = new HelloWorldPublisher();
    if(mypub->init())
    {
        mypub->run(samples);
    }

    delete mypub;
    return 0;
}
```

CMakeLists.txt

Include at the end of the CMakeList.txt file you created earlier the following code snippet. This adds all the source files needed to build the executable, and links the executable and the library together.

```
add_executable(DDSHelloWorldPublisher src/HelloWorldPublisher.cpp ${DDS_HELLOWORLD_
↪SOURCES_CXX})
target_link_libraries(DDSHelloWorldPublisher fastrtps fastcdr)
```

At this point the project is ready for building, compiling and running the publisher application. From the build directory in the workspace, run the following commands.

```
cmake ..
cmake --build .
./DDSHelloWorldPublisher
```

Write the Fast DDS subscriber

From the *src* directory in the workspace, execute the following command to download the HelloWorldSubscriber.cpp file.

```
wget -O HelloWorldSubscriber.cpp \
    https://raw.githubusercontent.com/eProsima/Fast-RTPS-docs/master/code/Examples/C++/
↪DDSHelloWorld/src/HelloWorldSubscriber.cpp
```

This is the C++ source code for the subscriber application. The application runs a subscriber until it receives 10 samples under the topic *HelloWorldTopic*. At this point the subscriber stops.

```
1 // Copyright 2016 Proyectos y Sistemas de Mantenimiento SL (eProsima).
2 //
3 // Licensed under the Apache License, Version 2.0 (the "License");
4 // you may not use this file except in compliance with the License.
```

(continues on next page)

(continued from previous page)

```
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

/**
 * @file HelloWorldSubscriber.cpp
 *
 */

#include "HelloWorldPubSubTypes.h"

#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include <fastdds/dds/subscriber/Subscriber.hpp>
#include <fastdds/dds/subscriber/DataReader.hpp>
#include <fastdds/dds/subscriber/DataReaderListener.hpp>
#include <fastdds/dds/subscriber/qos/DataReaderQos.hpp>
#include <fastdds/dds/subscriber/SampleInfo.hpp>

using namespace eprosima::fastdds::dds;

class HelloWorldSubscriber
{
private:

    DomainParticipant* participant_;

    Subscriber* subscriber_;

    DataReader* reader_;

    Topic* topic_;

    TypeSupport type_;

    class SubListener : public DataReaderListener
    {
    public:

        SubListener()
            : samples_(0)
        {
        }

        ~SubListener() override
```

(continues on next page)

(continued from previous page)

```
        {
        }

        void on_subscription_matched(
                DataReader*,
                const SubscriptionMatchedStatus& info) override
        {
            if (info.current_count_change == 1)
            {
                std::cout << "Subscriber matched." << std::endl;
            }
            else if (info.current_count_change == -1)
            {
                std::cout << "Subscriber unmatched." << std::endl;
            }
            else
            {
                std::cout << info.current_count_change
                        << " is not a valid value for SubscriptionMatchedStatus current count change" << std::endl;
            }
        }

        void on_data_available(
                DataReader* reader) override
        {
            SampleInfo info;
            if (reader->take_next_sample(&hello_, &info) == ReturnCode_t::RETCODE_OK)
            {
                if (info.valid_data)
                {
                    samples_++;
                    std::cout << "Message: " << hello_.message() << " with index: " << hello_.index()
                                        << " RECEIVED." << std::endl;
                }
            }
        }

        HelloWorld hello_;

        std::atomic_int samples_;

    } listener_;

public:

    HelloWorldSubscriber()
        : participant_(nullptr)
        , subscriber_(nullptr)
        , topic_(nullptr)
        , reader_(nullptr)
```

(continues on next page)

(continued from previous page)

```
        , type_(new HelloWorldPubSubType())
    {
    }

    virtual ~HelloWorldSubscriber()
    {
        if (reader_ != nullptr)
        {
            subscriber_->delete_datareader(reader_);
        }
        if (topic_ != nullptr)
        {
            participant_->delete_topic(topic_);
        }
        if (subscriber_ != nullptr)
        {
            participant_->delete_subscriber(subscriber_);
        }
        DomainParticipantFactory::get_instance()->delete_participant(participant_);
    }

    //!Initialize the subscriber
    bool init()
    {
        DomainParticipantQos participantQos;
        participantQos.name("Participant_subscriber");
        participant_ = DomainParticipantFactory::get_instance()->create_participant(0,␣
↪participantQos);

        if (participant_ == nullptr)
        {
            return false;
        }

        // Register the Type
        type_.register_type(participant_);

        // Create the subscriptions Topic
        topic_ = participant_->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_
↪DEFAULT);

        if (topic_ == nullptr)
        {
            return false;
        }

        // Create the Subscriber
        subscriber_ = participant_->create_subscriber(SUBSCRIBER_QOS_DEFAULT, nullptr);

        if (subscriber_ == nullptr)
        {
            return false;
```

(continues on next page)

(continued from previous page)

```
        }

        // Create the DataReader
        reader_ = subscriber_->create_datareader(topic_, DATAREADER_QOS_DEFAULT, &
↪listener_);

        if (reader_ == nullptr)
        {
            return false;
        }

        return true;
    }

    //!Run the Subscriber
    void run(
        uint32_t samples)
    {
        while(listener_.samples_ < samples)
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
    }
};

int main(
        int argc,
        char** argv)
{
    std::cout << "Starting subscriber." << std::endl;
    uint32_t samples = 10;

    HelloWorldSubscriber* mysub = new HelloWorldSubscriber();
    if(mysub->init())
    {
        mysub->run(samples);
    }

    delete mysub;
    return 0;
}
```

Examining the code

Since the source code of both the publisher and subscriber applications is mostly identical, this document will focus on the main differences between them, omitting the parts of the code that have already been explained.

Following the same structure as in the publisher explanation, the first step is the includes of the C++ header files. In these, the files that include the publisher class are replaced by the subscriber class and the data writer class by the data reader class.

- *Subscriber*. It is the object responsible for the creation and configuration of DataReaders.
- *DataReader*. It is the object responsible for the actual reception of the data. It registers in the application the topic (TopicDescription) that identifies the data to be read and accesses the data received by the subscriber.
- *DataReaderListener*. This is the listener assigned to the data reader.
- *DataReaderQoS*. Structure that defines the QoS of the DataReader.
- *SampleInfo*. It is the information that accompanies each sample that is 'read' or 'taken.'

```
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/subscriber/SampleInfo.hpp>
```

The next line defines the `HelloWorldSubscriber` class that implements a subscriber.

```
class HelloWorldSubscriber
```

Starting with the private data members of the class, it is worth mentioning the implementation of the data reader listener. The private data members of the class will be the participant, the subscriber, the topic, the data reader, and the data type. As it was the case with the data writer, the listener implements the callbacks to be executed in case an event occurs. The first overridden callback of the SubListener is the *on_subscription_matched()*, which is the analog of the *on_publication_matched()* callback of the DataWriter.

```
void on_subscription_matched(
        DataReader*,
        const SubscriptionMatchedStatus& info) override
{
    if (info.current_count_change == 1)
    {
        std::cout << "Subscriber matched." << std::endl;
    }
    else if (info.current_count_change == -1)
    {
        std::cout << "Subscriber unmatched." << std::endl;
    }
    else
    {
        std::cout << info.current_count_change
                << " is not a valid value for SubscriptionMatchedStatus current count␣
↪change" << std::endl;
    }
}
```

The second overridden callback is *on_data_available()*. In this, the next received sample that the data reader can access is taken and processed to display its content. It is here that the object of the *SampleInfo* class is defined, which determines whether a sample has already been read or taken. Each time a sample is read, the counter of samples received is increased.

```
void on_data_available(
        DataReader* reader) override
{
    SampleInfo info;
    if (reader->take_next_sample(&hello_, &info) == ReturnCode_t::RETCODE_OK)
    {
        if (info.valid_data)
        {
            samples_++;
            std::cout << "Message: " << hello_.message() << " with index: " << hello_.
↪index()
                      << " RECEIVED." << std::endl;
        }
    }
}
```

The public constructor and destructor of the class is defined below.

```
HelloWorldSubscriber()
    : participant_(nullptr)
    , subscriber_(nullptr)
    , topic_(nullptr)
    , reader_(nullptr)
    , type_(new HelloWorldPubSubType())
{
}

virtual ~HelloWorldSubscriber()
{
    if (reader_ != nullptr)
    {
        subscriber_->delete_datareader(reader_);
    }
    if (topic_ != nullptr)
    {
        participant_->delete_topic(topic_);
    }
    if (subscriber_ != nullptr)
    {
        participant_->delete_subscriber(subscriber_);
    }
    DomainParticipantFactory::get_instance()->delete_participant(participant_);
}
```

Next comes the subscriber initialization public member function. This is the same as the initialization public member function defined for the `HelloWorldPublisher`. The QoS configuration for all entities, except for the participant's name, is the default QoS (*PARTICIPANT_QOS_DEFAULT*, *SUBSCRIBER_QOS_DEFAULT*, *TOPIC_QOS_DEFAULT*, *DATAREADER_QOS_DEFAULT*). The default value of the QoS of each DDS Entity can be checked in the DDS standard.

```
//!Initialize the subscriber
bool init()
{
    DomainParticipantQos participantQos;
```

(continues on next page)

(continued from previous page)

```
    participantQos.name("Participant_subscriber");
    participant_ = DomainParticipantFactory::get_instance()->create_participant(0,␣
↪participantQos);

    if (participant_ == nullptr)
    {
        return false;
    }

    // Register the Type
    type_.register_type(participant_);

    // Create the subscriptions Topic
    topic_ = participant_->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_
↪DEFAULT);

    if (topic_ == nullptr)
    {
        return false;
    }

    // Create the Subscriber
    subscriber_ = participant_->create_subscriber(SUBSCRIBER_QOS_DEFAULT, nullptr);

    if (subscriber_ == nullptr)
    {
        return false;
    }

    // Create the DataReader
    reader_ = subscriber_->create_datareader(topic_, DATAREADER_QOS_DEFAULT, &listener_);

    if (reader_ == nullptr)
    {
        return false;
    }

    return true;
}
```

The public member function `run()` ensures that the subscriber runs until all the samples have been received. This member function implements an active wait of the subscriber, with a 100ms sleep interval to ease the CPU.

```
//!Run the Subscriber
void run(
    uint32_t samples)
{
    while(listener_.samples_ < samples)
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}
```

Finally, the participant that implements a subscriber is initialized and run in main.

```
int main(
        int argc,
        char** argv)
{
    std::cout << "Starting subscriber." << std::endl;
    uint32_t samples = 10;

    HelloWorldSubscriber* mysub = new HelloWorldSubscriber();
    if(mysub->init())
    {
        mysub->run(samples);
    }

    delete mysub;
    return 0;
}
```

CMakeLists.txt

Include at the end of the CMakeList.txt file you created earlier the following code snippet. This adds all the source files needed to build the executable, and links the executable and the library together.

```
add_executable(DDSHelloWorldSubscriber src/HelloWorldSubscriber.cpp ${DDS_HELLOWORLD_
↪SOURCES_CXX})
target_link_libraries(DDSHelloWorldSubscriber fastrtps fastcdr)
```

At this point the project is ready for building, compiling and running the subscriber application. From the build directory in the workspace, run the following commands.

```
cmake ..
cmake --build .
./DDSHelloWorldSubscriber
```

Putting all together

Finally, from the build directory, run the publisher and subscriber applications from two terminals.

```
./DDSHelloWorldPublisher
./DDSHelloWorldSubscriber
```

Summary

In this tutorial you have built a publisher and a subscriber DDS application. You have also learned how to build the CMake file for source code compilation, and how to include and use the Fast DDS and Fast CDR libraries in your project.

Next steps

In the *eProsima Fast DDS* Github repository you will find more complex examples that implement DDS communication for a multitude of use cases and scenarios. You can find them here.

6.15.4 Writing a simple Python publisher and subscriber application

This section details how to create a simple Fast DDS application with a publisher and a subscriber using Python API step by step.

- *Background*
- *Prerequisites*
- *Create the application workspace*
- *Import linked libraries and its dependencies*
  - *Colcon installation*
- *Build the topic data type*
  - *CMakeLists.txt*
- *Write the Fast DDS publisher*
  - *Examining the code*
- *Write the Fast DDS subscriber*
  - *Examining the code*
- *Putting all together*
- *Summary*
- *Next steps*

Background

DDS is a data-centric communications middleware that implements the DCPS model. This model is based on the development of a publisher, a data generating element; and a subscriber, a data consuming element. These entities communicate by means of the topic, an element that binds both DDS entities. Publishers generate information under a topic and subscribers subscribe to this same topic to receive information.

Prerequisites

First of all, you need to follow the steps outlined in the Installation Manual for the installation of *eProsima Fast DDS* and all its dependencies. You also need to have completed the steps outlined in the Installation Manual for the installation of the *eProsima Fast DDS-Gen* tool. Moreover, all the commands provided in this tutorial are outlined for a Linux environment.

Create the application workspace

The application workspace will have the following structure at the end of the project. Files `HelloWorldPublisher.py` and `HelloWorldSubscriber.py` are the Publisher application and Subscriber application respectively.

```
.
├── CMakeCache.txt
├── CMakeFiles
├── CMakeLists.txt
├── HelloWorld.cxx
├── HelloWorld.h
├── HelloWorld.i
├── HelloWorld.idl
├── HelloWorld.py
├── HelloWorldCdrAux.hpp
├── HelloWorldCdrAux.ipp
├── HelloWorldPubSubTypes.cxx
├── HelloWorldPubSubTypes.h
├── HelloWorldPubSubTypes.i
├── HelloWorldPublisher.py
├── HelloWorldSubscriber.py
├── Makefile
├── _HelloWorldWrapper.so
├── cmake_install.cmake
└── libHelloWorld.so
```

Let's create the directory tree first.

```
mkdir workspace_HelloWorld && cd workspace_HelloWorld
```

Import linked libraries and its dependencies

The DDS application requires the Fast DDS, Fast CDR and Fast DDS Python bindings libraries. Depending on the installation procedure followed the process of making these libraries available for our DDS application will be slightly different.

Colcon installation

From a Colcon installation there are several ways to import the libraries. If the libraries need to be available just for the current session, run the following command.

```
source <path/to/Fast-DDS-python/workspace>/install/setup.bash
```

They can be made accessible from any session by adding the Fast DDS installation directory to your `$PATH` variable in the shell configuration files for the current user running the following command.

```
echo 'source <path/to/Fast-DDS-python/workspace>/install/setup.bash' >> ~/.bashrc
```

This will set up the environment after each of this user's logins.

Build the topic data type

eProsima Fast DDS-Gen is a Java application that generates source code using the data types defined in an Interface Description Language (IDL) file. This application can do two different things:

1. Generate C++ definitions for your custom topic.
2. Generate SWIG interface files to generate the Python bindings for your custom topic.

For this project, we will use the Fast DDS-Gen application to define the data type of the messages that will be sent by the publishers and received by the subscribers.

In the workspace directory, execute the following commands:

```
touch HelloWorld.idl
```

This creates the HelloWorld.idl file. Open the file in a text editor and copy and paste the following snippet of code.

```
struct HelloWorld
{
    unsigned long index;
    string message;
};
```

By doing this we have defined the `HelloWorld` data type, which has two elements: an *index* of type `uint32_t` and a *message* of type `std::string`. All that remains is to generate the source code that implements this data type in C++11 and the SWIG interface files for the Python bindings. To do this, run the following command.

```
<path/to/Fast DDS-Gen>/scripts/fastddsgen -python HelloWorld.idl
```

This must have generated the following files:

- HelloWorld.cxx: HelloWorld C++ type definition.
- HelloWorld.h: C++ header file for HelloWorld.cxx.
- HelloWorld.i: SWIG interface file for HelloWorld C++ type definition.

- HelloWorldPubSubTypes.cxx: C++ interface used by Fast DDS to support HelloWorld type.
- HelloWorldPubSubTypes.h: C++ header file for HelloWorldPubSubTypes.cxx.
- HelloWorldPubSubTypes.i: SWIG interface file for C++ Serialization and Deserialization code.
- HelloWorldCdrAux.ipp: C++ serialization and deserialization code for the HelloWorld type.
- HelloWorldCdrAux.hpp: C++ header file for HelloWorldCdrAux.ipp.
- CMakeLists.txt: CMake file to generate C++ source code and Python module from the SWIG interface files, compile and generate C++ libraries.
- HelloWorld.py: Python module to be imported by your Python example.

CMakeLists.txt

At this point the project is ready for building, compiling and generating Python bindings for this data type. From the workspace, run the following commands.

```
cmake .
make
```

Write the Fast DDS publisher

From the workspace, run the following command to download the HelloWorldPublisher.py file.

```
wget -O HelloWorldPublisher.py \
    https://raw.githubusercontent.com/eProsima/Fast-RTPS-docs/master/code/Examples/
↪Python/HelloWorld/HelloWorldPublisher.py
```

This is the Python source code for the publisher application. It is going to send 10 publications under the topic *HelloWorldTopic*.

```
# Copyright 2022 Proyectos y Sistemas de Mantenimiento SL (eProsima).
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
HelloWorld Publisher
"""
from threading import Condition
import time

import fastdds
import HelloWorld
```

(continues on next page)

(continued from previous page)

```

DESCRIPTION = """HelloWorld Publisher example for Fast DDS python bindings"""
USAGE = ('python3 HelloWorldPublisher.py')

class WriterListener (fastdds.DataWriterListener) :
    def __init__(self, writer) :
        self._writer = writer
        super().__init__()


    def on_publication_matched(self, datawriter, info) :
        if (0 < info.current_count_change) :
            print ("Publisher matched subscriber {}".format(info.last_subscription_
↪handle))
            self._writer._cvDiscovery.acquire()
            self._writer._matched_reader += 1
            self._writer._cvDiscovery.notify()
            self._writer._cvDiscovery.release()
        else :
            print ("Publisher unmatched subscriber {}".format(info.last_subscription_
↪handle))
            self._writer._cvDiscovery.acquire()
            self._writer._matched_reader -= 1
            self._writer._cvDiscovery.notify()
            self._writer._cvDiscovery.release()


class Writer:


    def __init__(self):
        self._matched_reader = 0
        self._cvDiscovery = Condition()
        self.index = 0

        factory = fastdds.DomainParticipantFactory.get_instance()
        self.participant_qos = fastdds.DomainParticipantQos()
        factory.get_default_participant_qos(self.participant_qos)
        self.participant = factory.create_participant(0, self.participant_qos)

        self.topic_data_type = HelloWorld.HelloWorldPubSubType()
        self.topic_data_type.setName("HelloWorld")
        self.type_support = fastdds.TypeSupport(self.topic_data_type)
        self.participant.register_type(self.type_support)

        self.topic_qos = fastdds.TopicQos()
        self.participant.get_default_topic_qos(self.topic_qos)
        self.topic = self.participant.create_topic("HelloWorldTopic", self.topic_data_
↪type.getName(), self.topic_qos)

        self.publisher_qos = fastdds.PublisherQos()
        self.participant.get_default_publisher_qos(self.publisher_qos)
```

(continues on next page)

(continued from previous page)

```
        self.publisher = self.participant.create_publisher(self.publisher_qos)

        self.listener = WriterListener(self)
        self.writer_qos = fastdds.DataWriterQos()
        self.publisher.get_default_datawriter_qos(self.writer_qos)
        self.writer = self.publisher.create_datawriter(self.topic, self.writer_qos, self.
↪listener)


    def write(self):
        data = HelloWorld.HelloWorld()
        data.message("Hello World")
        data.index(self.index)
        self.writer.write(data)
        print("Sending {message} : {index}".format(message=data.message(), index=data.
↪index()))
        self.index = self.index + 1


    def wait_discovery(self) :
        self._cvDiscovery.acquire()
        print ("Writer is waiting discovery...")
        self._cvDiscovery.wait_for(lambda : self._matched_reader != 0)
        self._cvDiscovery.release()
        print("Writer discovery finished...")


    def run(self):
        self.wait_discovery()
        for x in range(10) :
            time.sleep(1)
            self.write()
        self.delete()


    def delete(self):
        factory = fastdds.DomainParticipantFactory.get_instance()
        self.participant.delete_contained_entities()
        factory.delete_participant(self.participant)


if __name__ == '__main__':
    print('Starting publisher.')
    writer = Writer()
    writer.run()
    exit()
```

Examining the code

At the beginning of the file we import the Fast DDS Python bindings.

```
import fastdds
```

and also the Python module generated by Fast-DDS-Gen as described in *Build the topic data type* section.

```
import HelloWorld
```

Then, the `WriterListener` class is defined by inheriting from the *DataWriterListener* class. This class overrides the default DataWriter listener callbacks, which allows the execution of routines in case of an event. The overridden callback *on_publication_matched()* allows the definition of a series of actions when a new DataReader is detected listening to the topic under which the DataWriter is publishing. The `info.current_count_change()` detects these changes of DataReaders that are matched to the DataWriter. This is a member in the *MatchedStatus* structure that allows tracking changes in the status of subscriptions.

```
class WriterListener (fastdds.DataWriterListener) :
    def __init__(self, writer) :
        self._writer = writer
        super().__init__()


    def on_publication_matched(self, datawriter, info) :
        if (0 < info.current_count_change) :
            print ("Publisher matched subscriber {}".format(info.last_subscription_
↪handle))
            self._writer._cvDiscovery.acquire()
            self._writer._matched_reader += 1
            self._writer._cvDiscovery.notify()
            self._writer._cvDiscovery.release()
        else :
            print ("Publisher unmatched subscriber {}".format(info.last_subscription_
↪handle))
            self._writer._cvDiscovery.acquire()
            self._writer._matched_reader -= 1
            self._writer._cvDiscovery.notify()
            self._writer._cvDiscovery.release()
```

The next block creates the `Writer` class that implements a publisher.

```
class Writer:
```

The publisher's initialization member function of the `Writer` class are defined below. This function performs several actions:

1. Uses the *DomainParticipantFactory* to create the participant.
2. Registers the data type defined in the IDL.
3. Creates the topic for the publications.
4. Creates the publisher.
5. Creates the DataWriter with the listener previously created.

```
def __init__(self):
    self._matched_reader = 0
    self._cvDiscovery = Condition()
    self.index = 0

    factory = fastdds.DomainParticipantFactory.get_instance()
    self.participant_qos = fastdds.DomainParticipantQos()
    factory.get_default_participant_qos(self.participant_qos)
    self.participant = factory.create_participant(0, self.participant_qos)

    self.topic_data_type = HelloWorld.HelloWorldPubSubType()
    self.topic_data_type.setName("HelloWorld")
    self.type_support = fastdds.TypeSupport(self.topic_data_type)
    self.participant.register_type(self.type_support)

    self.topic_qos = fastdds.TopicQos()
    self.participant.get_default_topic_qos(self.topic_qos)
    self.topic = self.participant.create_topic("HelloWorldTopic", self.topic_data_type.
↪getName(), self.topic_qos)

    self.publisher_qos = fastdds.PublisherQos()
    self.participant.get_default_publisher_qos(self.publisher_qos)
    self.publisher = self.participant.create_publisher(self.publisher_qos)

    self.listener = WriterListener(self)
    self.writer_qos = fastdds.DataWriterQos()
    self.publisher.get_default_datawriter_qos(self.writer_qos)
    self.writer = self.publisher.create_datawriter(self.topic, self.writer_qos, self.
↪listener)
```

To make the publication, the public member function `write()` is implemented. This is simply the *writing* of a change by the DataWriter object.

```
    def write(self):
        data = HelloWorld.HelloWorld()
        data.message("Hello World")
        data.index(self.index)
        self.writer.write(data)
        print("Sending {message} : {index}".format(message=data.message(), index=data.
↪index()))
        self.index = self.index + 1
```

To detect when a DataReader has matched, the public member function `wait_discovery()` is implemented. In the DataWriter's listener callback which states that the DataWriter has matched with a DataReader that listens to the publication topic, the data member `_matched_reader` is updated. It contains the number of DataReaders discovered. Therefore, when the first DataReader has been discovered, the application starts to publish.

```
def wait_discovery(self) :
    self._cvDiscovery.acquire()
    print ("Writer is waiting discovery...")
    self._cvDiscovery.wait_for(lambda : self._matched_reader != 0)
    self._cvDiscovery.release()
    print("Writer discovery finished...")
```

The public *run* function waits until a DataReader is discovered and executes the action of publishing 10 samples.

```
def run(self):
    self.wait_discovery()
    for x in range(10) :
        time.sleep(1)
        self.write()
    self.delete()
```

Finally, the Writer is initialized and run in main.

```
if __name__ == '__main__':
    print('Starting publisher.')
    writer = Writer()
    writer.run()
    exit()
```

Write the Fast DDS subscriber

From the workspace, run the following command to download the HelloWorldSubscriber.py file.

```
wget -O HelloWorldSubscriber.py \
    https://raw.githubusercontent.com/eProsima/Fast-RTPS-docs/master/code/Examples/
↪Python/HelloWorld/HelloWorldSubscriber.py
```

This is the Python source code for the subscriber application. The application runs a subscriber until the user press *Ctrl+C* receiving samples under the topic HelloWorldTopic.

```
# Copyright 2022 Proyectos y Sistemas de Mantenimiento SL (eProsima).
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""
HelloWorld Subscriber
"""
import signal

import fastdds
import HelloWorld

DESCRIPTION = """HelloWorld Subscriber example for Fast DDS python bindings"""
USAGE = ('python3 HelloWorldSubscriber.py')

# To capture ctrl+C
```

(continues on next page)

(continued from previous page)

```
def signal_handler(sig, frame):
    print('Interrupted!')

class ReaderListener(fastdds.DataReaderListener):


    def __init__(self):
        super().__init__()


    def on_subscription_matched(self, datareader, info) :
        if (0 < info.current_count_change) :
            print ("Subscriber matched publisher {}".format(info.last_publication_
↪handle))
        else :
            print ("Subscriber unmatched publisher {}".format(info.last_publication_
↪handle))


    def on_data_available(self, reader):
        info = fastdds.SampleInfo()
        data = HelloWorld.HelloWorld()
        reader.take_next_sample(data, info)

        print("Received {message} : {index}".format(message=data.message(), index=data.
↪index()))


class Reader:


    def __init__(self):
        factory = fastdds.DomainParticipantFactory.get_instance()
        self.participant_qos = fastdds.DomainParticipantQos()
        factory.get_default_participant_qos(self.participant_qos)
        self.participant = factory.create_participant(0, self.participant_qos)

        self.topic_data_type = HelloWorld.HelloWorldPubSubType()
        self.topic_data_type.setName("HelloWorld")
        self.type_support = fastdds.TypeSupport(self.topic_data_type)
        self.participant.register_type(self.type_support)

        self.topic_qos = fastdds.TopicQos()
        self.participant.get_default_topic_qos(self.topic_qos)
        self.topic = self.participant.create_topic("HelloWorldTopic", self.topic_data_
↪type.getName(), self.topic_qos)

        self.subscriber_qos = fastdds.SubscriberQos()
        self.participant.get_default_subscriber_qos(self.subscriber_qos)
        self.subscriber = self.participant.create_subscriber(self.subscriber_qos)

        self.listener = ReaderListener()
```

(continues on next page)

(continued from previous page)

```
        self.reader_qos = fastdds.DataReaderQos()
        self.subscriber.get_default_datareader_qos(self.reader_qos)
        self.reader = self.subscriber.create_datareader(self.topic, self.reader_qos,␣
↪self.listener)


    def delete(self):
        factory = fastdds.DomainParticipantFactory.get_instance()
        self.participant.delete_contained_entities()
        factory.delete_participant(self.participant)


    def run(self):
        signal.signal(signal.SIGINT, signal_handler)
        print('Press Ctrl+C to stop')
        signal.pause()
        self.delete()


if __name__ == '__main__':
    print('Creating subscriber.')
    reader = Reader()
    reader.run()
    exit()
```

Examining the code

Since the source code of both the publisher and subscriber applications is mostly identical, this document will focus on the main differences between them, omitting the parts of the code that have already been explained.

Following the same structure as in the publisher explanation, the first step is the implementation of the data reader listener. The first overridden callback of the ReaderListener is the *on_subscription_matched()*, which is the analog of the *on_publication_matched()* callback of the DataWriter.

```
def on_subscription_matched(self, datareader, info) :
    if (0 < info.current_count_change) :
        print ("Subscriber matched publisher {}".format(info.last_publication_handle))
    else :
        print ("Subscriber unmatched publisher {}".format(info.last_publication_handle))
```

The second overridden callback is *on_data_available()*. In this, the next received sample that the data reader can access is taken and processed to display its content. It is here that the object of the *SampleInfo* class is defined, which determines whether a sample has already been read or taken.

```
def on_data_available(self, reader):
    info = fastdds.SampleInfo()
    data = HelloWorld.HelloWorld()
    reader.take_next_sample(data, info)
```

The next line defines the `Reader` class that implements a subscriber.

```
class Reader:
```

Next comes the subscriber initialization public member function. This is the same as the initialization public member function defined for the `Writer`.

```
def __init__(self):
    factory = fastdds.DomainParticipantFactory.get_instance()
    self.participant_qos = fastdds.DomainParticipantQos()
    factory.get_default_participant_qos(self.participant_qos)
    self.participant = factory.create_participant(0, self.participant_qos)

    self.topic_data_type = HelloWorld.HelloWorldPubSubType()
    self.topic_data_type.setName("HelloWorld")
    self.type_support = fastdds.TypeSupport(self.topic_data_type)
    self.participant.register_type(self.type_support)

    self.topic_qos = fastdds.TopicQos()
    self.participant.get_default_topic_qos(self.topic_qos)
    self.topic = self.participant.create_topic("HelloWorldTopic", self.topic_data_type.
↪getName(), self.topic_qos)

    self.subscriber_qos = fastdds.SubscriberQos()
    self.participant.get_default_subscriber_qos(self.subscriber_qos)
    self.subscriber = self.participant.create_subscriber(self.subscriber_qos)

    self.listener = ReaderListener()
    self.reader_qos = fastdds.DataReaderQos()
    self.subscriber.get_default_datareader_qos(self.reader_qos)
    self.reader = self.subscriber.create_datareader(self.topic, self.reader_qos, self.
↪listener)
```

The public member function `run()` ensures that the subscriber runs until the user press *Ctrl+C*.

```
def run(self):
    signal.signal(signal.SIGINT, signal_handler)
    print('Press Ctrl+C to stop')
    signal.pause()
    self.delete()
```

Finally, the participant that implements a subscriber is initialized and run in main.

```
if __name__ == '__main__':
    print('Creating subscriber.')
    reader = Reader()
    reader.run()
    exit()
```

Putting all together

Finally, from the build directory, run the publisher and subscriber applications from two terminals.

```
python3 HelloWorldPublisher.py
python3 HelloWorldSubscriber.py
```

Summary

In this tutorial you have built a Python publisher and a subscriber DDS application. You have also learned how to generate from an IDL file the specific Python module for your Topic data type.

Next steps

In the *eProsima Fast DDS* Github repository you will find more complex examples that implement DDS communication for a multitude of use cases and scenarios. You can find them here.

6.16 Library Overview

Fast DDS (formerly Fast RTPS) is an efficient and high-performance implementation of the DDS specification, a data-centric communications middleware (DCPS) for distributed application software. This section reviews the architecture, operation and key features of Fast DDS.

6.16.1 Architecture

The architecture of *Fast DDS* is shown in the figure below, where a layer model with the following different environments can be seen.

- **Application layer**. The user application that makes use of the *Fast DDS* API for the implementation of communications in distributed systems.
- **Fast DDS layer**. Robust implementation of the DDS communications middleware. It allows the deployment of one or more DDS domains in which DomainParticipants within the same domain exchange messages by publishing/subscribing under a domain topic.
- **RTPS layer**. Implementation of the Real-Time Publish-Subscribe (RTPS) protocol for interoperability with DDS applications. This layer acts an abstraction layer of the transport layer.
- **Transport Layer**. *Fast DDS* can be used over various transport protocols such as unreliable transport protocols (UDP), reliable transport protocols (TCP), or shared memory transport protocols (SHM).

Fig. 4: *Fast DDS* layer model architecture

DDS Layer

Several key elements for communication are defined in the DDS layer of *Fast DDS*. The user will create these elements in their application, thus incorporating DDS application elements and creating a data-centric communication system. *Fast DDS*, following the DDS specification, defines these elements involved in communication as **Entities**. A DDS **Entity** is any object that supports Quality of Service configuration (QoS), and that implements a listener.

- **QoS**. The mechanism by which the behavior of each of the entities is defined.
- **Listener**. The mechanism by which the entities are notified of the possible events that arise during the application's execution.

Below are listed the DDS Entities together with their description and functionality. For a more detailed explanation of each entity, their QoS, and their listeners, please refer to *DDS Layer* section.

- **Domain**. A positive integer which identifies the DDS domain. Each DomainParticipant will have an assigned DDS domain, so that DomainParticipants in the same domain can communicate, as well as isolate communications between DDS domains. This value must be given by the application developer when creating the DomainParticipants.
- **DomainParticipant**. Object containing other DDS entities such as publishers, subscribers, topics and multitopics. It is the entity that allows the creation of the previous entities it contains, as well as the configuration of their behavior.
- **Publisher**. The Publisher publishes data under a topic using a DataWriter, which writes the data to the transport. It is the entity that creates and configures the DataWriter entities it contains, and may contain one or more of them.
- **DataWriter**. It is the entity in charge of publishing messages. The user must provide a Topic when creating this entity which will be the Topic under which the data will be published. Publication is done by writing the data-objects as a change in the DataWriterHistory.
- **DataWriterHistory**. This is a list of changes to the data-objects. When the DataWriter proceeds to publish data under a specific Topic, it actually creates a *change* in this data. It is this *change* that is registered in the History. These *changes* are then sent to the DataReader that subscribes to that specific topic.
- **Subscriber**. The Subscriber subscribes to a topic using a DataReader, which reads the data from the transport. It is the entity that creates and configures the DataReader entities it contains, and may contain one or more DataReader entities.
- **DataReader**. It is the entity that subscribes to the topics for the reception of publications. The user must provide a subscription Topic when creating this entity. A DataReader receives the messages as changes in its HistoryDataReader.
- **DataReaderHistory**. It contains the *changes* in the data-objects that the DataReader receives as a result of subscribing to a certain Topic.
- **Topic**. Entity that binds Publishers' DataWriters with Subscribers' DataReaders.

RTPS layer

As mentioned above, the RTPS protocol in *Fast DDS* allows the abstraction of DDS application entities from the transport layer. According to the graph shown above, the RTPS layer has four main **Entities**.

- **RTPSDomain**. It is the extension of the DDS domain to the RTPS protocol.
- **RTPSParticipant**. Entity containing other RTPS entities. It allows the configuration and creation of the entities it contains.
- **RTPSWriter**. The source of the messages. It reads the changes written in the DataWriterHistory and transmits them to all the RTPSReaders to which it has previously matched.

- **RTPSReader**. Receiving entity of the messages. It writes the changes reported by the RTPSWriter into the DataReaderHistory.

For a more detailed explanation of each entity, their attributes, and their listeners, please refer to *RTPS Layer* section.

Transport layer

Fast DDS supports the implementation of applications over various transport protocols. Those are UDPv4, UDPv6, TCPv4, TCPv6 and Shared Memory Transport (SHM). By default, a DomainParticipant implements a UDPv4 and a SHM transport protocol. The configuration of all supported transport protocols is detailed in the *Transport Layer* section.

6.16.2 Programming and execution model

Fast DDS is concurrent and event-based. The following explains the multithreading model that governs the operation of *Fast DDS* as well as the possible events.

Concurrency and multithreading

Fast DDS implements a concurrent multithreading system. Each DomainParticipant spawns a set of threads to take care of background tasks such as logging, message reception, and asynchronous communication. This should not impact the way you use the library, i.e. the *Fast DDS* API is thread safe, so you can fearlessly call any methods on the same DomainParticipant from different threads. However, this multithreading implementation must be taken into account when external functions access to resources that are modified by threads running internally in the library. An example of this is the modified resources in the entity listener callbacks.

The complete set of threads spawned by Fast DDS is shown below. Transport related threads (marked as UDP, TCP and SHM types) are only created when the appropriate Transport is used.

Name	Type	Cardinality	Description
Event	General	One per DomainParticipant	Processes periodic and triggered time events
Discovery Server Event	General	One per DomainParticipant	Synchronizes access to the Discovery Server Database
Asynchronous Writer	General	One per enabled asynchronous flow controller Minimum 1.	Manages asynchronous writes. Even for synchronous writers, some forms of communication must be initiated in the background.
Datasharing Listener	General	One per DataReader	Listener thread that processes messages received via Datasharing
Reception	UDP	One per port	Listener thread that processes incoming UDP messages
Reception	TCP	One per port	Listener thread that processes incoming TCP messages
Keep Alive	TCP	One per port	Keep alive thread for TCP connections.
Reception	SHM	One per port	Listener thread that processes incoming messages via SHM segments
Logging	SHM	One per SHM descriptor	Stores and dumps transferred packets to a file.
Watchdog	SHM	One	Monitors health of open shared memory segments.
General Logging	Log	One	Accumulates and writes to the appropriate consumer log entries.
Security Logging	Log	One per DomainParticipant	Accumulates and writes security log entries.
Watchdog	Filewatch	One	Tracks the status of the watched file for modifications
Callback	Filewatch	One	Runs the registered callback when the watched file changes.

Some of these threads are only spawned when certain conditions are met:

- Datasharing listener thread is created only when Datasharing is in use.
- Discovery Server Event thread is only created when the DomainParticipant is configured as a Discovery Server SERVER.
- TCP keep alive thread requires the keep alive period to be configured to a value greater than zero.
- Security logging and Shared Memory packet logging threads both require certain configuration options to be enabled.
- Filewatch threads are only spawned if the *FASTDDS_ENVIRONMENT_FILE* is in use.

Regarding transport threads, Fast DDS by default uses both a UDP and a Shared Memory transport. Port configuration can be configured to suit the specific needs of the deployment, but the default configuration is to always use a metatraffic port and a unicast user traffic port. This applies both to UDP and Shared Memory since TCP does not support multicast. More information can be found at the *Default Listening Locators* page.

Event-driven architecture

There is a time-event system that enables *Fast DDS* to respond to certain conditions, as well as schedule periodic operations. Few of them are visible to the user since most are related to DDS and RTPS metadata. However, the user can define in their application periodic time-events by inheriting from the `TimedEvent` class.

6.16.3 Functionalities

Fast DDS has some added features that can be implemented and configured by the user in their application. These are outlined below.

Discovery Protocols

The discovery protocols define the mechanisms by which DataWriters publishing under a given Topic, and DataReaders subscribing to that same Topic are matched, so that they can start sharing data. This applies at any point in the communication process. *Fast DDS* provides the following discovery mechanisms:

- **Simple Discovery**. This is the default discovery mechanism, which is defined in the RTPS standard and provides compatibility with other DDS implementations. Here the DomainParticipants are discovered individually at an early stage to subsequently match the DataWriter and DataReader they implement.
- **Discovery Server**. This discovery mechanism uses a centralized discovery architecture, where servers act as hubs for meta traffic discovery.
- **Static Discovery**. This implements the discovery of DomainParticipants to each other but it is possible to skip the discovery of the entities contained in each DomainParticipant (DataReader/DataWriter) if these entities are known in advance by the remote DomainParticipants.
- **Manual Discovery**. This mechanism is only compatible with the RTPS layer. It allows the user to manually match and unmatch RTPSParticipants, RTPSWriters, and RTPSReaders using whatever external meta-information channel of its choice.

The detailed explanation and configuration of all the discovery protocols implemented in *Fast DDS* can be seen in the *Discovery* section.

Security

Fast DDS can be configured to provide secure communications by implementing pluggable security at three levels:

- Authentication of remote DomainParticipants. The **DDS:Auth:PKI-DH** plugin provides authentication using a trusted Certificate Authority (CA) and ECDSA Digital Signature Algorithms to perform the mutual authentication. It also establishes a shared secret using Elliptic Curve Diffie-Hellman (ECDH) Key Agreement protocol.
- Access control of entities. The **DDS:Access:Permissions** plugin provides access control to DomainParticipants at the DDS Domain and Topic level.
- Encryption of data. The **DDS:Crypto:AES-GCM-GMAC** plugin provides authenticated encryption using Advanced Encryption Standard (AES) in Galois Counter Mode (AES-GCM).

More information about security configuration in *Fast DDS* is available in the *Security* section.

Logging

Fast DDS provides an extensible Logging system. `Log` class is the entry point of the Logging system. It exposes three macro definitions to ease its usage: `EPROSIMA_LOG_INFO`, `EPROSIMA_LOG_WARNING` and `EPROSIMA_LOG_ERROR`. Moreover, it allows the definition of new categories, in addition to those already available (`INFO_MSG`, `WARN_MSG` and `ERROR_MSG`). It provides filtering by category using regular expressions, as well as control of the verbosity of the Logging system. Details of the possible Logging system configurations can be found in the *Logging* section.

XML profiles configuration

Fast DDS offers the possibility to make changes in its default settings by using XML profile configuration files. Thus, the behavior of the DDS Entities can be modified without the need for the user to implement any program source code or re-build an existing application.

The user has XML tags for each of the API functionalities. Therefore, it is possible to build and configure DomainParticipant profiles through the <participant> tag, or the DataWriter and DataReader profiles with the <data_writer> and <data_reader> tags respectively.

For a better understanding of how to write and use these XML profiles configuration files you can continue reading the *XML profiles* section.

Environment variables

Environment variables are those variables that are defined outside the scope of the program, through operating system functionalities. *Fast DDS* relies on environment variables so that the user can easily customize the default settings of DDS applications. Please, refer to the *Environment variables* section for a complete list and description of the environment variables affecting *Fast DDS*.

6.17 DDS Layer

eProsima Fast DDS exposes two different APIs to interact with the communication service at different levels. The main API is the Data Distribution Service (DDS) Data-Centric Publish-Subscribe (DCPS) Platform Independent Model (PIM) API, or *DDS DCPS PIM* for short, which is defined by the Data Distribution Service (DDS) version 1.4 specification, to which *Fast DDS* complies. This section is devoted to explain the main characteristics and modes-of-use of this API under *Fast DDS*, providing an in depth explanation of the five modules into which it is divided:

- *Core*: It defines the abstract classes and interfaces that are refined by the other modules. It also provides the Quality of Service (QoS) definitions, as well as support for the notification-based interaction style with the middleware.
- *Domain*: It contains the `DomainParticipant` class that acts as an entry-point of the Service, as well as a factory for many of the classes. The `DomainParticipant` also acts as a container for the other objects that make up the Service.
- *Publisher*: It describes the classes used on the publication side, including `Publisher` and `DataWriter` classes, as well as the `PublisherListener` and `DataWriterListener` interfaces.
- *Subscriber*: It describes the classes used on the subscription side, including `Subscriber` and `DataReader` classes, as well as the `SubscriberListener` and `DataReaderListener` interfaces.
- *Topic*: It describes the classes used to define communication topics and data types, including `Topic` and `TopicDescription` classes, as well as `TypeSupport`, and the `TopicListener` interface.

6.17.1 Core

This module defines the infrastructure classes and types that will be used by the other ones. It contains the definition of Entity class, QoS policies, and Statuses.

- **Entity:** An *Entity* is a DDS communication object that has a *Status* and can be configured with *Policies*.
- **Policy:** Each of the configuration objects that govern the behavior of an *Entity*.
- **Status:** Each of the objects associated with an *Entity*, whose values represent the *communication status* of that *Entity*.

Entity

Entity is the abstract base class for all the DDS entities, meaning an object that supports QoS policies, a listener, and statuses.

Types of Entities

- **DomainParticipant**: This entity is the entry-point of the Service and acts as a factory for Publishers, Subscribers, and Topics. See *DomainParticipant* for further details.
- **Publisher**: It acts as a factory that can create any number of DataWriters. See *Publisher* for further details.
- **Subscriber**: It acts as a factory that can create any number of DataReaders. See *Subscriber* for further details.
- **Topic**: This entity fits between the publication and subscription entities and acts as a channel. See *Topic* for further details.
- **DataWriter**: Is the object responsible for the data distribution. See *DataWriter* for further details.
- **DataReader**: Is the object used to access the received data. See *DataReader* for further details.

The following figure shows the hierarchy between all DDS entities:

Common Entity Characteristics

All entity types share some characteristics that are common to the concept of an entity. Those are:

Entity Identifier

Each entity is identified by a unique ID, which is shared between the DDS entity and its corresponding RTPS entity if it exists. That ID is stored on an Instance Handle object declared on Entity base class, which can be accessed using the getter function *get_instance_handle()*.

QoS policy

The behavior of each entity can be configured with a set of configuration policies. For each entity type, there is a corresponding Quality of Service (QoS) class that groups all the policies that affect said entity type. Users can create instances of these QoS classes, modify the contained policies to their needs, and use them to configure the entities, either during their creation or at a later time with the `set_qos()` function that every entity exposes (*DomainParticipant::set_qos()*, *Publisher::set_qos()*, *Subscriber::set_qos()*, *Topic::set_qos()*, *DataWriter::set_qos()*, *DataReader::set_qos()*). See *Policy* for a list of the available policies and their description. The QoS classes and the policies they contain are explained in the documentation for each entity type.

Listener

A listener is an object with functions that an entity will call in response to events. Therefore, the listener acts as an asynchronous notification system that allows the entity to notify the application about the *Status* changes in the entity.

All entity types define an abstract listener interface, which contains the callback functions that the entity will trigger to communicate the Status changes to the application. Users can implement their own listeners inheriting from these interfaces and implementing the callbacks that are needed on their application. Then they can link these listeners to each entity, either during their creation or at a later time with the `set_listener()` function that every entity exposes (*DomainParticipant::set_listener()*, *Publisher::set_listener()*, *Subscriber::set_listener()*, *Topic::set_listener()*, *DataWriter::set_listener()*, *DataReader::set_listener()*). The listener interfaces that each entity type and their callbacks are explained in the documentation for each entity type. When an event occurs it is handled by the lowest level entity with a listener that is non-null and has the corresponding callback enabled in its *StatusMask*. Higher level listeners inherit from the lower level ones as shown in the following diagram:

Fig. 5: Listeners inheritance diagram.

Note: The *on_data_on_readers()* callback intercepts messages before *on_data_available()*. This implies that if *DomainParticipantListener* is enabled, users should take into account that by default the listener uses *StatusMask::all()*. As the callback entity hierarchy is kept, the *on_data_on_readers()* is going to be called in this case. If an application wants to use *on_data_available()* instead, the corresponding bit of *StatusMask* should be disabled.

Important: Using *StatusMask::none()* when creating the *Entity* only disables the DDS standard callbacks:

- *on_sample_rejected()*
- *on_liveliness_changed()*
- *on_requested_deadline_missed()*
- *on_requested_incompatible_qos()*
- *on_data_available()*
- *on_subscription_matched()*
- *on_sample_lost()*
- *on_offered_incompatible_qos()*
- *on_offered_deadline_missed()*
- *on_liveliness_lost()*

- *on_publication_matched()*
- *on_inconsistent_topic()*
- *on_data_on_readers()*

Any callback specific to *Fast DDS* is always enabled:

- *on_participant_discovery()*
- *onParticipantAuthentication()*
- *on_subscriber_discovery()*
- *on_publisher_discovery()*
- *on_type_discovery()*
- *on_type_dependencies_reply()*
- *on_type_information_received()*
- *on_unacknowledged_sample_removed()*

Warning: Only one thread is created to listen for every listener implemented, so it is encouraged to keep listener functions simple, leaving the process of such information to the proper class.

Warning: Do not create or delete any Entity within the scope of a Listener member function, since it could lead to an undefined behavior. It is recommended instead to use the Listener class as an information channel and the upper Entity class to encapsulate such behaviour.

Status

Each entity is associated with a set of status objects whose values represent the *communication status* of that entity. The changes on these status values are the ones that trigger the invocation of the appropriate Listener callback to asynchronously inform the application. See *Status* for a list of all the status objects and a description of their content. There you can also find which status applies to which entity type.

StatusCondition

Every entity owns a StatusCondition that will be notified whenever its enabled statuses change. The StatusCondition provides the link between an Entity and a Wait-set. See section *Conditions and Wait-sets* for more information.

Enabling Entities

All the entities can be created either enabled or not enabled. By default, the factories are configured to create the entities enabled, but it can be changed using the *EntityFactoryQosPolicy* on enabled factories. A disabled factory creates disabled entities regardless of its QoS. A disabled entity has its operations limited to the following ones:

- Set/Get the entity QoS Policy.
- Set/Get the entity Listener.
- Create/Delete subentities.
- Get the Status of the entity, even if they will not change.
- Lookup operations.

Any other function called in this state will return `NOT_ENABLED`.

Policy

The Quality of Service (QoS) is used to specify the behavior of the Service, allowing the user to define how each entity will behave. To increase the flexibility of the system, the QoS is decomposed in several QoS Policies that can be configured independently. However, there may be cases where several policies conflict. Those conflicts are notified to the user through the *ReturnCodes* that the QoS setter functions returns.

Each Qos Policy has a unique ID defined in the `QosPolicyId_t` enumerator. This ID is used in some *Status* instances to identify the specific Qos Policy to which the Status refers.

There are QoS Policies that are immutable, which means that only can be specified either at the entity creation or before calling the enable operation.

Each DDS Entity has a specific set of QoS Policies that can be a mix of Standard QoS Policies, XTypes Extensions and eProsima Extensions.

Standard QoS Policies

This section explains each of the DDS standard QoS Policies:

- *DeadlineQosPolicy*
- *DestinationOrderQosPolicy*
- *DurabilityQosPolicy*
- *DurabilityServiceQosPolicy*
- *EntityFactoryQosPolicy*
- *GroupDataQosPolicy*
- *HistoryQosPolicy*
- *LatencyBudgetQosPolicy*
- *LifespanQosPolicy*
- *LivelinessQosPolicy*
- *OwnershipQosPolicy*
- *OwnershipStrengthQosPolicy*

- *PartitionQosPolicy*
- *PresentationQosPolicy*
- *ReaderDataLifecycleQosPolicy*
- *ReliabilityQosPolicy*
- *ResourceLimitsQosPolicy*
- *TimeBasedFilterQosPolicy*
- *TopicDataQosPolicy*
- *TransportPriorityQosPolicy*
- *UserDataQosPolicy*
- *WriterDataLifecycleQosPolicy*

DeadlineQosPolicy

This QoS policy raises an alarm when the frequency of new samples falls below a certain threshold. It is useful for cases where data is expected to be updated periodically (see `DeadlineQosPolicy`).

On the publishing side, the deadline defines the maximum period in which the application is expected to supply a new sample. On the subscribing side, it defines the maximum period in which new samples should be received.

For *Topics* with keys, this QoS is applied by key. Suppose that the positions of some vehicles have to be published periodically. In that case, it is possible to set the ID of the vehicle as the key of the data type and the deadline QoS to the desired publication period.

List of QoS Policy data members:

Data Member Name	Type	Default Value
`period`	`Duration_t`	`c_TimeInfinite`

Note: This QoS Policy concerns to *Topic*, *DataReader* and *DataWriter* entities.

It can be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

Compatibility Rule

To maintain the compatibility between DeadlineQosPolicy in DataReaders and DataWriters, the offered deadline period (configured on the DataWriter) must be less than or equal to the requested deadline period (configured on the DataReader), otherwise, the entities are considered to be incompatible.

The DeadlineQosPolicy must be set consistently with the *TimeBasedFilterQosPolicy*, which means that the deadline period must be higher or equal to the minimum separation.

Example

C++

```
DeadlineQosPolicy deadline;
//The DeadlineQosPolicy is default constructed with an infinite period.
//Change the period to 1 second
deadline.period.seconds = 1;
deadline.period.nanosec = 0;
```

XML

```
<data_writer profile_name="writer_xml_conf_deadline_profile">
    <qos>
        <deadline>
            <period>
                <sec>1</sec>
            </period>
        </deadline>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_deadline_profile">
    <qos>
        <deadline>
            <period>
                <sec>1</sec>
            </period>
        </deadline>
    </qos>
</data_reader>
```

DestinationOrderQosPolicy

Warning: This QoS Policy will be implemented in future releases.

Multiple *DataWriters* can send messages in the same *Topic* using the same key, and on the *DataReader* side all those messages are stored within the same instance of data (see *DestinationOrderQosPolicy*). This QoS policy controls the criteria used to determine the logical order of those messages. The behavior of the system depends on the value of the *DestinationOrderQosPolicyKind*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*DestinationOrderQosPolicyKind*	*BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS*

Note: This QoS Policy concerns to Topic, DataReader and DataWriter entities.

It cannot be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

DestinationOrderQosPolicyKind

There are two possible values (see *DestinationOrderQosPolicyKind*):

- *BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS*: This indicates that the data is ordered based on the reception time at each DataReader, which means that the last received value should be the one kept. This option may cause that each DataReader ends up with a different final value, since the DataReaders may receive the data at different times.
- *BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS*: This indicates that the data is ordered based on the DataWriter timestamp at the time the message is sent. This option guarantees the consistency of the final value.

Both options depend on the values of the *OwnershipQosPolicy* and *OwnershipStrengthQosPolicy*, meaning that if the Ownership is set to EXCLUSIVE and the last value came from a DataWriter with low ownership strength, it will be discarded.

Compatibility Rule

To maintain the compatibility between DestinationOrderQosPolicy in DataReaders and DataWriters when they have different kind values, the DataWriter kind must be higher or equal to the DataReader kind. And the order between the different kinds is:

BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS < *BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS*

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS	*BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS*	Yes
BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS	*BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS*	No
BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS	*BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS*	Yes
BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS	*BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS*	Yes

DurabilityQosPolicy

A *DataWriter* can send messages throughout a *Topic* even if there are no *DataReaders* on the network. Moreover, a DataReader that joins to the Topic after some data has been written could be interested in accessing that information (see *DurabilityQosPolicy*).

The DurabilityQoSPolicy defines how the system will behave regarding those samples that existed on the Topic before the DataReader joins. The behavior of the system depends on the value of the *DurabilityQosPolicyKind*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*DurabilityQosPolicyKind*	*VOLATILE_DURABILITY_QOS* for DataReaders *TRANSIENT_LOCAL_DURABILITY_QOS* for DataWriters

Note: This QoS Policy concerns to Topic, DataReader and DataWriter entities.

It cannot be changed on enabled entities.

Important: In order to receive past samples in the DataReader, besides setting this Qos Policy, it is required that the *ReliabilityQosPolicy* is set to *RELIABLE_RELIABILITY_QOS*.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

DurabilityQosPolicyKind

There are four possible values (see *DurabilityQosPolicyKind*):

- *VOLATILE_DURABILITY_QOS*: Past samples are ignored and a joining DataReader receives samples generated after the moment it matches.
- *TRANSIENT_LOCAL_DURABILITY_QOS*: When a new DataReader joins, its History is filled with past samples.
- *TRANSIENT_DURABILITY_QOS*: When a new DataReader joins, its History is filled with past samples, which are stored on persistent storage (see *Persistence Service*).
- *PERSISTENT_DURABILITY_QOS*: (*Not Implemented*): All the samples are stored on a permanent storage, so that they can outlive a system session.

Compatibility Rule

To maintain the compatibility between DurabilityQosPolicy in DataReaders and DataWriters when they have different kind values, the DataWriter kind must be higher or equal to the DataReader kind. And the order between the different kinds is:

VOLATILE_DURABILITY_QOS < *TRANSIENT_LOCAL_DURABILITY_QOS* < *TRANSIENT_DURABILITY_QOS* < *PERSISTENT_DURABILITY_QOS*

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
VOLATILE_DURABILITY_QOS	*VOLATILE_DURABILITY_QOS*	Yes
VOLATILE_DURABILITY_QOS	*TRANSIENT_LOCAL_DURABILITY_QOS*	No
VOLATILE_DURABILITY_QOS	*TRANSIENT_DURABILITY_QOS*	No
TRANSIENT_LOCAL_DURABILITY_QOS	*VOLATILE_DURABILITY_QOS*	Yes
TRANSIENT_LOCAL_DURABILITY_QOS	*TRANSIENT_LOCAL_DURABILITY_QOS*	Yes
TRANSIENT_LOCAL_DURABILITY_QOS	*TRANSIENT_DURABILITY_QOS*	No
TRANSIENT_DURABILITY_QOS	*VOLATILE_DURABILITY_QOS*	Yes
TRANSIENT_DURABILITY_QOS	*TRANSIENT_LOCAL_DURABILITY_QOS*	Yes
TRANSIENT_DURABILITY_QOS	*TRANSIENT_DURABILITY_QOS*	Yes

Example

C++

```
DurabilityQosPolicy durability;
//The DurabilityQosPolicy is default constructed with kind = VOLATILE_DURABILITY_QOS
//Change the kind to TRANSIENT_LOCAL_DURABILITY_QOS
durability.kind = TRANSIENT_LOCAL_DURABILITY_QOS;
```

XML

```
<data_writer profile_name="writer_xml_conf_durability_profile">
    <qos>
        <durability>
            <kind>TRANSIENT_LOCAL</kind>
         </durability>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_durability_profile">
    <qos>
        <durability>
            <kind>VOLATILE</kind>
        </durability>
    </qos>
</data_reader>
```

DurabilityServiceQosPolicy

Warning: This QoS Policy will be implemented in future releases.

This QoS Policy is used to configure the *HistoryQosPolicy* and *ResourceLimitsQosPolicy* of the fictitious *DataReader* and *DataWriter* used when the *DurabilityQosPolicy* kind is set to *TRANSIENT_DURABILITY_QOS* or *PERSISTENT_DURABILITY_QOS* (see *DurabilityServiceQosPolicy*).

Those entities are used to simulate the persistent storage. The fictitious DataReader reads the data written on the *Topic* and stores it, so that if the user DataWriter does not have the information requested by the user DataReaders, the fictitious DataWriter takes care of sending that information.

List of QoS Policy data members:

Data Member Name	Type	Default Value
service_cleanup_delay	*Duration_t*	*c_TimeZero*
history_kind	*HistoryQosPolicyKind*	*KEEP_LAST_HISTORY_QOS*
history_depth	int32_t	1
max_samples	int32_t	-1 (Length Unlimited)
max_instances	int32_t	-1 (Length Unlimited)
max_samples_per_instance	int32_t	-1 (Length Unlimited)

- *service_cleanup_delay*: It controls when the service can remove all the information regarding a data instance. That information is kept until all the following conditions are met:
  - The instance has been explicitly disposed and its InstanceState becomes *NOT_ALIVE_DISPOSED_INSTANCE_STATE*.
  - There is not any alive DataWriter writing the instance, which means that all existing writers either unregister the instance or lose their liveliness.
  - A time interval longer than the one established on the *service_cleanup_delay* has elapsed since the moment the service detected that the two previous conditions were met.
- *history_kind*: Controls the kind of the *HistoryQosPolicy* associated with the Durability Service fictitious entities.
- *history_depth*: Controls the depth of the *HistoryQosPolicy* associated with the Durability Service fictitious entities.
- *max_samples*: Controls the maximum number of samples of the *ResourceLimitsQosPolicy* associated with the Durability Service fictitious entities. This value must be higher than the maximum number of samples per instance.
- *max_instances*: Controls the maximum number of instances of the *ResourceLimitsQosPolicy* associated with the Durability Service fictitious entities.
- *max_samples_per_instance*: Controls the maximum number of samples within an instance of the *ResourceLimitsQosPolicy* associated with the Durability Service fictitious entities. This value must be lower than the maximum number of samples.

Note: This QoS Policy concerns to Topic and DataWriter entities.

It cannot be changed on enabled entities.

EntityFactoryQosPolicy

This QoS Policy controls the behavior of an *Entity* when it acts as a factory for other entities. By default, all the entities are created enabled, but if you change the value of the *autoenable_created_entities* to `false`, the new entities will be created disabled (see *EntityFactoryQosPolicy*).

List of QoS Policy data members:

Data Member Name	Type	Default Value
autoenable_created_entities	bool	true

Note: This QoS Policy concerns to *DomainParticipantFactory* (as factory for *DomainParticipant*), DomainParticipant

(as factory for *Publisher*, *Subscriber* and *Topic*), Publisher (as factory for *DataWriter*) and Subscriber (as factory for *DataReader*).

It can be changed on enabled entities, but it only affects those entities created after the change.

Example

C++

```
EntityFactoryQosPolicy entity_factory;
//The EntityFactoryQosPolicy is default constructed with autoenable_created_entities =␣
↪true
//Change it to false
entity_factory.autoenable_created_entities = false;
```

XML

This QoS Policy cannot be configured using XML for the moment.

GroupDataQosPolicy

Allows the application to attach additional information to created *Publishers* or *Subscribers*. This data is common to all *DataWriters*/*DataReaders* belonging to the Publisher/Subscriber and it is propagated by means of the built-in topics (see *GroupDataQosPolicy*).

This QoS Policy can be used in combination with DataWriter and DataReader listeners to implement a matching policy similar to the *PartitionQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
collection	std::vector<*octet*>	Empty vector

Note: This QoS Policy concerns to Publisher and Subscriber entities.

It can be changed on enabled entities.

Example

C++

```
GroupDataQosPolicy group_data;
//The GroupDataQosPolicy is default constructed with an empty collection
//Collection is a private member so you need to use getters and setters to access
//Add data to the collection
std::vector<eprosima::fastrtps::rtps::octet> vec;
vec = group_data.data_vec(); // Getter function

//Add two new octets to group data vector
```

(continues on next page)

(continued from previous page)

```
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
val = 10;
vec.push_back(val);
group_data.data_vec(vec); //Setter function
```

XML

```
<data_writer profile_name="writer_xml_conf_groupdata_profile">
    <qos>
        <groupData>
            <value>3.a</value>
        </groupData>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_groupdata_profile">
    <qos>
        <groupData>
            <value>3.a</value>
        </groupData>
    </qos>
</data_reader>
```

HistoryQosPolicy

This QoS Policy controls the behavior of the system when the value of an instance changes one or more times before it can be successfully communicated to the existing DataReader entities.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*HistoryQosPolicyKind*	*KEEP_LAST_HISTORY_QOS*
depth	int32_t	1

- *kind*: Controls if the service should deliver only the most recent values, all the intermediate values or do something in between. See *HistoryQosPolicyKind* for further details.
- *depth*: Establishes the maximum number of samples that must be kept on the history. It only has effect if the kind is set to *KEEP_LAST_HISTORY_QOS* and it needs to be consistent with the *ResourceLimitsQosPolicy*, which means that its value must be lower or equal to max_samples_per_instance.

Note: This QoS Policy concerns to Topic, DataWriter and DataReader entities.

It cannot be changed on enabled entities.

HistoryQosPolicyKind

There are two possible values (see *HistoryQosPolicyKind*):

- *KEEP_LAST_HISTORY_QOS*: The service will only attempt to keep the most recent values of the instance and discard the older ones. The maximum number of samples to keep and deliver is defined by the *depth* of the HistoryQosPolicy, which needs to be consistent with the *ResourceLimitsQosPolicy* settings. If the limit defined by *depth* is reached, the system will discard the oldest sample to make room for a new one.
- *KEEP_ALL_HISTORY_QOS*: The service will attempt to keep all the values of the instance until it can be delivered to all the existing Subscribers. If this option is selected, the depth will not have any effect, so the history is only limited by the values set in *ResourceLimitsQosPolicy*.

Consistency rule

The HistoryQos must be set consistently with the *ResourceLimitsQosPolicy*, but also other QoS as *DurabilityQosPolicy* and *ReliabilityQosPolicy*, so there are several cases to take into account:

- The *depth* is only considered if the *kind* is set to *KEEP_LAST_HISTORY_QOS*.
- The *depth* must be consistent with the *ResourceLimitsQosPolicy* settings, which means that the *depth* must be lower or equal than the *ResourceLimitsQosPolicy*'s *max_samples_per_instance*. Also, *max_samples* must be equal or higher than the product of *max_samples_per_instance* times *max_instances*.
- The *depth* cannot be lower or equal than zero. If an unlimited depth is required, please consider using *kind* as *KEEP_ALL_HISTORY_QOS*.
- Setting the *kind* as *KEEP_ALL_HISTORY_QOS* entails that limits are set by the *ResourceLimitsQosPolicy* limits (*max_samples_per_instance* prior than *max_samples*).
- In the case of the *ReliabilityQosPolicy* *ReliabilityQosPolicyKind* being set to *RELIABLE_RELIABILITY_QOS* and the *HistoryQosPolicy* *kind* being set to *KEEP_ALL_HISTORY_QOS*, when the resource limits are reached, the behavior of the service is depends on the *DurabilityQosPolicy*:
  - If the *DurabilityQosPolicy* *kind* is configured as *VOLATILE_DURABILITY_QOS*, the DataWriter *write()* call will discard the oldest sample in the history. Note that the removed sample may belong to different *instances* than the newly written one.
  - If the *DurabilityQosPolicy* *kind* is configured as *TRANSIENT_LOCAL_DURABILITY_QOS* or *TRANSIENT_DURABILITY_QOS*, the DataWriter *write()* call will be blocked until the history has space for the new sample.

Example

C++

```
HistoryQosPolicy history;
//The HistoryQosPolicy is default constructed with kind = KEEP_LAST and depth = 1.
//Change the depth to 20
history.depth = 20;
//You can also change the kind to KEEP_ALL but after that the depth will not have effect.
history.kind = KEEP_ALL_HISTORY_QOS;
```

XML

```
<topic>
    <historyQos>
        <kind>KEEP_LAST</kind> <!-- string -->
        <depth>20</depth> <!-- uint32 -->
    </historyQos>
</topic>
```

LatencyBudgetQosPolicy

Warning: This QoS Policy will be implemented in future releases.

This QoS Policy specifies the maximum acceptable delay from the time the data is written until the data is inserted on the DataReader History and notified of the fact. That delay by default is set to 0 in order to optimize the internal operations (see *LatencyBudgetQosPolicy*).

List of QoS Policy data members:

Data Member Name	Type	Default Value
duration	*Duration_t*	*c_TimeZero*

Note: This QoS Policy concerns to *Topic*, *DataWriter* and *DataReader* entities.

It can be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

Compatibility Rule

To maintain the compatibility between LatencyBudgetQosPolicy in DataReaders and DataWriters, the DataWriter duration must be lower or equal to the DataReader duration.

LifespanQosPolicy

Each data sample written by a *DataWriter* has an associated expiration time beyond which the data is removed from the DataWriter and DataReader history as well as from the transient and persistent information caches (see *LifespanQosPolicy*).

By default, the *duration* is infinite, which means that there is not a maximum duration for the validity of the samples written by the DataWriter.

The expiration time is computed by adding the *duration* to the source timestamp, which can be calculated automatically if *write()* member function is called or supplied by the application by means of `write_w_timestamp()` member function. The DataReader is allowed to use the reception timestamp instead of the source timestamp.

List of QoS Policy data members:

Data Member Name	Type	Default Value
duration	*Duration_t*	*c_TimeInfinite*

Note: This QoS Policy concerns to *Topic*, *DataReader* and *DataWriter* entities.

It can be changed on enabled entities.

Example

C++

```
LifespanQosPolicy lifespan;
//The LifespanQosPolicy is default constructed with duration set to infinite.
//Change the duration to 5 s
lifespan.duration = {5, 0};
```

XML

```
<data_writer profile_name="writer_xml_conf_lifespan_profile">
    <qos>
        <lifespan>
            <duration>
                <sec>5</sec>
            </duration>
        </lifespan>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_lifespan_profile">
    <qos>
        <lifespan>
            <duration>
                <sec>5</sec>
            </duration>
        </lifespan>
    </qos>
</data_reader>
```

LivelinessQosPolicy

This QoS Policy controls the mechanism used by the service to ensure that a particular entity on the network is still alive. There are different settings that allow distinguishing between applications where data is updated periodically and applications where data is changed sporadically. It also allows customizing the application regarding the kind of failures that should be detected by the liveliness mechanism (see *LivelinessQosPolicy*).

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*LivelinessQosPolicyKind*	*AUTOMATIC_LIVELINESS_QOS*
lease_duration	*Duration_t*	*c_TimeInfinite*
announcement_period	*Duration_t*	*c_TimeInfinite*

- *kind*: This data member establishes if the service needs to assert the liveliness automatically or if it needs to wait until the liveliness is asserted by the publishing side. See *LivelinessQosPolicyKind* for further details.
- *lease_duration*: Amount of time to wait since the last time the DataWriter asserts its liveliness to consider that it is no longer alive.
- *announcement_period*: Amount of time between consecutive liveliness messages sent by the DataWriter. This data member only takes effect if the kind is *AUTOMATIC_LIVELINESS_QOS* or *MANUAL_BY_PARTICIPANT_LIVELINESS_QOS* and needs to be lower than the *lease_duration*.

Note: This QoS Policy concerns to *Topic*, *DataReader* and *DataWriter* entities.

It cannot be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

LivelinessQosPolicyKind

There are three possible values (see *LivelinessQosPolicyKind*):

- *AUTOMATIC_LIVELINESS_QOS*: The service takes the responsibility for renewing the leases at the required rates, as long as the local process where the participant is running and the link connecting it to remote participants exists, the entities within the remote participant will be considered alive. This kind is suitable for applications that only need to detect whether a remote application is still running.
- The two *Manual* modes require that the application on the publishing side asserts the liveliness periodically before the lease_duration timer expires. Publishing any new data value implicitly asserts the DataWriter's liveliness, but it can be done explicitly by calling the *assert_liveliness* member function.
  - *MANUAL_BY_PARTICIPANT_LIVELINESS_QOS*: If one of the entities in the publishing side asserts its liveliness, the service deduces that all other entities within the same DomainParticipant are also alive.
  - *MANUAL_BY_TOPIC_LIVELINESS_QOS*: This mode is more restrictive and requires that at least one instance within the DataWriter is asserted to consider that the DataWriter is alive.

Compatibility Rule

To maintain the compatibility between LivelinessQosPolicy in DataReaders and DataWriters, the DataWriter kind must be higher or equal to the DataReader kind. And the order between the different kinds is:

```
|AUTOMATIC_LIVELINESS_QOS-api| < |MANUAL_BY_PARTICIPANT_LIVELINESS_QOS-api| < |MANUAL_BY_
↪TOPIC_LIVELINESS_QOS-api|
```

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
AUTOMATIC_LIVELINESS_QOS	*AUTOMATIC_LIVELINESS_QOS*	Yes
AUTOMATIC_LIVELINESS_QOS	*MANUAL_BY_PARTICIPANT_LIVELINESS_QOS*	No
AUTOMATIC_LIVELINESS_QOS	*MANUAL_BY_TOPIC_LIVELINESS_QOS*	No
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	*AUTOMATIC_LIVELINESS_QOS*	Yes
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	*MANUAL_BY_PARTICIPANT_LIVELINESS_QOS*	Yes
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	*MANUAL_BY_TOPIC_LIVELINESS_QOS*	No
MANUAL_BY_TOPIC_LIVELINESS_QOS	*AUTOMATIC_LIVELINESS_QOS*	Yes
MANUAL_BY_TOPIC_LIVELINESS_QOS	*MANUAL_BY_PARTICIPANT_LIVELINESS_QOS*	Yes
MANUAL_BY_TOPIC_LIVELINESS_QOS	*MANUAL_BY_TOPIC_LIVELINESS_QOS*	Yes

Additionally, the *lease_duration* of the DataWriter must not be greater than the *lease_duration* of the DataReader.

Example

C++

```
LivelinessQosPolicy liveliness;
//The LivelinessQosPolicy is default constructed with kind = AUTOMATIC
//Change the kind to MANUAL_BY_PARTICIPANT
liveliness.kind = MANUAL_BY_PARTICIPANT_LIVELINESS_QOS;
//The LivelinessQosPolicy is default constructed with lease_duration set to infinite
//Change the lease_duration to 1 second
liveliness.lease_duration = {1, 0};
//The LivelinessQosPolicy is default constructed with announcement_period set to infinite
//Change the announcement_period to 1 ms
liveliness.announcement_period = {0, 1000000};
```

XML

```
<data_writer profile_name="writer_xml_conf_liveliness_profile">
    <qos>
        <liveliness>
            <announcement_period>
                <nanosec>1000000</nanosec>
            </announcement_period>
            <lease_duration>
                <sec>1</sec>
            </lease_duration>
            <kind>AUTOMATIC</kind>
        </liveliness>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_liveliness_profile">
    <qos>
        <liveliness>
            <lease_duration>
                <sec>1</sec>
```

(continues on next page)

(continued from previous page)

```
            </lease_duration>
            <kind>AUTOMATIC</kind>
        </liveliness>
    </qos>
</data_reader>
```

OwnershipQosPolicy

This QoS Policy specifies whether it is allowed for multiple DataWriters to update the same instance of data, and if so, how these modifications should be arbitrated (see *OwnershipQosPolicy*).

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*OwnershipQosPolicyKind*	*SHARED_OWNERSHIP_QOS*

Note: This QoS Policy concerns to *Topic*, *DataReader* and *DataWriter* entities.

It cannot be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

OwnershipQosPolicyKind

There are two possible values (see *OwnershipQosPolicyKind*):

- *SHARED_OWNERSHIP_QOS*: This option indicates that the service does not enforce unique ownership for each instance. In this case, multiple DataWriters are allowed to update the same data instance and all the updates are made available to the existing DataReaders. Those updates are also subject to the *TimeBasedFilterQosPolicy* or *HistoryQosPolicy* settings, so they can be filtered.
- *EXCLUSIVE_OWNERSHIP_QOS*: This option indicates that each instance can only be updated by one DataWriter, meaning that at any point in time a single DataWriter owns each instance and is the only one whose modifications will be visible for the existing DataReaders. The owner can be changed dynamically according to the highest *strength* between the alive DataWriters, which has not violated the deadline contract concerning the data instances. That *strength* can be changed using the *OwnershipStrengthQosPolicy*. In case two DataWriters have the same *strength* value, the DataWriter with a lower *GUID* value would be the owner of the topic.

Compatibility Rule

To maintain the compatibility between OwnershipQosPolicy in *DataReaders* and *DataWriters*, the DataWriter kind must be equal to the DataReader kind.

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
SHARED_OWNERSHIP_QOS	*SHARED_OWNERSHIP_QOS*	Yes
SHARED_OWNERSHIP_QOS	*EXCLUSIVE_OWNERSHIP_QOS*	No
EXCLUSIVE_OWNERSHIP_QOS	*SHARED_OWNERSHIP_QOS*	No
EXCLUSIVE_OWNERSHIP_QOS	*EXCLUSIVE_OWNERSHIP_QOS*	Yes

Example

C++

```
OwnershipQosPolicy ownership;
//The OwnershipQosPolicy is default constructed with kind = SHARED.
//Change the kind to EXCLUSIVE
ownership.kind = EXCLUSIVE_OWNERSHIP_QOS;
```

XML

```
<data_writer profile_name="writer_xml_conf_ownership_profile">
    <qos>
      <ownership>
            <kind>EXCLUSIVE</kind>
      </ownership>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_ownership_profile">
    <qos>
        <ownership>
            <kind>EXCLUSIVE</kind>
        </ownership>
    </qos>
</data_reader>
```

OwnershipStrengthQosPolicy

This QoS Policy specifies the value of the *strength* used to arbitrate among multiple DataWriters that attempt to modify the same data instance. It is only applicable if the *OwnershipQosPolicy* kind is set to *EXCLUSIVE_OWNERSHIP_QOS*. See *OwnershipStrengthQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
value	uint32_t	0

Note: This QoS Policy concerns to DataWriter entities.

It can be changed on enabled entities.

Example

C++

```
OwnershipStrengthQosPolicy ownership_strength;
//The OwnershipStrengthQosPolicy is default constructed with value 0
//Change the strength to 10
ownership_strength.value = 10;
```

XML

```
<data_writer profile_name="writer_xml_conf_ownership_strength_profile">
    <qos>
        <ownershipStrength>
            <value>10</value>
        </ownershipStrength>
    </qos>
</data_writer>
```

PartitionQosPolicy

This Qos Policy allows the introduction of a logical partition inside the physical partition introduced by a domain. For a DataReader to see the changes made by a DataWriter, not only the Topic must match, but also they have to share at least one logical partition (see *PartitionQosPolicy*).

The empty string is also considered as a valid partition and it matches with other partition names using the same rules of string matching and regular-expression matching used for any other partition name.

List of QoS Policy data members:

Data Member Name	Type	Default Value
max_size	uint32_t	0 (Length Unlimited)
names	*SerializedPayload_t*	Empty List

- *max_size*: Maximum size for the list of partition names.
- *names*: List of partition names.

Note: This QoS Policy concerns to Publisher and Subscriber entities.

Partitions can also be explicitly defined at the endpoint level to override this configuration. Information to do so can be found *here*.

It can be changed on enabled Publishers and Subscribers.

Example

C++

```
PartitionQosPolicy partitions;
//The PartitionsQosPolicy is default constructed with max_size = 0.
//Max_size is a private member so you need to use getters and setters to access
//Change the max_size to 20
partitions.set_max_size(20); //Setter function
//The PartitionsQosPolicy is default constructed with an empty list of partitions
//Partitions is a private member so you need to use getters and setters to access
//Add new partitions
std::vector<std::string> part = partitions.names(); //Getter function
part.push_back("part1");
part.push_back("part2");
partitions.names(part); //Setter function
```

XML

```
<data_writer profile_name="pub_partition_example">
    <qos>
        <partition>
            <names>
                <name>part1</name>
                <name>part2</name>
            </names>
        </partition>
    </qos>
</data_writer>

<data_reader profile_name="sub_partition_example">
    <qos>
        <partition>
            <names>
                <name>part1</name>
                <name>part2</name>
            </names>
        </partition>
    </qos>
</data_reader>
```

PresentationQosPolicy

Warning: This QoS Policy will be implemented in future releases.

This QoS Policy specifies how the samples representing changes to data instances are presented to the subscribing application. It controls the extent to which changes to data instances can be made dependent on each other, as well as the kind of dependencies that can be propagated and maintained. See *PresentationQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
access_scope	*PresentationQosPolicyAccessScopeKind*	INSTANCE_PRESENTATION_QOS
coherent_access	bool	false
ordered_access	bool	false

- access_scope: Determines the largest scope spanning the entities for which the order and coherency can be preserved. See *PresentationQosPolicyAccessScopeKind* for further details.
- coherent_access: Controls whether the service will preserve grouping of changes made on the publishing side, such that they are received as a unit on the subscribing side.
- ordered_access: Controls whether the service supports the ability of the subscriber to see changes in the same order as they occurred on the publishing side.

Note: This QoS Policy concerns to *Publisher* and *Subscriber* entities.

It cannot be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

PresentationQosPolicyAccessScopeKind

There are three possible values, which have different behaviors depending on the values of coherent_access and ordered_access variables (see PresentationQosPolicyAccessScopeKind):

- INSTANCE_PRESENTATION_QOS: The changes to a data instance do not need to be coherent nor ordered with respect to the changes to any other instance, which means that the order and coherent changes apply to each instance separately.
  - Enabling the *coherent_access*, in this case, has no effect on how the subscriber can access the data as the scope is limited to each instance, changes to separate instances are considered independent and thus cannot be grouped by a coherent change.
  - Enabling the *ordered_access*, in this case, only affects to the changes within the same instance. Therefore, the changes made to two instances are not necessarily seen in the order they occur even if the same application thread and DataWriter made them.
- TOPIC_PRESENTATION_QOS: The scope spans to all the instances within the same DataWriter.
  - Enabling the *coherent_access* makes that the grouping made with changes within the same DataWriter will be available as coherent with respect to other changes to instances in that DataWriter, but will not be grouped with changes made to instances belonging to different DataWriters.
  - Enabling the *ordered_access* means that the changes made by a single DataWriter are made available to the subscribers in the same order that they occur, but the changes made to instances through different DataWriters are not necessarily seen in order.
- GROUP_PRESENTATION_QOS: The scope spans to all the instances belonging to DataWriters within the same Publisher.
  - Enabling the *coherent_access*, means that the coherent changes made to instances through DataWriters attached to a common Publisher are made available as a unit to remote subscribers.
  - Enabling the *ordered_access* with this scope makes that the changes done by any of the DataWriters attached to the same Publisher are made available to the subscribers in the same order they occur.

Compatibility Rule

To maintain the compatibility between PresentationQosPolicy in DataReaders and DataWriters, the Publisher *access_scope* must be higher or equal to the Subscriber *access_scope*. And the order between the different access scopes is:

```
|INSTANCE_PRESENTATION_QOS-api| < |TOPIC_PRESENTATION_QOS-api| < |GROUP_PRESENTATION_QOS-
↪api|
```

Table with the possible combinations:

Publisher scope	Subscriber scope	Compatibility
INSTANCE_PRESENTATION_QOS	*INSTANCE_PRESENTATION_QOS*	Yes
INSTANCE_PRESENTATION_QOS	*TOPIC_PRESENTATION_QOS*	No
INSTANCE_PRESENTATION_QOS	*GROUP_PRESENTATION_QOS*	No
TOPIC_PRESENTATION_QOS	*INSTANCE_PRESENTATION_QOS*	Yes
TOPIC_PRESENTATION_QOS	*TOPIC_PRESENTATION_QOS*	Yes
TOPIC_PRESENTATION_QOS	*GROUP_PRESENTATION_QOS*	No
GROUP_PRESENTATION_QOS	*INSTANCE_PRESENTATION_QOS*	Yes
GROUP_PRESENTATION_QOS	*TOPIC_PRESENTATION_QOS*	Yes
GROUP_PRESENTATION_QOS	*GROUP_PRESENTATION_QOS*	Yes

Additionally, the coherent_access and ordered_access of the Subscriber can only be enabled if they are also enabled on the Publisher.

ReaderDataLifecycleQosPolicy

Warning: This QoS Policy will be implemented in future releases.

This QoS Policy specifies the behavior of the *DataReader* with respect to the lifecycle of the data instances it manages, that is, the instances that have been received and for which the DataReader maintains some internal resources. The DataReader maintains the samples that have not been taken by the application, subject to the constraints imposed by *HistoryQosPolicy* and *ResourceLimitsQosPolicy*. See *ReaderDataLifecycleQosPolicy*.

Under normal circumstances, the DataReader can only reclaim the resources associated with data instances if there are no writers and all the samples have been taken. But this fact can cause problems if the application does not take those samples as the service will prevent the DataReader from reclaiming the resources and they will remain in the DataReader indefinitely. This QoS exist to avoid that situation.

List of QoS Policy data members:

Data Member Name	Type	Default Value
autopurge_no_writer_samples_delay	*Duration_t*	*c_TimeInfinite*
autopurge_disposed_samples_delay	*Duration_t*	*c_TimeInfinite*

- *autopurge_no_writer_samples_delay*: Defines the maximum duration the DataReader must retain the information regarding an instance once its *instance_state* becomes *NOT_ALIVE_NO_WRITERS_INSTANCE_STATE*. After this time elapses, the DataReader purges all the internal information of the instance, including the untaken samples that will be lost.

- *autopurge_disposed_samples_delay*: Defines the maximum duration the DataReader must retain the information regarding an instance once its *instance_state* becomes *NOT_ALIVE_DISPOSED_INSTANCE_STATE*. After this time elapses, the DataReader purges all the samples for the instance.

Note: This QoS Policy concerns to DataReader entities.

It can be changed on enabled entities.

ReliabilityQosPolicy

This QoS Policy indicates the level of reliability offered and requested by the service. See *ReliabilityQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*ReliabilityQosPolicyKind*	*BEST_EFFORT_RELIABILITY_QOS* for DataReaders *RELIABLE_RELIABILITY_QOS* for DataWriters
max_blocking_time	*Duration_t*	100 ms

- *kind*: Specifies the behavior of the service regarding delivery of the samples. See *ReliabilityQosPolicyKind* for further details.
- *max_blocking_time*: Configures the maximum duration that the write operation can be blocked.

Note: This QoS Policy concerns to *Topic*, *DataWriter* and *DataReader* entities.

It cannot be changed on enabled entities.

Important: Setting this QoS Policy to *BEST_EFFORT_RELIABILITY_QOS* affects to the *DurabilityQosPolicy*, making the endpoints behave as *VOLATILE_DURABILITY_QOS*.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

ReliabilityQosPolicyKind

There are two possible values ():

- *BEST_EFFORT_RELIABILITY_QOS*: It indicates that it is acceptable not to retransmit the missing samples, so the messages are sent without waiting for an arrival confirmation. Presumably new values for the samples are generated often enough that it is not necessary to re-send any sample. However, the data samples sent by the same DataWriter will be stored in the DataReader history in the same order they occur. In other words, even if the DataReader misses some data samples, an older value will never overwrite a newer value.
- *RELIABLE_RELIABILITY_QOS*: It indicates that the service will attempt to deliver all samples of the DataWriter's history expecting an arrival confirmation from the DataReader. The data samples sent by the same DataWriter cannot be made available to the DataReader if there are previous samples that have not been received

yet. The service will retransmit the lost data samples in order to reconstruct a correct snapshot of the DataWriter history before it is accessible by the DataReader.

This option may block the write operation, hence the *max_blocking_time* is set that will unblock it once the time expires. But if the *max_blocking_time* expires before the data is sent, the write operation will return an error.

Compatibility Rule

To maintain the compatibility between ReliabilityQosPolicy in DataReaders and DataWriters, the DataWriter kind must be higher or equal to the DataReader kind. And the order between the different kinds is:

```
|BEST_EFFORT_RELIABILITY_QOS-api| < |RELIABLE_RELIABILITY_QOS-api|
```

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
BEST_EFFORT_RELIABILITY_QOS	*BEST_EFFORT_RELIABILITY_QOS*	Yes
BEST_EFFORT_RELIABILITY_QOS	*RELIABLE_RELIABILITY_QOS*	No
RELIABLE_RELIABILITY_QOS	*BEST_EFFORT_RELIABILITY_QOS*	Yes
RELIABLE_RELIABILITY_QOS	*RELIABLE_RELIABILITY_QOS*	Yes

Example

C++

```
ReliabilityQosPolicy reliability;
//The ReliabilityQosPolicy is default constructed with kind = BEST_EFFORT
//Change the kind to RELIABLE
reliability.kind = RELIABLE_RELIABILITY_QOS;
//The ReliabilityQosPolicy is default constructed with max_blocking_time = 100ms
//Change the max_blocking_time to 1s
reliability.max_blocking_time = {1, 0};
```

XML

```
<data_writer profile_name="writer_xml_conf_reliability_profile">
    <qos>
        <reliability>
            <kind>RELIABLE</kind>
            <max_blocking_time>
                <sec>1</sec>
            </max_blocking_time>
        </reliability>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_reliability_profile">
    <qos>
        <reliability>
            <kind>BEST_EFFORT</kind>
        </reliability>
```

(continues on next page)

(continued from previous page)

```
    </qos>
</data_reader>
```

ResourceLimitsQosPolicy

This QoS Policy controls the resources that the service can use in order to meet the requirements imposed by the application and other QoS Policies. See *ResourceLimitsQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
max_samples	int32_t	5000
max_instances	int32_t	10
max_samples_per_instance	int32_t	400
allocated_samples	int32_t	100
extra_samples	int32_t	1

- *max_samples*: Controls the maximum number of samples that the DataWriter or DataReader can manage across all the instances associated with it. In other words, it represents the maximum samples that the middleware can store for a DataReader or DataWriter. **Value 0 means infinite resources.**
- *max_instances*: Controls the maximum number of instances that a DataWriter or DataReader can manage. **Value 0 means infinite resources.**
- *max_samples_per_instance*: Controls the maximum number of samples within an instance that the DataWriter or DataReader can manage. **Value 0 means infinite resources.**
- *allocated_samples*: States the number of samples that will be allocated on initialization.
- *extra_samples*: States the number of extra samples that will be allocated on the pool, so the maximum number of samples on the pool will be *max_samples* plus *extra_samples*. These extra samples act as a reservoir of samples even when the history is full.

Note: This QoS Policy concerns to Topic, DataWriter and DataReader entities.

It cannot be changed on enabled entities.

Consistency Rule

To maintain the consistency within the ResourceLimitsQosPolicy, the values of the data members must follow the next conditions:

- The value of *max_samples* must be higher or equal to the value of *max_samples_per_instance*.
- The value established for the *HistoryQosPolicy depth* must be lower or equal to the value stated for *max_samples_per_instance*.

Example

C++

```
ResourceLimitsQosPolicy resource_limits;
//The ResourceLimitsQosPolicy is default constructed with max_samples = 5000
//Change max_samples to 200
resource_limits.max_samples = 200;
//The ResourceLimitsQosPolicy is default constructed with max_instances = 10
//Change max_instances to 20
resource_limits.max_instances = 20;
//The ResourceLimitsQosPolicy is default constructed with max_samples_per_instance = 400
//Change max_samples_per_instance to 100 as it must be lower than max_samples
resource_limits.max_samples_per_instance = 100;
//The ResourceLimitsQosPolicy is default constructed with allocated_samples = 100
//Change allocated_samples to 50
resource_limits.allocated_samples = 50;
```

XML

```
<data_writer profile_name="writer_xml_conf_resource_limits_profile">
    <topic>
        <resourceLimitsQos>
            <max_samples>200</max_samples>
            <max_instances>20</max_instances>
            <max_samples_per_instance>100</max_samples_per_instance>
            <allocated_samples>50</allocated_samples>
        </resourceLimitsQos>
    </topic>
</data_writer>

<data_reader profile_name="reader_xml_conf_resource_limits_profile">
    <topic>
        <resourceLimitsQos>
            <max_samples>200</max_samples>
            <max_instances>20</max_instances>
            <max_samples_per_instance>100</max_samples_per_instance>
            <allocated_samples>50</allocated_samples>
        </resourceLimitsQos>
    </topic>
</data_reader>
```

TimeBasedFilterQosPolicy

Warning: This QoS Policy will be implemented in future releases.

Filter that allows a *DataReader* to specify that it is interested only in a subset of the values of the data. This filter states that the DataReader does not want to receive more than one value each *minimum_separation*, regardless of how fast the changes occur. See *TimeBasedFilterQosPolicy*.

The *minimum_separation* must be lower than the *DeadlineQosPolicy* *period*. By default, the

minimum_separation is zero, which means that the DataReader is potentially interested in all the values.

List of QoS Policy data members:

Data Member Name	Type	Default Value
minimum_separation	*Duration_t*	*c_TimeZero*

Note: This QoS Policy concerns to DataReader entities.

It can be changed on enabled entities.

TopicDataQosPolicy

Allows the application to attach additional information to a created *Topic* so that when it is discovered by a remote application, it can access the data and use it. See *TopicDataQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
collection	std::vector<*octet*>	Empty vector

Note: This QoS Policy concerns to Topic entities.

It can be changed even if it is already created.

Example

C++

```
//The TopicDataQosPolicy is default constructed with an empty vector.
TopicDataQosPolicy topic_data;
std::vector<eprosima::fastrtps::rtps::octet> vec;
vec = topic_data.data_vec(); // Getter Function

//Add two new octets to topic data vector
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
val = 10;
vec.push_back(val);
topic_data.data_vec(vec); //Setter Function
```

XML

```
<data_writer profile_name="writer_xml_conf_topicdata_profile">
    <qos>
        <topicData>
            <value>3.a</value>
        </topicData>
```

(continues on next page)

(continued from previous page)

```
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_topicdata_profile">
    <qos>
        <topicData>
            <value>3.a</value>
        </topicData>
    </qos>
</data_reader>
```

TransportPriorityQosPolicy

Warning: This QoS Policy will be implemented in future releases.

The purpose of this QoS Policy is to allow the service to take advantage of those transports capable of sending messages with different priorities. It establishes the priority of the underlying transport used to send the data. See *TransportPriorityQosPolicy*

You can choose any value within the 32-bit range for the priority. The higher the value, the higher the priority.

List of QoS Policy data members:

Data Member Name	Type	Default Value
value	uint32_t	0

Note: This QoS Policy concerns to *Topic* and *DataWriter* entities.

It can be changed on enabled entities.

UserDataQosPolicy

Allows the application to attach additional information to the *Entity* object so that when the entity is discovered the remote application can access the data and use it. For example, it can be used to attach the security credentials to authenticate the source from the remote application. See *UserDataQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
collection	std::vector<*octet*>	Empty vector

Note: This QoS Policy concerns to all DDS entities.

It can be changed on enabled entities.

Example

C++

```
//The TopicDataQosPolicy is default constructed with an empty vector.
UserDataQosPolicy user_data;
std::vector<eprosima::fastrtps::rtps::octet> vec;
vec = user_data.data_vec(); // Getter Function

//Add two new octets to user data vector
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
val = 10;
vec.push_back(val);
user_data.data_vec(vec); //Setter Function
```

XML

```
<participant profile_name="participant_xml_conf_userdata_profile">
    <rtps>
        <userData>
            <value>3.a</value>
        </userData>
    </rtps>
</participant>

<data_writer profile_name="writer_xml_conf_userdata_profile">
    <qos>
        <userData>
            <value>3.a</value>
        </userData>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_userdata_profile">
    <qos>
        <userData>
            <value>3.a</value>
        </userData>
    </qos>
</data_reader>
```

WriterDataLifecycleQosPolicy

Warning: This QoS Policy will be implemented in future releases.

This QoS Policy specifies the behavior of the DataWriter with respect to the lifecycle of the data instances it manages , that is, the instance that has been either explicitly registered with the DataWriter using the register operations or implicitly by directly writing data.

The *autodispose_unregistered_instances* controls whether a DataWriter will automatically dispose an instance

each time it is unregistered. Even if it is disabled, the application can still get the same result if it uses the dispose operation before unregistering the instance.

List of QoS Policy data members:

Data Member Name	Type	Default Value
autodispose_unregistered_instances	bool	true

Note: This QoS Policy concerns to DataWriter entities.

It can be changed on enabled entities.

eProsima Extensions

The eProsima QoS Policies extensions are those that allow changing the values of the RTPS layer configurable settings.

- *DataSharingQosPolicy*
- *DisablePositiveACKsQosPolicy*
- *FlowControllersQos*
- *ParticipantResourceLimitsQos*
- *PropertyPolicyQos*
- *PublishModeQosPolicy*
- *ReaderResourceLimitsQos*
- *RTPSEndpointQos*
- *RTPSReliableReaderQos*
- *RTPSReliableWriterQos*
- *TransportConfigQos*
- *TypeConsistencyQos*
- *WireProtocolConfigQos*
- *WriterResourceLimitsQos*

DataSharingQosPolicy

This additional QoS allows configuring the data-sharing delivery communication between a writer and a reader. Please, see *Data-sharing delivery* for a description of the data-sharing delivery functionality.

List of QoS Policy data members:

Data Member	Type	Accessor	Default Value
Data-sharing kind	*DataSharingKind*	*kind()*	AUTO
Shared memory directory	string	*shm_directory()*	Empty string
Maximum domain number	uint32_t	*max_domains()*	0 (unlimited)
Data-sharing domain IDs	vector<uint64_t>	*domain_ids()*	Empty

- Data-sharing kind: Specifies the behavior of data-sharing delivery. See *DataSharingKind* for a description of possible values and their effect.
- Shared memory directory: The directory that will be used for the memory-mapped files. If none is configured, then the system default directory will be used.
- Maximum domain number: Establishes the maximum number of data-sharing domain IDs in the local or remote endpoints. Domain IDs are exchanged between data-sharing delivery compatible endpoints. If this value is lower that the size of the list for any remote endpoint, the matching may fail. A value of zero represents unlimited number of IDs.
- Data sharing domain IDs: The list of data-sharing domain IDs configured for the current *DataWriter* or *DataReader*. If no ID is provided, the system will create a unique one for the current machine.

Note: This QoS Policy concerns to *DataWriter* and *DataReader* entities.

It cannot be changed on enabled entities.

DataSharingKind

There are three possible values (see `DataSharingKind`):

- `OFF`: The data-sharing delivery is disabled. No communication will be performed using data-sharing delivery functionality.
- `ON`: The data-sharing delivery is manually enabled. An error will occur if the current topic is not *compatible* with data-sharing delivery. Communication with remote entities that share at least one data-sharing domain ID will be done using data-sharing delivery functionality.
- `AUTO`: data-sharing delivery will be activated if the current topic is *compatible* with data-sharing, and deactivated if not.

Data-sharing configuration helper functions

In order to set the data-sharing delivery configuration, one of the following helper member functions must be used. There is one for each *DataSharingKind* flavor:

Function	Resulting DataSharingKind	Shared memory directory	Data sharing domain IDs
`automatic()`	`AUTO`	Optional	Optional
`on()`	`ON`	Mandatory	Optional
`off()`	`OFF`	N/A	N/A

Instead of defining the data-sharing domain IDs on these helper functions, you can add them later with the `add_domain_id()` function. Beware that adding a new domain ID counts as modifying the QosPolicy, so it must be done before the entity is enabled.

Example

C++

```
DataSharingQosPolicy datasharing;

// Configure the DataSharing as AUTO with two user-defined IDs
std::vector<uint16_t> ids;
ids.push_back(0x1234);
ids.push_back(0xABCD);
datasharing.automatic(ids);

// Alternatively, configure with no IDs and add them afterwards
datasharing.automatic();
datasharing.add_domain_id(uint16_t(0x1234));
datasharing.add_domain_id(uint16_t(0xABCD));

// Or you can leave the IDs empty and the system will create one for you
// unique for the current machine
datasharing.automatic();
```

XML

```
<data_writer profile_name="writer_profile_qos_datasharing">
    <qos>
        <data_sharing>
            <kind>AUTOMATIC</kind>
            <domain_ids>
                <domainId>123</domainId>
                <domainId>098</domainId>
            </domain_ids>
        </data_sharing>
    </qos>
</data_writer>

<data_reader profile_name="reader_profile_qos_datasharing">
    <qos>
        <data_sharing>
            <kind>AUTOMATIC</kind>
            <domain_ids>
                <domainId>123</domainId>
                <domainId>098</domainId>
            </domain_ids>
        </data_sharing>
    </qos>
</data_reader>
```

DisablePositiveACKsQosPolicy

This additional QoS allows reducing network traffic when strict reliable communication is not required and bandwidth is limited. It consists in changing the default behavior by which positive acks are sent from readers to writers. Instead, only negative acks will be sent when a reader is missing a sample, but writers will keep data for an adjustable time before considering it as acknowledged. See *DisablePositiveACKsQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
enabled	bool	false
duration	*Duration_t*	*c_TimeInfinite*

- *enabled*: Specifies if the QoS is enabled or not. If it is true means that the positive acks are disabled and the DataReader only sends negative acks. Otherwise, both positive and negative acks are sent.
- *duration*: State the duration that the DataWriters keep the data before considering it as acknowledged. This value does not apply to DataReaders.

Note: This QoS Policy concerns to *DataWriter* and *DataReader* entities.

The *enabled* Data Member cannot be modified on enabled entities. Thus, this feature must be set up during initialization. Only the *duration* Data Member can be modified at runtime.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See *Compatibility Rule* for further details.

Compatibility Rule

To maintain the compatibility between DisablePositiveACKsQosPolicy in DataReaders and DataWriters, the DataReader cannot have this QoS enabled if the DataWriter have it disabled.

Table with the possible combinations:

DataWriter *enabled* value	DataReader *enabled* value	Compatibility
true	true	Yes
true	false	Yes
false	true	No
false	false	Yes

Example

C++

```
DisablePositiveACKsQosPolicy disable_acks;
//The DisablePositiveACKsQosPolicy is default constructed with enabled = false
//Change enabled to true
disable_acks.enabled = true;
//The DisablePositiveACKsQosPolicy is default constructed with infinite duration
```

(continues on next page)

(continued from previous page)

```
//Change the duration to 1 second
disable_acks.duration = {1, 0};
```

XML

```
<data_writer profile_name="writer_xml_conf_disable_positive_acks_profile">
    <qos>
        <disablePositiveAcks>
            <enabled>true</enabled>
            <duration>
                <sec>1</sec>
            </duration>
        </disablePositiveAcks>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_disable_positive_acks_profile">
    <qos>
        <disablePositiveAcks>
            <enabled>true</enabled>
        </disablePositiveAcks>
    </qos>
</data_reader>
```

FlowControllersQos

This QoS configures the list of flow controllers of a participant, so they can later be used on its DataWriters. It is a vector of shared pointers to *FlowControllerDescriptor*, which has the following fields:

Data Member Name	Type	Default Value
name	const char *	
scheduler	*FlowControllerSchedulerPolicy*	*FIFO*
max_bytes_per_period	int32_t	0 (i.e. infinite)
period_ms	uint64_t	100

Please refer to *Flow Controllers* section for more information.

Note: This QoS Policy concerns to *DomainParticipant* entities.

It cannot be changed on enabled entities.

ParticipantResourceLimitsQos

This QoS configures allocation limits and the use of physical memory for internal resources. See *ParticipantResourceLimitsQos*.

List of QoS Policy data members:

Data Member Name	Type
locators	*RemoteLocatorsAllocationAttributes*
participants	*ResourceLimitedContainerConfig*
readers	*ResourceLimitedContainerConfig*
writers	*ResourceLimitedContainerConfig*
send_buffers	*SendBuffersAllocationAttributes*
data_limits	*VariableLengthDataLimits*
content_filter	*ContentFilterProperty::AllocationConfiguration*

- *locators*: Defines the limits for collections of remote locators.
- *participants*: Specifies the allocation behavior and limits for collections dependent on the total number of participants.
- *readers*: Specifies the allocation behavior and limits for collections dependent on the total number of readers per participant.
- *writers*: Specifies the allocation behavior and limits for collections dependent on the total number of writers per participant.
- *send_buffers*: Defines the allocation behavior and limits for the send buffer manager.
- *data_limits*: States the limits for variable-length data.
- *content_filter*: States the limits for content-filter discovery information.

Note: This QoS Policy concerns to *DomainParticipant* entities.

It cannot be changed on enabled entities.

RemoteLocatorsAllocationAttributes

This structure holds the limits for the remote locators' collections. See *RemoteLocatorsAllocationAttributes*.

List of structure members:

Member Name	Type	Default Value
max_unicast_locators	size_t	4
max_multicast_locators	size_t	1

- *max_unicast_locators*: This member controls the maximum number of unicast locators to keep for each discovered remote entity. It is recommended to use the highest number of local addresses found on all the systems belonging to the same domain.
- *max_multicast_locators*: This member controls the maximum number of multicast locators to keep for each discovered remote entity. The default value is usually enough, as it does not make sense to add more than one multicast locator per entity.

ResourceLimitedContainerConfig

This structure holds the limits of a resource limited collection, as well as the allocation configuration, which can be fixed size or dynamic size.

List of structure members:

Member Name	Type	Default Value
`initial`	`size_t`	0
`maximum`	`size_t`	`std::numeric_limits<size_t>::max()`
`increment`	`size_t`	1 (dynamic size), 0 (fixed size)

- `initial`: Indicates the number of elements to preallocate in the collection.
- `maximum`: Specifies the maximum number of elements allowed in the collection.
- `increment`: States the number of items to add when the reserved capacity limit is reached. This member has a different default value depending on the allocation configuration chosen.

SendBuffersAllocationAttributes

This structure holds the limits for the allocations of the send buffers. See *SendBuffersAllocationAttributes*.

List of structure members:

Member Name	Type	Default Value
preallocated_number	`size_t`	0
dynamic	`bool`	`false`

- *preallocated_number*: This member controls the initial number of send buffers to be allocated. The default value will perform an initial guess of the number of buffers required, based on the number of threads from which a send operation could be started.
- *dynamic*: This member controls how the buffer manager behaves when a send buffer is not available. When true, a new buffer will be created. Otherwise, it will wait for a buffer to be returned.

VariableLengthDataLimits

This structure holds the limits for variable-length data. See *VariableLengthDataLimits*.

List of structure members:

Member Name	Type	Default Value
max_properties	`size_t`	0
max_user_data	`size_t`	0
max_partitions	`size_t`	0

- *max_properties*: Defines the maximum size, in octets, of the properties data in the local or remote participant.
- *max_user_data*: Establishes the maximum size, in octets, of the user data in the local or remote participant.
- *max_partitions*: States the maximum size, in octets, of the partitions data in the local or remote participant.

ContentFilterProperty::AllocationConfiguration

This structure holds the limits for content-filter related discovery information. See *ContentFilterProperty::AllocationConfiguration*.

List of structure members:

Member Name	Type	Default Value
expression_initial_size	size_t	0
expression_parameters	*ResourceLimitedContainerConfig*	{0, 100, 1}

- *expression_initial_size*: Preallocated size of the filter expression.
- *expression_parameters*: Allocation configuration for the list of expression parameters.

Example

C++

```
ParticipantResourceLimitsQos participant_limits;
//Set the maximum size of participant resource limits collection to 3 and it allocation␣
↪configuration to fixed size
participant_limits.participants =␣
↪eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(
    3u);
//Set the maximum size of reader's resource limits collection to 2 and its allocation␣
↪configuration to fixed size
participant_limits.readers = eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_
↪size_configuration(2u);
//Set the maximum size of writer's resource limits collection to 1 and its allocation␣
↪configuration to fixed size
participant_limits.writers = eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_
↪size_configuration(1u);
//Set the maximum size of the partition data to 256
participant_limits.data_limits.max_partitions = 256u;
//Set the maximum size of the user data to 256
participant_limits.data_limits.max_user_data = 256u;
//Set the maximum size of the properties data to 512
participant_limits.data_limits.max_properties = 512u;
//Set the preallocated filter expression size to 512
participant_limits.content_filter.expression_initial_size = 512u;
//Set the maximum number of expression parameters to 4 and its allocation configuration␣
↪to fixed size
participant_limits.content_filter.expression_parameters =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(4u);
```

XML

```
<!--
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
-->
    <participant profile_name="participant_alloc_qos_example">
        <rtps>
            <allocation>
                <!-- We know we have 3 participants on the domain -->
                <total_participants>
                    <initial>3</initial>
                    <maximum>3</maximum>
                    <increment>0</increment>
                </total_participants>
                <!-- We know we have at most 2 readers on each participant -->
                <total_readers>
                    <initial>2</initial>
                    <maximum>2</maximum>
                    <increment>0</increment>
                </total_readers>
                <!-- We know we have at most 1 writer on each participant -->
                <total_writers>
                    <initial>1</initial>
                    <maximum>1</maximum>
                    <increment>0</increment>
                </total_writers>
                <max_partitions>256</max_partitions>
                <max_user_data>256</max_user_data>
                <max_properties>512</max_properties>

                <!-- content_filter cannot be configured using XML (yet) -->
            </allocation>
        </rtps>
    </participant>
```

PropertyPolicyQos

This additional QoS Policy (*PropertyPolicyQos*) stores name/value pairs that can be used to configure certain DDS settings that cannot be configured directly using an standard QoS Policy. For the complete list of settings that can be configured with this QoS Policy, please refer to *PropertyPolicyQos Options*.

This QoS also allows to add custom user properties that could be sent to the external entities. This could be done by setting as `true` the `propagate` value of the Property.

Example

C++

```
PropertyPolicyQos property_policy;
//Add new property for the Auth:PKI-DH plugin
property_policy.properties().emplace_back("dds.sec.auth.plugin", "builtin.PKI-DH");
//Add new property for the Access:Permissions plugin
property_policy.properties().emplace_back(eprosima::fastrtps::rtps::Property("dds.sec.
↪access.plugin",
        "builtin.Access-Permissions"));

//Add new user custom property to send to external Participants
property_policy.properties().emplace_back("Custom Property Name", "Custom value", true);
```

XML

```
<participant profile_name="secure_participant_conf_all_plugin_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.plugin</name>
                    <value>builtin.PKI-DH</value>
                </property>

                <!-- Activate Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.plugin</name>
                    <value>builtin.Access-Permissions</value>
                </property>

                <!-- User Custom Property to send externally -->
                <property>
                    <name>Custom Property Name</name>
                    <value>Custom value</value>
                    <propagate>true</propagate>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

PublishModeQosPolicy

This QoS Policy configures how the *DataWriter* sends the data. See *PublishModeQosPolicy*.

It also configures the name of the flow controller to use when asynchronous publishing is used. It should be the name of a flow controller registered on the creation of the DomainParticipant. See *FlowControllersQos*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*PublishModeQosPolicyKind*	*SYNCHRONOUS_PUBLISH_MODE*
flow_controller_name	const char *	*FASTDDS_FLOW_CONTROLLER_DEFAULT*

Note: This QoS Policy concerns to DataWriter entities.

It cannot be changed on enabled entities.

PublishModeQosPolicyKind

There are two possible values (see *PublishModeQosPolicyKind*):

- *SYNCHRONOUS_PUBLISH_MODE*: The data is sent in the context of the user thread that calls the write operation.
- *ASYNCHRONOUS_PUBLISH_MODE*: An internal thread takes the responsibility of sending the data asynchronously. The write operation returns before the data is actually sent.

Example

C++

```
PublishModeQosPolicy publish_mode;
//The PublishModeQosPolicy is default constructed with kind = SYNCHRONOUS
//Change the kind to ASYNCHRONOUS
publish_mode.kind = ASYNCHRONOUS_PUBLISH_MODE;
```

XML

```
<data_writer profile_name="writer_profile_qos_publishmode">
    <qos>
        <publishMode>
            <kind>ASYNCHRONOUS</kind>
        </publishMode>
    </qos>
</data_writer>
```

ReaderResourceLimitsQos

This QoS Policy states the limits for the matched *DataWriters*' resource limited collections based on the maximum number of DataWriters that are going to match with the *DataReader*. See `ReaderResourceLimitsQos`.

List of QoS Policy data members:

Data Member Name	Type
`matched_publisher_allocation`	*ResourceLimitedContainerConfig*

Note: This QoS Policy concerns to DataReader entities.

It cannot be changed on enabled entities.

Example

C++

```
ReaderResourceLimitsQos reader_limits;
//Set the maximum size for writer matched resource limits collection to 1 and its␣
↪allocation configuration to fixed size
reader_limits.matched_publisher_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(1u);
```

XML

```
<data_reader profile_name="alloc_qos_example_sub">
    <!-- we know we will only have one matching publisher -->
    <matchedPublishersAllocation>
        <initial>1</initial>
        <maximum>1</maximum>
        <increment>0</increment>
    </matchedPublishersAllocation>
</data_reader>
```

RTPSEndpointQos

This QoS Policy configures the aspects of an RTPS endpoint, such as the list of locators, the identifiers, and the history memory policy. See `RTPSEndpointQos`.

List of QoS Policy data members:

Data Member Name	Type	Default Value
unicast_locator_list	*LocatorList*	Empty List
multicast_locator_list	*LocatorList*	Empty List
remote_locator_list	*LocatorList*	Empty List
external_unicast_locators	*ExternalLocators*	Empty
ignore_non_matching_locators	bool	false
user_defined_id	int16_t	-1
entity_id	int16_t	-1
history_memory_policy	*MemoryManagementPolicy*	*PREALLOCATED_WITH_REALLOC_MEMORY_MODE*

- *unicast_locator_list*: Defines the list of unicast locators associated to the DDS Entity. DataReaders and DataWriters inherit the list of unicast locators set in the DomainParticipant, but it can be changed by means of this QoS.
- *multicast_locator_list*: Stores the list of multicast locators associated to the DDS Entity. By default, DataReaders and DataWriters do not use any multicast locator, but it can be changed by means of this QoS.
- *remote_locator_list*: States the list of remote locators associated to the DDS Entity.
- *external_unicast_locators*: Defines the *External Locators* to announce for the communication with this DDS Entity.
- *ignore_non_matching_locators*: Defines whether to ignore locators received on announcements from other DDS entities when they don't match with any of the locators announced by this DDS Entity.
- *user_defined_id*: Establishes the unique identifier used for StaticEndpointDiscovery.
- *entity_id*: The user can specify the identifier for the endpoint.
- *history_memory_policy*: Indicates the way the memory is managed in terms of dealing with the CacheChanges.

Note: This QoS Policy concerns to *DataWriter* and *DataReader* entities.

It cannot be changed on enabled entities.

MemoryManagementPolicy

There are four possible values (see *MemoryManagementPolicy*):

- *PREALLOCATED_MEMORY_MODE*: This option sets the size to the maximum of each data type. It produces the largest memory footprint but the smallest allocation count.
- *PREALLOCATED_WITH_REALLOC_MEMORY_MODE*: This option set the size to the default for each data type and it requires reallocation when a bigger message arrives. It produces a lower memory footprint at the expense of increasing the allocation count.
- *DYNAMIC_RESERVE_MEMORY_MODE*: This option allocates the size dynamically at the time of message arrival. It produces the least memory footprint but the highest allocation count.
- *DYNAMIC_REUSABLE_MEMORY_MODE*: This option is similar to DYNAMIC_RESERVE_MEMORY_MODE, but the allocated memory is reused for future messages.

Example

C++

```
RTPSEndpointQos endpoint;
//Add new unicast locator with port 7800
eprosima::fastrtps::rtps::Locator_t new_unicast_locator;
new_unicast_locator.port = 7800;
endpoint.unicast_locator_list.push_back(new_unicast_locator);
//Add new multicast locator with IP 239.255.0.4 and port 7900
eprosima::fastrtps::rtps::Locator_t new_multicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(new_multicast_locator, "239.255.0.4");
new_multicast_locator.port = 7900;
endpoint.multicast_locator_list.push_back(new_multicast_locator);
// Add an external locator with IP 100.100.100.10, port 12345, mask 24, externality 1,␣
↪and cost 0
eprosima::fastdds::rtps::LocatorWithMask external_locator;
external_locator.kind = LOCATOR_KIND_UDPv4;
external_locator.port = 12345;
external_locator.mask(24);
endpoint.external_unicast_locators[1][0].push_back(external_locator);
// Drop non matching locators
endpoint.ignore_non_matching_locators = true;
//Set 3 as user defined id
endpoint.user_defined_id = 3;
//Set 4 as entity id
endpoint.entity_id = 4;
//The RTPSEndpointQos is default constructed with history_memory_policy = PREALLOCATED
//Change the history_memory_policy to DYNAMIC_RESERVE
endpoint.history_memory_policy = eprosima::fastrtps::rtps::DYNAMIC_RESERVE_MEMORY_MODE;
```

XML

```
<data_writer profile_name="writer_xml_conf_unicast_locators_profile">
    <userDefinedID>3</userDefinedID>
    <entityID>2</entityID> <!-- Int16 -->
    <unicastLocatorList>
        <locator>
            <udpv4>
                <port>7800</port>
            </udpv4>
        </locator>
    </unicastLocatorList>
    <multicastLocatorList>
        <locator>
            <udpv4>
                <address>239.255.0.4</address>
                <port>7900</port>
            </udpv4>
        </locator>
```

(continues on next page)

(continued from previous page)

```
    </multicastLocatorList>
    <external_unicast_locators>
        <udpv4 externality="1" cost="0" mask="24">
            <address>100.100.100.10</address>
            <port>12345</port>
        </udpv4>
    </external_unicast_locators>
    <ignore_non_matching_locators>true</ignore_non_matching_locators>
    <!-- The history memory policy is changed to DYNAMIC_RESERVE -->
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</data_writer>

<data_reader profile_name="reader_xml_conf_unicast_locators_profile">
    <userDefinedID>5</userDefinedID>
    <entityID>4</entityID> <!-- Int16 -->
    <unicastLocatorList>
        <locator>
            <udpv4>
                <port>7800</port>
            </udpv4>
        </locator>
    </unicastLocatorList>
    <multicastLocatorList>
        <locator>
            <udpv4>
                <address>239.255.0.4</address>
                <port>7900</port>
            </udpv4>
        </locator>
    </multicastLocatorList>
    <external_unicast_locators>
        <udpv4 externality="1" cost="0" mask="24">
            <address>100.100.100.10</address>
            <port>12345</port>
        </udpv4>
    </external_unicast_locators>
    <ignore_non_matching_locators>true</ignore_non_matching_locators>
    <historyMemoryPolicy>PREALLOCATED_WITH_REALLOC</historyMemoryPolicy>
</data_reader>
```

RTPSReliableReaderQos

This RTPS QoS Policy allows the configuration of several RTPS reliable reader's aspects. See *RTPSReliableReaderQos*.

List of QoS Policy data members:

Data Member Name	Type
times	*ReaderTimes*
disable_positive_ACKs	*DisablePositiveACKsQosPolicy*

- *times*: Defines the duration of the RTPSReader events. See *ReaderTimes* for further details.

- *disable_positive_ACKs*: Configures the settings to disable the positive acks. See *DisablePositiveACKsQosPolicy* for further details.

Note: This QoS Policy concerns to *DataReader* entities.

Only the *duration* Data Member of the *DisablePositiveACKsQosPolicy* and the *times* Data Member can be modified on enabled entities.

ReaderTimes

This structure defines the times associated with the Reliable Readers' events. See *ReaderTimes*.

List of structure members:

Member Name	Type	Default Value
initialAcknackDelay	*Duration_t*	70 ms
heartbeatResponseDelay	*Duration_t*	5 ms

- *initialAcknackDelay*: Defines the duration of the initial acknack delay.
- *heartbeatResponseDelay*: Establishes the duration of the delay applied when a heartbeat message is received.

Example

C++

```
RTPSReliableReaderQos reliable_reader_qos;
//The RTPSReliableReaderQos is default constructed with initialAcknackDelay = 70 ms
//Change the initialAcknackDelay to 70 nanoseconds
reliable_reader_qos.times.initialAcknackDelay = {0, 70};
//The RTPSReliableWriterQos is default constructed with heartbeatResponseDelay = 5 ms
//Change the heartbeatResponseDelay to 5 nanoseconds
reliable_reader_qos.times.heartbeatResponseDelay = {0, 5};
//You can also change the DisablePositiveACKsQosPolicy. For further details see␣
↪DisablePositiveACKsQosPolicy section.
reliable_reader_qos.disable_positive_ACKs.enabled = true;
```

XML

```
<data_reader profile_name="sub_profile_name">
    <times> <!-- readerTimesType -->
        <initialAcknackDelay> <!-- DURATION -->
            <nanosec>70</nanosec>
        </initialAcknackDelay>
        <heartbeatResponseDelay> <!-- DURATION -->
            <nanosec>5</nanosec>
        </heartbeatResponseDelay>
    </times>
    <!--You can also change the values of DisablePositiveACKsQosPolicy.-->
```

(continues on next page)

(continued from previous page)

```
    <!--See DisablePositiveACKsQosPolicy section for further details-->
</data_reader>
```

RTPSReliableWriterQos

This RTPS QoS Policy allows the configuration of several RTPS reliable writer's aspects. See *RTPSReliableWriterQos*.

List of QoS Policy data members:

Data Member Name	Type
times	*WriterTimes*
disable_positive_acks	*DisablePositiveACKsQosPolicy*
disable_heartbeat_piggyback	*DisableHeartbeatPiggyback*

- *times*: Defines the duration of the RTPSWriter events. See *WriterTimes* for further details.
- *disable_positive_acks*: Configures the settings to disable the positive acks. See *DisablePositiveACKsQosPolicy* for further details.
- *disable_heartbeat_piggyback*: Configures the settings to disable the heartbeat piggyback mechanism. See *DisableHeartbeatPiggyback* for further details.

Note: This QoS Policy concerns to *DataWriter* entities.

Only the *duration* Data Member of the *DisablePositiveACKsQosPolicy* and the *times* Data Member can be modified on enabled entities.

WriterTimes

This structure defines the times associated with the Reliable Writers' events.

List of structure members:

Member Name	Type	Default Value
initialHeartbeatDelay	*Duration_t*	12ms
heartbeatPeriod	*Duration_t*	3s
nackResponseDelay	*Duration_t*	5ms
nackSupressionDuration	*Duration_t*	0s

- *initialHeartbeatDelay*: Defines duration of the initial heartbeat delay.
- *heartbeatPeriod*: Specifies the interval between periodic heartbeats.
- *nackResponseDelay*: Establishes the duration of the delay applied to the response of an ACKNACK message.
- *nackSupressionDuration*: The RTPSWriter ignores the nack messages received after sending the data until the duration time elapses.

DisableHeartbeatPiggyback

Besides sending heartbeats periodically using the *heartbeatPeriod* (see *WriterTimes*), reliable DataWriters also use a mechanism to append a heartbeat submessage in the same message where data is being delivered to the DataReaders. This mechanism acts in specific situations where the reliable communication state must be up to date to maintain optimal communication:

- When the DataWriter sends as many bytes to the *socket* as the length of the *socket* buffer, a heartbeat submessage is appended after the last data.
- When the DataWriter's history is full, the DataWriter starts to append heartbeat submessages after each data.

This mechanism can be disabled using this policy.

Example

C++

```
RTPSReliableWriterQos reliable_writer_qos;
//The RTPSReliableWriterQos is default constructed with initialHeartbeatDelay = 12 ms
//Change the initialHeartbeatDelay to 20 nanoseconds
reliable_writer_qos.times.initialHeartbeatDelay = {0, 20};
//The RTPSReliableWriterQos is default constructed with heartbeatPeriod = 3 s
//Change the heartbeatPeriod to 5 seconds
reliable_writer_qos.times.heartbeatPeriod = {5, 0};
//The RTPSReliableWriterQos is default constructed with nackResponseDelay = 5 ms
//Change the nackResponseDelay to 10 nanoseconds
reliable_writer_qos.times.nackResponseDelay = {0, 10};
//The RTPSReliableWriterQos is default constructed with nackSupressionDuration = 0 s
//Change the nackSupressionDuration to 20 nanoseconds
reliable_writer_qos.times.nackSupressionDuration = {0, 20};
//You can also change the DisablePositiveACKsQosPolicy. For further details see␣
↪DisablePositiveACKsQosPolicy section.
reliable_writer_qos.disable_positive_acks.enabled = true;
//The RTPSReliableWriterQos is default constructed with disable_heartbeat_piggyback =␣
↪false
//Disable the heartbeat piggyback mechanism.
reliable_writer_qos.disable_heartbeat_piggyback = true;
```

XML

```
<data_writer profile_name="pub_profile_name">
    <times> <!-- writerTimesType -->
        <initialHeartbeatDelay> <!-- DURATION -->
            <nanosec>20</nanosec>
        </initialHeartbeatDelay>
        <heartbeatPeriod> <!-- DURATION -->
            <sec>5</sec>
        </heartbeatPeriod>
        <nackResponseDelay> <!-- DURATION -->
            <nanosec>10</nanosec>
```

(continues on next page)

(continued from previous page)

```
        </nackResponseDelay>
        <nackSupressionDuration> <!-- DURATION -->
            <nanosec>20</nanosec>
        </nackSupressionDuration>
    </times>

    <!--You can also change the values of DisablePositiveACKsQosPolicy.-->
    <!--See DisablePositiveACKsQosPolicy section for further details-->
    <qos>
        <!--Disable heartbeat piggyback mechanism.-->
        <disable_heartbeat_piggyback>true</disable_heartbeat_piggyback>
    </qos>
</data_writer>
```

TransportConfigQos

This QoS Policy allows the configuration of the transport layer settings. See *TransportConfigQos*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
user_transports	std::vector<std::shared_ptr<TransportDescriptorInterface>>	Empty vector
use_builtin_transports	bool	true
send_socket_buffer_size	uint32_t	0
listen_socket_buffer_size	uint32_t	0

- *user_transports*: This data member defines the list of transports to use alongside or in place of builtins.
- *use_builtin_transports*: It controls whether the built-in transport layer is enabled or disabled. If it is set to false, the default UDPv4 implementation is disabled.
- *send_socket_buffer_size*: By default, Fast DDS creates socket buffers using the system default size. This data member allows to change the send socket buffer size used to send data.
- *listen_socket_buffer_size*: The listen socket buffer size is also created with the system default size, but it can be changed using this data member.

Note: This QoS Policy concerns to *DomainParticipant* entities.

It cannot be changed on enabled entities.

TransportDescriptorInterface

This structure is the base for the data type used to define transport configuration.

List of structure members:

Member Name	Type
maxMessageSize	uint32_t
maxInitialPeersRange	uint32_t

- *maxMessageSize*: This member sets the maximum size in bytes of the transport's message buffer.
- *maxInitialPeersRange*: This member states the maximum number of guessed initial peers to try to connect.

Example

C++

```
TransportConfigQos transport;
//Add new transport to the list of user transports
std::shared_ptr<eprosima::fastdds::rtps::UDPv4TransportDescriptor> descriptor =
        std::make_shared<eprosima::fastdds::rtps::UDPv4TransportDescriptor>();
descriptor->sendBufferSize = 9126;
descriptor->receiveBufferSize = 9126;
transport.user_transports.push_back(descriptor);
//Set use_builtin_transports to false
transport.use_builtin_transports = false;
```

XML

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>my_transport</transport_id>
        <type>UDPv4</type>
        <sendBufferSize>9216</sendBufferSize>
        <receiveBufferSize>9216</receiveBufferSize>
    </transport_descriptor>
</transport_descriptors>

<participant profile_name="my_transport">
    <rtps>
        <userTransports>
            <transport_id>my_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
    </rtps>
</participant>
```

Note: *TransportConfigQos* can also be configured modifying the builtin transports configuration by selecting one of

the available builtin transports options. See *Managing the Builtin Transports* or `setup_transports()`.

TypeConsistencyQos

This QoS Policy allows the configuration of the *XTypes extension QoS* on the *DataReader*. See `TypeConsistencyQos`.

List of QoS Policy data members:

Data Member Name	Type
`type_consistency`	*TypeConsistencyEnforcementQosPolicy*
`representation`	*DataRepresentationQosPolicy*

- `type_consistency`: It states the rules for the data types compatibility. See *TypeConsistencyEnforcementQosPolicy* for further details.
- `representation`: It specifies the data representations valid for the entities. See *DataRepresentationQosPolicy* for further details.

Note: This QoS Policy concerns to DataReader entities.

It cannot be changed on enabled entities.

Example

C++

```
TypeConsistencyQos consistency_qos;
//You can change the DataRepresentationQosPolicy. For further details see␣
↪DataRepresentationQosPolicySection section.
consistency_qos.representation.m_value.push_back(DataRepresentationId_t::XCDR2_DATA_
↪REPRESENTATION);
//You can change the TypeConsistencyEnforcementQosPolicy. For further details see␣
↪TypeConsistencyEnforcementQosPolicy section.
consistency_qos.type_consistency.m_kind = TypeConsistencyKind::ALLOW_TYPE_COERCION;
```

XML

This QoS Policy cannot be configured using XML for the moment.

WireProtocolConfigQos

This QoS Policy allows the configuration of the wire protocol. See *WireProtocolConfigQos*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
prefix	*GuidPrefix_t*	0
participant_id	int32_t	-1
builtin	*BuiltinAttributes*	
port	*PortParameters*	
default_unicast_locator_list	*LocatorList*	Empty List
default_multicast_locator_list	*LocatorList*	Empty List
default_external_unicast_locators	*ExternalLocators*	Empty
ignore_non_matching_locators	bool	false

- *prefix*: This data member allows the user to set manually the GUID prefix.
- *participant_id*: It sets the participant identifier. By default, it will be automatically generated by the Domain.
- *builtin*: This data member allows the configuration of the built-in parameters.
- *port*: This data member allows the configuration of the port parameters and gains related to the RTPS protocol (*Well Known Ports*).
- *default_unicast_locator_list*: States the default list of unicast locators to be used for any endpoint defined inside the RTPSParticipant in the case that it was defined without unicast locators. This list should include at least one locator.
- *default_multicast_locator_list*: Stores the default list of multicast locators to be used for any endpoint defined inside the RTPSParticipant in the case that it was defined without multicast locators. This list is usually left empty.
- *default_external_unicast_locators*: Defines the *External Locators* to be used for any endpoint defined inside the participant in the case that it was defined without unicast locators.
- *ignore_non_matching_locators*: Defines whether to ignore locators received on announcements from other DDS participants when they don't match with any of the locators announced by this DDS participant.

Note: This QoS Policy concerns to DomainParticipant entities.

Important: The only mutable field on enabled entities is *m_DiscoveryServers*, which is contained in *discovery_config* within *builtin* (see *Modifying remote servers list at run time*).

Example

C++

```
WireProtocolConfigQos wire_protocol;
//Set the guid prefix
std::istringstream("72.61.73.70.66.61.72.6d.74.65.73.74") >> wire_protocol.prefix;
//Configure Builtin Attributes
wire_protocol.builtin.discovery_config.discoveryProtocol =
        eprosima::fastrtps::rtps::DiscoveryProtocol_t::SERVER;
//Add locator to unicast list
eprosima::fastrtps::rtps::Locator_t server_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(server_locator, "192.168.10.57");
server_locator.port = 56542;
wire_protocol.builtin.metatrafficUnicastLocatorList.push_back(server_locator);
// Add a metatraffic external locator with IP 100.100.100.10, port 34567, mask 24,␣
↪externality 1, and cost 0
eprosima::fastdds::rtps::LocatorWithMask meta_external_locator;
meta_external_locator.kind = LOCATOR_KIND_UDPv4;
meta_external_locator.port = 34567;
meta_external_locator.mask(24);
wire_protocol.builtin.metatraffic_external_unicast_locators[1][0].push_back(meta_
↪external_locator);
//Add locator to default unicast locator list
eprosima::fastrtps::rtps::Locator_t unicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(unicast_locator, 192, 168, 1, 41);
unicast_locator.port = 7400;
wire_protocol.default_unicast_locator_list.push_back(unicast_locator);
//Add locator to default multicast locator list
eprosima::fastrtps::rtps::Locator_t multicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(multicast_locator, 192, 168, 1, 41);
multicast_locator.port = 7400;
wire_protocol.default_multicast_locator_list.push_back(multicast_locator);
// Add a default external locator with IP 100.100.100.10, port 23456, mask 24,␣
↪externality 1, and cost 0
eprosima::fastdds::rtps::LocatorWithMask external_locator;
external_locator.kind = LOCATOR_KIND_UDPv4;
external_locator.port = 23456;
external_locator.mask(24);
wire_protocol.default_external_unicast_locators[1][0].push_back(external_locator);
// Drop non matching locators
wire_protocol.ignore_non_matching_locators = true;
```

XML

```
<participant profile_name="UDP SERVER WP" is_default_profile="true">
    <rtps>
        <prefix>72.61.73.70.66.61.72.6d.74.65.73.74</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.10.57</address>
                        <port>56542</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
            <metatraffic_external_unicast_locators>
                <udpv4 externality="1" cost="0" mask="24">
                    <address>100.100.100.10</address>
                    <port>34567</port>
                </udpv4>
            </metatraffic_external_unicast_locators>
        </builtin>
        <defaultUnicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
        </defaultUnicastLocatorList>

        <defaultMulticastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
        </defaultMulticastLocatorList>

        <default_external_unicast_locators>
            <udpv4 externality="1" cost="0" mask="24">
                <address>100.100.100.10</address>
                <port>23456</port>
            </udpv4>
        </default_external_unicast_locators>

        <ignore_non_matching_locators>true</ignore_non_matching_locators>
```

(continues on next page)

(continued from previous page)

```
    </rtps>
</participant>
```

WriterResourceLimitsQos

This QoS Policy states the limits for the matched *DataReaders*' resource limited collections based on the maximum number of DataReaders that are going to match with the *DataWriter*. See `WriterResourceLimitsQos`.

List of QoS Policy data members:

Data Member Name	Type
matched_subscriber_allocation	*ResourceLimitedContainerConfig*
reader_filters_allocation	*ResourceLimitedContainerConfig*

Note: This QoS Policy concerns to DataWriter entities.

It cannot be changed on enabled entities.

Example

C++

```
WriterResourceLimitsQos writer_limits;
//Set the maximum size for reader matched resource limits collection to 3 and its␣
↪allocation configuration to fixed size
writer_limits.matched_subscriber_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(3u);
// Set the maximum number of writer side content filters to 1 and its allocation␣
↪configuration to fixed size
writer_limits.reader_filters_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(1u);
```

XML

```
<data_writer profile_name="alloc_qos_example_pub_for_topic_1">
    <!-- we know we will have three matching subscribers -->
    <matchedSubscribersAllocation>
        <initial>3</initial>
        <maximum>3</maximum>
        <increment>0</increment>
    </matchedSubscribersAllocation>

    <!-- reader_filters_allocation cannot be configured using XML (yet) -->
</data_writer>
```

XTypes Extensions

This section explain those QoS Policy extensions defined in the XTypes Specification:

- *DataRepresentationQosPolicy*
- *TypeConsistencyEnforcementQosPolicy*

DataRepresentationQosPolicy

This XTypes QoS Policy states which data representations will be used by the *DataWriters* and *DataReaders*.

The DataWriters offer a single data representation that will be used to communicate with the matched DataReaders. The DataReaders can request one or more data representations and in order to have communication with the DataWriter, the offered data representation needs to be contained within the DataReader request. See *DataRepresentationQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
m_value	std::vector<*DataRepresentationId*>	Empty vector

Note: This QoS Policy concerns to Topic, DataReader and DataWriter entities.

It cannot be changed on enabled entities.

DataRepresentationId

There are three possible values (see *DataRepresentationId*):

- *XCDR_DATA_REPRESENTATION*: This option corresponds to the first version of the *Extended CDR Representation* encoding.
- *XML_DATA_REPRESENTATION*: This option corresponds to the *XML Data Representation*.
- *XCDR2_DATA_REPRESENTATION*: This option corresponds to the second version of the *Extended CDR Representation* encoding.

Example

C++

```
DataRepresentationQosPolicy data_representation;
//Add XCDR v1 data representation to the list of valid representations
data_representation.m_value.push_back(DataRepresentationId_t::XCDR_DATA_REPRESENTATION);
//Add XML data representation to the list of valid representations
data_representation.m_value.push_back(DataRepresentationId_t::XML_DATA_REPRESENTATION);
```

XML

This QoS Policy cannot be configured using XML for the moment.

TypeConsistencyEnforcementQosPolicy

This XTypes QoS Policy extension defines the rules for determining whether the data type used in the *DataWriter* is consistent with the one used in the *DataReader*. See `TypeConsistencyEnforcementQosPolicy`.

List of QoS Policy data members:

Data Member Name	Type	Default Value
`m_kind`	*TypeConsistencyKind*	`ALLOW_TYPE_COERCION`
`m_ignore_sequence_bounds`	`bool`	`true`
`m_ignore_string_bounds`	`bool`	`true`
`m_ignore_member_names`	`bool`	`true`
`m_prevent_type_widening`	`bool`	`true`
`m_force_type_validation`	`bool`	`true`

- `m_kind`: It determines whether the type in the DataWriter type must be equal to the type in the DataReader or not. See *TypeConsistencyKind* for further details.
- `m_ignore_sequence_bounds`: This data member controls whether the sequence bounds are taken into account for type assignability or not. If its value is true, the sequences maximum lengths are not considered, which means that a sequence T2 with length L2 would be assignable to a sequence T1 with length L1, even if L2 is greater than L1. But if it is false, L1 must be higher or equal to L2 to consider the sequences as assignable.
- `m_ignore_string_bounds`: It controls whether the string bounds are considered for type assignation or not. If its value is true, the strings maximum lengths are not considered, which means that a string S2 with length L2 would be assignable to a string S1 with length L1, even if L2 is greater than L1. But if it is false, L1 must be higher or equal to L2 to consider the strings as assignable.
- `m_ignore_member_names`: This boolean controls whether the member names are taken into consideration for type assignability or not. If it is true, apart from the member ID, the member names are considered as part of assignability, which means that the members with the same ID must also have the same name. But if the value is false, the member names are ignored.
- `m_prevent_type_widening`: This data member controls whether the type widening is allowed or not. If it is false, the type widening is permitted, but if true, a wider type cannot be assignable to a narrower type.
- `m_force_type_validation`: It controls if the service needs the type information to complete the matching between a DataWriter and a DataReader. If it is enabled, it must have the Complete Type Information, otherwise it is not necessary.

Note: This QoS Policy concerns to DataReader entities.

It cannot be changed on enabled entities.

TypeConsistencyKind

There are two possible values:

- *DISALLOW_TYPE_COERCION*: The DataWriter and the DataReader must support the same data type in order to communicate.
- *ALLOW_TYPE_COERCION*: The DataWriter and the DataReader do not need to support the same data type in order to communicate as long as the DataReader's type is assignable from the DataWriter's type.

Example

C++

```
TypeConsistencyEnforcementQosPolicy type_enforcement;
//The TypeConsistencyEnforcementQosPolicy is default constructed with kind = ALLOW_TYPE_
↪COERCION
//Change the kind to DISALLOW_TYPE_COERCION
type_enforcement.m_kind = TypeConsistencyKind::DISALLOW_TYPE_COERCION;
//Configures the system to ignore the sequence sizes in assignations
type_enforcement.m_ignore_sequence_bounds = true;
//Configures the system to ignore the string sizes in assignations
type_enforcement.m_ignore_string_bounds = true;
//Configures the system to ignore the member names. Members with same ID could have␣
↪different names
type_enforcement.m_ignore_member_names = true;
//Configures the system to allow type widening
type_enforcement.m_prevent_type_widening = false;
//Configures the system to not use the complete Type Information in entities match␣
↪process
type_enforcement.m_force_type_validation = false;
```

XML

This QoS Policy cannot be configured using XML for the moment.

Status

Each *Entity* is associated with a set of `Status` objects whose values represent the *communication status* of that Entity. Changes on the status values occur due to communication events related to each of the entities, e.g., when new data arrives, a new participant is discovered, or a remote endpoint is lost. The status is decomposed into several status objects, each concerning a different aspect of the communication, so that each of these status objects can vary independently of the others.

Changes on a status object trigger the corresponding *Listener* callbacks that allow the Entity to inform the application about the event. For a given status object with name `fooStatus`, the entity listener interface defines a callback function `on_foo()` that will be called when the status changes. Beware that some statuses have data members that are reset every time the corresponding listener is called. The only exception to this rule is when the entity has no listener attached, so the callback cannot be called. See the documentation of each status for details.

Conditions and Wait-sets provide the application with an alternative mechanism to make it aware of changes on status objects, by means of a *StatusCondition*. The advantage of this mechanism is that the application can wait for changes

on several entities at the same time. It will also help the determinism of your system, as the notification is not processed on an internal thread, as it is done when using listeners.

The entities expose functions to access the value of its statuses. For a given status with name `fooStatus`, the entity exposes a member function `get_foo()` to access the data in its `fooStatus`. The only exceptions are *DataOnReaders* and *DataAvailable*. These getter functions return a read-only struct where all data members are public and accessible to the application. Beware that some statuses have data members that are reset every time the getter function is called by the application. See the documentation of each status for details.

The following subsections describe each of the status objects, their data members, and to which Entity type they concern. The next table offers a quick reference as well as the corresponding bit for each status in the *StatusMask*.

Status Name	Entity	Listener callback	Accessor	Bit
InconsistentTopicStatus	*Topic*	*on_inconsistent_topic()*	*get_inconsistent_topic_status()*	0
OfferedDeadlineMissedStatus	*DataWriter*	*on_offered_deadline_missed()*	*get_offered_deadline_missed_status()*	1
RequestedDeadlineMissedStatus	*DataReader*	*on_requested_deadline_missed()*	*get_requested_deadline_missed_status()*	2
OfferedIncompatibleQosStatus	*DataWriter*	*on_offered_incompatible_qos()*	*get_offered_incompatible_qos_status()*	5
RequestedIncompatibleQosStatus	*DataReader*	*on_requested_incompatible_qos()*	*get_requested_incompatible_qos_status()*	6
SampleLostStatus	*DataReader*	*on_sample_lost()*	*get_sample_lost_status()*	7
SampleRejectedStatus	*DataReader*	*on_sample_rejected()*	*get_sample_rejected_status()*	8
DataOnReaders	*Subscriber*	*on_data_on_readers()*	N/A	9
DataAvailable	*DataReader*	*on_data_available()*	N/A	10
LivelinessLostStatus	*DataWriter*	*on_liveliness_lost()*	*get_liveliness_lost_status()*	11
LivelinessChangedStatus	*DataReader*	*on_liveliness_changed()*	*get_liveliness_changed_status()*	12
PublicationMatchedStatus	*DataWriter*	*on_publication_matched()*	*get_publication_matched_status()*	13
SubscriptionMatchedStatus	*DataReader*	*on_subscription_matched()*	*get_subscription_matched_status()*	14

InconsistentTopicStatus

This status changes every time an inconsistent remote Topic is discovered, that is, one with the same name but different characteristics than the current Topic. See *InconsistentTopicStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t

- *total_count*: Total cumulative count of inconsistent Topics discovered since the creation of the current Topic.
- *total_count_change*: The change in *total_count* since the last time *on_inconsistent_topic()* was called or the status was read.

Warning: Currently this status is not supported and will be implemented in future releases. As a result, trying to access this status will return `NOT_SUPPORTED` and the corresponding listener will never be called.

DataOnReaders

This status becomes active every time there is new data available for the application on any DataReader belonging to the current Subscriber. There is no getter function to access this status, as it does not keep track of any information related to the data itself. Its only purpose is to trigger the *on_data_on_readers()* callback on the listener attached to the DataReader.

DataAvailable

This status becomes active every time there is new data available for the application on the DataReader. There is no getter function to access this status, as it does not keep track of any information related to the data itself. Its only purpose is to trigger the *on_data_available()* callback on the listener attached to the DataReader.

LivelinessChangedStatus

This status changes every time the liveliness status of a matched DataWriter has changed. Either because a DataWriter that was *inactive* has become *active* or the other way around. See *LivelinessChangedStatus*.

List of status data members:

Data Member Name	Type
alive_count	int32_t
not_alive_count	int32_t
alive_count_change	int32_t
not_alive_count_change	int32_t
last_publication_handle	*InstanceHandle_t*

- *alive_count*: Total number of currently active DataWriters. This count increases every time a newly matched DataWriter asserts its liveliness or a DataWriter that was considered not alive reasserts its liveliness. It decreases every time an active DataWriter becomes not alive, either because it failed to asserts its liveliness or because it was deleted for any reason.
- *not_alive_count*: Total number of matched DataWriters that are currently considered not alive. This count increases every time an active DataWriter becomes not alive because it fails to assert its liveliness. It decreases every time a DataWriter that was considered not alive reasserts its liveliness. Normal matching and unmatching of DataWriters does not affect this count.
- *alive_count_change*: The change in *alive_count* since the last time *on_liveliness_changed()* was called or the status was read. It can have positive or negative values.
- *not_alive_count_change*: The change in *not_alive_count* since the last time *on_liveliness_changed()* was called or the status was read. It can have positive or negative values.
- *last_publication_handle*: Handle to the last DataWriter whose liveliness status was changed. If no liveliness has ever changed, it will have value `c_InstanceHandle_Unknown`.

RequestedDeadlineMissedStatus

This status changes every time the DataReader does not receive data within the deadline period configured on its *DataReaderQos*. See *RequestedDeadlineMissedStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
last_instance_handle	*InstanceHandle_t*

- *total_count*: Total cumulative count of missed deadlines for any instance read by the current DataReader. As the deadline period applies to each instance of the Topic independently, the count will will be incremented by one for each instance for which data was not received in the deadline period.
- *total_count_change*: The change in *total_count* since the last time *on_requested_deadline_missed()* was called or the status was read. It can only have zero or positive values.
- *last_instance_handle*: Handle to the last instance that missed the deadline. If no deadline was ever missed, it will have value `c_InstanceHandle_Unknown`.

RequestedIncompatibleQosStatus

This status changes every time the DataReader finds a DataWriter that matches the Topic and has a common partition, but with a QoS configuration incompatible with the one defined on the DataReader. See *RequestedIncompatibleQosStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
last_policy_id	*QosPolicyId_t*
policies	*QosPolicyCountSeq*

- *total_count*: Total cumulative count of DataWriters found matching the Topic and with a common partition, but with a QoS configuration that is incompatible with the one defined on the DataReader.
- *total_count_change*: The change in *total_count* since the last time *on_requested_incompatible_qos()* was called or the status was read. It can only have zero or positive values.
- *last_policy_id*: The policy ID of one of the policies that was found to be incompatible with the current DataReader. If more than one policy happens to be incompatible, only one of them will be reported in this member.
- *policies*: A collection that holds, for each policy, the total number of times that the policy was found to be incompatible with the one offered by a remote DataWriter that matched the Topic and with a common partition. See *QosPolicyCountSeq* and *QosPolicyCount* for more information the information that is stored for each policy.

QosPolicyCountSeq

Holds a *QosPolicyCount* for each *Policy*, indexed by its *QosPolicyId_t*. Therefore, the Qos Policy with ID N will be at position N in the sequence. See *QosPolicyCountSeq*.

```
DataReader* data_reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);

// Get how many times ReliabilityQosPolicy was not compatible with a remote writer
RequestedIncompatibleQosStatus status;
data_reader->get_requested_incompatible_qos_status(status);
uint32_t incompatible_reliability_count = status.policies[RELIABILITY_QOS_POLICY_ID].
↪count;
```

QosPolicyCount

This structure holds a counter for a policy. See *QosPolicyCount*.

List of data members:

Data Member Name	Type
policy_id	*QosPolicyId_t*
count	int32_t

- *policy_id*: The ID of the policy.
- *count*: The counter value for the policy.

SampleLostStatus

This status changes every time a new data sample is lost and will never be received. See *SampleLostStatus*.

There are two different criteria for considering a sample as lost depending on the *reliability()*:

- When using *BEST_EFFORT_RELIABILITY_QOS*, a not yet received sample is considered lost whenever a sample with a greater sequence number is received.
- When using *RELIABLE_RELIABILITY_QOS*, a not yet received sample is considered lost whenever the DataWriter informs, through an RTPS HEARTBEAT submessage, that the sample is not available anymore.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t

- *total_count*: Total cumulative count of lost samples under the Topic of the current DataReader.
- *total_count_change*: The change in *total_count* since the last time *on_sample_lost()* was called or the status was read. It can only be positive or zero.

SampleRejectedStatus

This status changes every time an incoming data sample is rejected by the DataReader. The reason for the rejection is defined by *SampleRejectedStatusKind*. For further information see *SampleRejectedStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
last_reason	*SampleRejectedStatusKind*
last_instance_handle	*InstanceHandle_t*

- *total_count*: Total cumulative count of rejected samples under the Topic of the current DataReader.
- *total_count_change*: The change in *total_count* since the last time *on_sample_rejected()* was called or the status was read. It can only be positive or zero.
- *last_reason*: The reason for rejecting the last rejected sample. If no sample was ever rejected, it will have value *NOT_REJECTED*. See *SampleRejectedStatusKind* for further details.
- *last_instance_handle*: Handle to the last instance whose sample was rejected. If no sample was ever rejected, it will have value c_InstanceHandle_Unknown.

SampleRejectedStatusKind

In Fast DDS, samples can be rejected due to resource limit reasons. However, the fact that the samples are rejected does not imply that they are lost, i.e. a rejected sample may be accepted in the future.

SampleRejectedStatusKind specifies the reason of the rejection:

- *NOT_REJECTED* specifies that the samples were not rejected.
- *REJECTED_BY_SAMPLES_LIMIT* specifies that the samples were rejected because there were not enough resources to stored them. This can happen even when there are free resources if those resources must be guaranteed to be available for other samples. This situation, which arises in the RTPS layer, occurs when there are yet to be received samples with lower sequence number and there is not enough resources for all of them (because *max_samples* has been reached).
- *REJECTED_BY_INSTANCES_LIMIT* specifies that the samples were rejected because there were not enough resources to allocate the samples' instances. This situation, which arises in the DDS layer, more precisely in the in the DataReader's history, occurs when the sample corresponds to a new instance for which the middleware should reserve resources but the history's number of instances has already reached *max_instances*.
- *REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT* specifies that the samples were rejected because there were not enough resources within their instance to stored them. This situation, which arises in the DDS layer, more precisely in the DataReader's history, occurs when the DataReader is configured with *KEEP_ALL_HISTORY_QOS* and the instance's number of samples has reached *max_samples_per_instance*.

SubscriptionMatchedStatus

This status changes every time the DataReader finds a DataWriter that matches the Topic and has a common partition and a compatible QoS, or has ceased to be matched with a DataWriter that was previously considered to be matched. See *SubscriptionMatchedStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
current_count	int32_t
current_count_change	int32_t
last_publication_handle	*InstanceHandle_t*

- *total_count*: Total cumulative count of remote DataWriters that have been discovered publishing on the same Topic and has a common partition and a compatible QoS. They may not all be matched at the moment.
- *total_count_change*: The change in *total_count* since the last time *on_subscription_matched()* was called or the status was read. It can only have zero or positive values.
- *current_count*: The number of remote DataWriters currently matched to the DataReader.
- *current_count_change*: The change in *current_count* since the last time *on_subscription_matched()* was called or the status was read. It can have positive or negative values.
- *last_publication_handle*: Handle to the last DataWriter that matched the DataReader. If no matching ever happened, it will have value *c_InstanceHandle_Unknown*.

LivelinessLostStatus

This status changes every time the DataWriter failed to assert its liveliness during the period configured on its *DataWriterQos*. This means that matched DataReader entities will consider the DataWriter as no longer *alive*. See *LivelinessLostStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t

- *total_count*: Total cumulative count of times that the DataWriter failed to assert its liveliness during the period configured on its *DataWriterQos*, becoming considered not *alive*. This count does not change when the DataWriter is already considered not *alive* and simply remains not *alive* for another liveliness period.
- *total_count_change*: The change in *total_count* since the last time *on_liveliness_lost()* was called or the status was read. It can only have zero or positive values.

OfferedDeadlineMissedStatus

This status changes every time the DataWriter fails to provide data within the deadline period configured on its *DataWriterQos*. See *OfferedDeadlineMissedStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
last_instance_handle	*InstanceHandle_t*

- *total_count*: Total cumulative count of missed deadlines for any instance written by the current DataWriter. As the deadline period applies to each instance of the Topic independently, the count will will be incremented by one for each instance for which data was not sent in the deadline period.
- *total_count_change*: The change in *total_count* since the last time *on_offered_deadline_missed()* was called or the status was read. It can only have zero or positive values.
- *last_instance_handle*: Handle to the last instance that missed the deadline. If no deadline was ever missed, it will have value *c_InstanceHandle_Unknown*.

OfferedIncompatibleQosStatus

This status changes every time the DataWriter finds a DataReader that matches the Topic and has a common partition, but with a QoS configuration that is incompatible with the one defined on the DataWriter. See *OfferedIncompatibleQosStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
last_policy_id	*QosPolicyId_t*
policies	*QosPolicyCountSeq*

- *total_count*: Total cumulative count of DataReaders found matching the Topic and with a common partition, but with a QoS configuration that is incompatible with the one defined on the DataWriter.
- *total_count_change*: The change in *total_count* since the last time *on_offered_incompatible_qos()* was called or the status was read. It can only have zero or positive values.
- *last_policy_id*: The policy ID of one of the policies that was found to be incompatible with the current DataWriter. If more than one policy happens to be incompatible, only one of them will be reported in this member.
- *policies*: A collection that holds, for each policy, the total number of times that the policy was found to be incompatible with the one requested by a remote DataReader that matched the Topic and with a common partition. See *QosPolicyCountSeq* and *QosPolicyCount* for more information the information that is stored for each policy.

PublicationMatchedStatus

This status changes every time the DataWriter finds a DataReader that matches the Topic and has a common partition and a compatible QoS, or has ceased to be matched with a DataReader that was previously considered to be matched. See *PublicationMatchedStatus*.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
current_count	int32_t
current_count_change	int32_t
last_subscription_handle	*InstanceHandle_t*

- *total_count*: Total cumulative count of remote DataReaders that have been discovered publishing on the same Topic and has a common partition and a compatible QoS. They may not all be matched at the moment.
- *total_count_change*: The change in *total_count* since the last time *on_publication_matched()* was called or the status was read. It can only have zero or positive values.
- *current_count*: The number of remote DataReaders currently matched to the DataWriter.
- *current_count_change*: The change in *current_count* since the last time *on_publication_matched()* was called or the status was read. It can have positive or negative values.
- *last_subscription_handle*: Handle to the last DataReader that matched the DataWriter. If no matching ever happened, it will have value *c_InstanceHandle_Unknown*.

Conditions and Wait-sets

Conditions (in conjunction with wait-sets) provide an alternative mechanism to allow the middleware to notify communication status changes (including arrival of data) to the application.

This mechanism is wait-based. Its general use pattern is as follows:

- The application indicates which relevant information it wants to get, by means of *Condition* objects (*GuardCondition*, *StatusCondition*, or *ReadCondition*) and attaching them to a *Wait-set* via the *attach_condition()* call.
- It then waits on that *Wait-set* via the *wait()* call until the trigger value of one or several *Condition* objects become true.
- It then uses the result of the *wait()* (i.e., the list of *Condition* objects with trigger_value == true) to actually get the information by calling:
  - *get_status_changes()*, then checking if any of the changes is relevant using the *StatusMask::is_active()* method on the result and finally calling get_<communication_status> on the relevant Entity, when the condition is a StatusCondition and the status changes refer to plain communication status. Refer to *Status* for additional information on the different statuses that can be queried.
  - *get_status_changes()* and then *Subscriber::get_datareaders()* on the relevant Subscriber, when the condition is a StatusCondition and the status changes refer to DataOnReaders.
  - *get_status_changes()* and then *DataReader::read()*/*DataReader::take()* on the relevant DataReader, when the condition is a StatusCondition and the status changes refer to DataAvailable.

  - Directly *DataReader::read_w_condition()*/*DataReader::take_w_condition()* on the DataReader with the *Condition* as a parameter, when it is a ReadCondition

- When a Condition is no longer relevant it can be detached from a *Wait-set* via the *detach_condition()* call.

The first step is usually done in an initialization phase, while the others are put in the application main loop.

```
class ApplicationJob
{
    WaitSet wait_set_;
    GuardCondition terminate_condition_;
    std::thread thread_;

    void main_loop()
    {
        // Main loop is repeated until the terminate condition is triggered
        while (false == terminate_condition_.get_trigger_value())
        {
            // Wait for any of the conditions to be triggered
            ReturnCode_t ret_code;
            ConditionSeq triggered_conditions;
            ret_code = wait_set_.wait(triggered_conditions, eprosima::fastrtps::c_
↪TimeInfinite);
            if (ReturnCode_t::RETCODE_OK != ret_code)
            {
                // ... handle error
                continue;
            }

            // Process triggered conditions
            for (Condition* cond : triggered_conditions)
            {
                StatusCondition* status_cond = dynamic_cast<StatusCondition*>(cond);
                if (nullptr != status_cond)
                {
                    Entity* entity = status_cond->get_entity();
                    StatusMask changed_statuses = entity->get_status_changes();

                    // Process status. Liveliness changed and data available are␣
↪depicted as an example
                    if (changed_statuses.is_active(StatusMask::liveliness_changed()))
                    {
                        std::cout << "Liveliness changed reported for entity " << entity-
↪>get_instance_handle() <<
                            std::endl;
                    }

                    if (changed_statuses.is_active(StatusMask::data_available()))
                    {
                        std::cout << "Data avilable on reader " << entity->get_instance_
↪handle() << std::endl;

                        FooSeq data_seq;
                        SampleInfoSeq info_seq;
```

(continues on next page)

(continued from previous page)

```
                        DataReader* reader = static_cast<DataReader*>(entity);

                        // Process all the samples until no one is returned
                        while (ReturnCode_t::RETCODE_OK == reader->take(data_seq, info_
↪seq,
                                LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
                                ANY_VIEW_STATE, ANY_INSTANCE_STATE))
                        {
                            // Both info_seq.length() and data_seq.length() will have␣
↪the number of samples returned
                            for (FooSeq::size_type n = 0; n < info_seq.length(); ++n)
                            {
                                // Only samples with valid data should be accessed
                                if (info_seq[n].valid_data && reader->is_sample_valid(&
↪data_seq[n], &info_seq[n]))
                                {
                                    // Process sample on data_seq[n]
                                }
                            }

                            // must return the loaned sequences when done processing
                            reader->return_loan(data_seq, info_seq);
                        }
                    }
                }
            }
        }
    }

public:

    ApplicationJob(
            const std::vector<DataReader*>& readers,
            const std::vector<DataWriter*>& writers)
    {
        // Add a GuardCondition, so we can signal the processing thread to stop
        wait_set_.attach_condition(terminate_condition_);

        // Add the status condition of every reader and writer
        for (DataReader* reader : readers)
        {
            wait_set_.attach_condition(reader->get_statuscondition());
        }
        for (DataWriter* writer : writers)
        {
            wait_set_.attach_condition(writer->get_statuscondition());
        }

        thread_ = std::thread(&ApplicationJob::main_loop, this);
    }

    ~ApplicationJob()
```

(continues on next page)

(continued from previous page)

```
    {
        // Signal the GuardCondition to force the WaitSet to wake up
        terminate_condition_.set_trigger_value(true);
        // Wait for the thread to finish
        thread_.join();
    }

};

// Application initialization
ReturnCode_t ret_code;
std::vector<DataReader*> application_readers;
std::vector<DataWriter*> application_writers;

// Create the participant, topics, readers, and writers.
ret_code = create_dds_application(application_readers, application_writers);
if (ReturnCode_t::RETCODE_OK != ret_code)
{
    // ... handle error
    return;
}

{
    ApplicationJob main_loop_thread(application_readers, application_writers);

    // ... wait for application termination signaling (signal handler, user input, etc)

    // ... Destructor of ApplicationJob takes care of stopping the processing thread
}

// Destroy readers, writers, topics, and participant
destroy_dds_application();
```

Calling the *wait()* operation on the *Wait-set* will block the calling thread if the trigger value of all the conditions attached to it are false. The thread will wake up, and the *wait()* operation will return RETCODE_OK, whenever the trigger value of any of the attached conditions becomes true.

GuardCondition

A condition for which the trigger value is completely controlled by the application via its *set_trigger_value()* operation.

StatusCondition

A condition that triggers whenever there are changes on the communication statuses of an Entity.

The sensitivity of the *StatusCondition* to a particular communication status is controlled by the list of enabled_statuses set on the condition by means of the `set_enabled_statuses()` operation.

ReadCondition

A condition that triggers whenever the DataReader that created it contains at least a sample with SampleState, ViewState, and InstanceState matching those of the ReadCondition.

The fact that the trigger value of a ReadCondition is dependent on the presence of samples on the associated DataReader implies that a single take operation can potentially change the trigger value of several ReadCondition conditions. For example, if all samples are taken, any ReadCondition associated with the DataReader that were triggered before, will see their trigger value changed to false. Note that this does not guarantee that WaitSet objects that were separately attached to those conditions will not be woken up. Once we have trigger_value == true on a condition, it may wake up the attached *Wait-set*. The condition transitioning to trigger_value == false does not necessarily 'unwakeup' the *Wait-set*, as 'unwakening' may not be possible in general. The consequence is that an application blocked on a *Wait-set* may return from the wait with a list of conditions, some of which are no longer triggered. This also may be the consequence of user actions. A user manually calling `set_trigger_value()` could potentially trigger the same behavior. This is unavoidable if multiple threads are concurrently waiting on separate *Wait-set* objects and taking data associated with the same DataReader entity.

To elaborate further, consider the following example: A ReadCondition that has a sample_state_mask = {NOT_READ} will have trigger_value == true whenever a new sample arrives and will transition to false as soon as all the newly-arrived samples are either read (so their status changes to READ) or taken (so they are no longer managed by the DataReader). However, if the same ReadCondition had a sample_state_mask = {READ, NOT_READ}, then the trigger_value would only become false once all the newly-arrived samples are taken (it is not sufficient to read them as that would only change the SampleState to READ which overlaps the mask on the ReadCondition).

6.17.2 Domain

A domain represents a separate communication plane. It creates a logical separation among the Entities that share a common communication infrastructure. Conceptually, it can be seen as a *virtual network* linking all applications running on the same domain and isolating them from applications running on different domains. This way, several independent distributed applications can coexist in the same physical network without interfering, or even being aware of each other.

Every domain has a unique identifier, called domainId, that is implemented as a `uint32` value. Applications that share this domainId belong to the same domain and will be able to communicate.

For an application to be added to a domain, it must create an instance of *DomainParticipant* with the appropriate domainId. Instances of DomainParticipant are created through the *DomainParticipantFactory* singleton.

Partitions introduce another entity isolation level within the domain. While DomainParticipant will be able to communicate with each other if they are in the same domain, it is still possible to isolate their *Publishers* and *Subscribers* assigning them to different *Partitions*.

Fig. 6: Domain class diagram

DomainParticipant

A *DomainParticipant* is the entry point of the application to a domain. Every DomainParticipant is linked to a single domain from its creation, and contains all the Entities related to that domain. It also acts as a factory for *Publisher*, *Subscriber* and *Topic*.

The behavior of the DomainParticipant can be modified with the QoS values specified on DomainParticipantQos. The QoS values can be set at the creation of the DomainParticipant, or modified later with `DomainParticipant::set_qos()` member function.

As an Entity, DomainParticipant accepts a *DomainParticipantListener* that will be notified of status changes on the DomainParticipant instance.

DomainParticipantQos

`DomainParticipantQos` controls the behavior of the DomainParticipant. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
UserDataQosPolicy	`user_data()`	Yes
EntityFactoryQosPolicy	`entity_factory()`	Yes
ParticipantResourceLimitsQos	`allocation()`	No
PropertyPolicyQos	`properties()`	No
WireProtocolConfigQos	`wire_protocol()`	No*
TransportConfigQos	`transport()` and `setup_transports()`	No
FlowControllersQos	`flow_controllers()`	No

Important: The only mutable field in *WireProtocolConfigQos* is `m_DiscoveryServers`, which is contained in `discovery_config` within `builtin` (see *Modifying remote servers list at run time*).

Important: Upon the call to `create_participant()`, if Fast DDS is compiled with statistics support (enabled by default, see *CMake options*), the internal `DomainParticipantQos` may differ from the input `DomainParticipantQos` (see *Statistics Module Settings*). This entails that applications willing to further modify the `DomainParticipantQos` after `DomainParticipant` creation should:

1. Retrieve the internal `DomainParticipantQos` by the means of `DomainParticipant::get_qos()`.
2. Perform the desired modifications.
3. Update the `DomainParticipantQos` by the means of `DomainParticipant::set_qos()`.

Refer to the detailed description of each QosPolicy class for more information about their usage and default values.

The QoS value of a previously created DomainParticipant can be modified using the `DomainParticipant::set_qos()` member function. Trying to modify an immutable QosPolicy on an already enabled DomainParticipant will result on an error. In such case, no changes will be applied and the DomainParticipant will keep its previous DomainParticipantQos.

```
// Create a DomainParticipant with default DomainParticipantQos
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
```

(continues on next page)

(continued from previous page)

```
if (nullptr == participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
DomainParticipantQos qos = participant->get_qos();

// Modify QoS attributes
qos.entity_factory().autoenable_created_entities = false;

// Assign the new Qos to the object
participant->set_qos(qos);
```

Default DomainParticipantQos

The default DomainParticipantQos refers to the value returned by the *get_default_participant_qos()* member function on the *DomainParticipantFactory* singleton. The special value `PARTICIPANT_QOS_DEFAULT` can be used as QoS argument on *create_participant()* or *DomainParticipant::set_qos()* member functions to indicate that the current default DomainParticipantQos should be used.

When the system starts, the default DomainParticipantQos is equivalent to the default constructed value *DomainParticipantQos()*. The default DomainParticipantQos can be modified at any time using the *set_default_participant_qos()* member function on the DomainParticipantFactory singleton. Modifying the default DomainParticipantQos will not affect already existing DomainParticipant instances.

```
// Get the current QoS or create a new one from scratch
DomainParticipantQos qos_type1 = DomainParticipantFactory::get_instance()->get_default_
↪participant_qos();

// Modify QoS attributes
// (...)

// Set as the new default TopicQos
if (DomainParticipantFactory::get_instance()->set_default_participant_qos(qos_type1) !=
        ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DomainParticipant with the new default DomainParticipantQos.
DomainParticipant* participant_with_qos_type1 =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant_with_qos_type1)
{
    // Error
    return;
}
```

(continues on next page)

(continued from previous page)

```
// Get the current QoS or create a new one from scratch
DomainParticipantQos qos_type2;

// Modify QoS attributes
// (...)

// Set as the new default TopicQos
if (DomainParticipantFactory::get_instance()->set_default_participant_qos(qos_type2) !=
        ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Topic with the new default TopicQos.
DomainParticipant* participant_with_qos_type2 =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DomainParticipantQos to the original default constructed values
if (DomainParticipantFactory::get_instance()->set_default_participant_qos(PARTICIPANT_
↪QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (DomainParticipantFactory::get_instance()->set_default_participant_
↪qos(DomainParticipantQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

set_default_participant_qos() member function also accepts the value `PARTICIPANT_QOS_DEFAULT` as input argument. This will reset the current default DomainParticipantQos to the default constructed value *DomainParticipantQos()*.

```
// Create a custom DomainParticipantQos
DomainParticipantQos custom_qos;

// Modify QoS attributes
// (...)
```

(continues on next page)

(continued from previous page)

```
// Create a DomainParticipant with a custom DomainParticipantQos

DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
↪participant(0, custom_qos);
if (nullptr == participant)
{
    // Error
    return;
}

// Set the QoS on the participant to the default
if (participant->set_qos(PARTICIPANT_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (participant->set_qos(DomainParticipantFactory::get_instance()->get_default_
↪participant_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `PARTICIPANT_QOS_DEFAULT` has different meaning depending on where it is used:

- On *create_participant()* and *DomainParticipant::set_qos()* it refers to the default DomainParticipantQos as returned by *get_default_participant_qos()*.
- On *set_default_participant_qos()* it refers to the default constructed *DomainParticipantQos()*.

DomainParticipantListener

DomainParticipantListener is an abstract class defining the callbacks that will be triggered in response to state changes on the *DomainParticipant*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

DomainParticipantListener inherits from *TopicListener*, *PublisherListener*, and *SubscriberListener*. Therefore, it has the ability to react to every kind of event that is reported to any of its attached Entities. Since events are always notified to the most specific Entity Listener that can handle the event, callbacks that DomainParticipantListener inherits from other Listeners will only be called if no other Entity was able to handle the event, either because it has no Listener attached, or because the callback is disabled by the *StatusMask* on the Entity.

Additionally, DomainParticipantListener adds the following non-standard callbacks:

- *on_participant_discovery()*: A new DomainParticipant is discovered in the same domain, a previously known DomainParticipant has been removed, or some DomainParticipant has changed its QoS. This method provides an overload with an additional boolean output parameter so a discovery callback can tell the middle-

ware if a newly discovered participant has to be ignored via the use of the *ignore_participant()*. This overload should be used when there is a need to ignore participants inside the discovery callback, since calling *ignore_participant()* inside the listener might deadlock. If both callbacks are implemented, the discovery callback with the `should_be_ignored` boolean flag takes precedence. The second discovery callback is only executed if the discovered DomainParticipant is not ignored in the first callback (`should_be_ignored` parameter returns `false`).

- *on_subscriber_discovery()*: A new *Subscriber* is discovered in the same domain, a previously known Subscriber has been removed, or some Subscriber has changed its QoS.
- *on_publisher_discovery()*: A new *Publisher* is discovered in the same domain, a previously known Publisher has been removed, or some Publisher has changed its QoS.
- *on_type_discovery()*: A new data Type is discovered in the same domain.
- *on_type_dependencies_reply()*: The Type lookup client received a replay to a `getTypeDependencies()` request. This callback can be used to retrieve the new type using the `getTypes()` request and create a new dynamic type using the retrieved type object.
- *on_type_information_received()*: A new *TypeInformation* has been received from a newly discovered DomainParticipant.
- *onParticipantAuthentication()*: Informs about the result of the authentication process of a remote DomainParticipant (either on failure or success).

Important: For more information about callbacks and its hierarchy, please refer to *Listener*.

```
class CustomDomainParticipantListener : public DomainParticipantListener
{

public:

    CustomDomainParticipantListener()
        : DomainParticipantListener()
    {
    }

    virtual ~CustomDomainParticipantListener()
    {
    }

    void on_participant_discovery(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ParticipantDiscoveryInfo&& info,
            bool& should_be_ignored) override
    {
        should_be_ignored = false;
        if (info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOVERED_PARTICIPANT)
        {
            std::cout << "New participant discovered" << std::endl;
            // The following line can be modified to evaluate whether the discovered␣
↪participant should be ignored
            // (usually based on fields present in the discovery information)
            bool ignoring_condition = false;
```

(continues on next page)

(continued from previous page)

```
            if (ignoring_condition)
            {
                should_be_ignored = true; // Request the ignoring of the discovered␣
↪participant
            }
        }
        else if (info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::REMOVED_PARTICIPANT ||
                info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DROPPED_PARTICIPANT)
        {
            std::cout << "New participant lost" << std::endl;
        }
    }

#if HAVE_SECURITY
    void onParticipantAuthentication(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ParticipantAuthenticationInfo&& info) override
    {
        if (info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::AUTHORIZED_PARTICIPANT)
        {
            std::cout << "A participant was authorized" << std::endl;
        }
        else if (info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::UNAUTHORIZED_PARTICIPANT)
        {
            std::cout << "A participant failed authorization" << std::endl;
        }
    }

#endif // if HAVE_SECURITY

    void on_subscriber_discovery(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info) override
    {
        if (info.status == eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERED_
↪READER)
        {
            std::cout << "New subscriber discovered" << std::endl;
        }
        else if (info.status == eprosima::fastrtps::rtps::ReaderDiscoveryInfo::REMOVED_
↪READER)
        {
            std::cout << "New subscriber lost" << std::endl;
        }
    }

    void on_publisher_discovery(
            DomainParticipant* /*participant*/,
```

(continues on next page)

(continued from previous page)

```
            eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info) override
    {
        if (info.status == eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERED_
↪WRITER)
        {
            std::cout << "New publisher discovered" << std::endl;
        }
        else if (info.status == eprosima::fastrtps::rtps::WriterDiscoveryInfo::REMOVED_
↪WRITER)
        {
            std::cout << "New publisher lost" << std::endl;
        }
    }

    void on_type_discovery(
            DomainParticipant* participant,
            const eprosima::fastrtps::rtps::SampleIdentity& request_sample_id,
            const eprosima::fastrtps::string_255& topic,
            const eprosima::fastrtps::types::TypeIdentifier* identifier,
            const eprosima::fastrtps::types::TypeObject* object,
            eprosima::fastrtps::types::DynamicType_ptr dyn_type) override
    {
        static_cast<void>(participant);
        static_cast<void>(request_sample_id);
        static_cast<void>(topic);
        static_cast<void>(identifier);
        static_cast<void>(object);
        static_cast<void>(dyn_type);
        std::cout << "New data type discovered" << std::endl;

    }

    void on_type_dependencies_reply(
            DomainParticipant* participant,
            const eprosima::fastrtps::rtps::SampleIdentity& request_sample_id,
            const eprosima::fastrtps::types::TypeIdentifierWithSizeSeq& dependencies)␣
↪override
    {
        static_cast<void>(participant);
        static_cast<void>(request_sample_id);
        static_cast<void>(dependencies);
        std::cout << "Answer to a request for type dependencies was received" <<␣
↪std::endl;
    }

    void on_type_information_received(
            DomainParticipant* participant,
            const eprosima::fastrtps::string_255 topic_name,
            const eprosima::fastrtps::string_255 type_name,
            const eprosima::fastrtps::types::TypeInformation& type_information) override
    {
        static_cast<void>(participant);
```

(continues on next page)

(continued from previous page)

```
        static_cast<void>(topic_name);
        static_cast<void>(type_name);
        static_cast<void>(type_information);
        std::cout << "New data type information received" << std::endl;
    }

};
```

DomainParticipantFactory

The sole purpose of this class is to allow the creation and destruction of *DomainParticipant* objects. *DomainParticipantFactory* itself has no factory, it is a singleton object that can be accessed through the *get_instance()* static member function on the *DomainParticipantFactory* class.

The behavior of the DomainParticipantFactory can be modified with the QoS values specified on *DomainParticipantFactoryQos*. Since the DomainParticipantFactory is a singleton, its QoS can only be modified with the *DomainParticipantFactory::set_qos()* member function.

DomainParticipantFactory does not accept any Listener, since it is not an Entity.

DomainParticipantFactoryQos

DomainParticipantFactoryQos controls the behavior of the *DomainParticipantFactory*. Internally it contains the following *QosPolicy* objects:

QosPolicy class	Accessor/Mutator	Mutable
EntityFactoryQosPolicy	*entity_factory()*	Yes

Since the DomainParticipantFactory is a singleton, its QoS can only be modified with the *DomainParticipantFactory::set_qos()* member function.

```
DomainParticipantFactoryQos qos;

// Setting autoenable_created_entities to true makes the created DomainParticipants
// to be enabled upon creation
qos.entity_factory().autoenable_created_entities = true;
if (DomainParticipantFactory::get_instance()->set_qos(qos) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DomainParticipant with the new DomainParticipantFactoryQos.
// The returned DomainParticipant is already enabled
DomainParticipant* enabled_participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == enabled_participant)
{
    // Error
```

(continues on next page)

(continued from previous page)

```
    return;
}

// Setting autoenable_created_entities to false makes the created DomainParticipants
// to be disabled upon creation
qos.entity_factory().autoenable_created_entities = false;
if (DomainParticipantFactory::get_instance()->set_qos(qos) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DomainParticipant with the new DomainParticipantFactoryQos.
// The returned DomainParticipant is disabled and will need to be enabled explicitly
DomainParticipant* disabled_participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == disabled_participant)
{
    // Error
    return;
}
```

Loading profiles from an XML file

To create Entities based on XML profiles, the file containing such profiles must be loaded first.

If the profile is described in one of the default loaded files, it will be automatically available on initialization. Otherwise, *load_XML_profiles_file()* member function can be used to load the profiles in the XML. See section *XML profiles* for more information regarding XML profile format and automatic loading.

Once loaded, the name of the profiles can be used to create Entities that will have QoS settings according to the profile specifications.

```
// Load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Profiles can now be used to create Entities
DomainParticipant* participant_with_profile =
        DomainParticipantFactory::get_instance()->create_participant_with_profile(0,
↪"participant_profile");
if (nullptr == participant_with_profile)
{
    // Error
    return;
}
```

Creating a DomainParticipant

Creation of a *DomainParticipant* is done with the *create_participant()* member function on the *DomainParticipantFactory* singleton, that acts as a factory for the DomainParticipant.

Mandatory arguments are:

- The `DomainId` that identifies the domain where the DomainParticipant will be created.
- The *DomainParticipantQos* describing the behavior of the DomainParticipant. If the provided value is `TOPIC_QOS_DEFAULT`, the value of the DomainParticipantQos is used.

Optional arguments are:

- A Listener derived from *DomainParticipantListener*, implementing the callbacks that will be triggered in response to events and state changes on the DomainParticipant. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the *DomainParticipantListener*. By default all events are enabled.

Warning: Following the DDSI-RTPS V2.2 standard (Section 9.6.1.1), the default ports are calculated depending on the `DomainId`, as it is explained in section *Well Known Ports*. Thus, it is encouraged to use `DomainId` lower than 200 (over `DomainId` 233 default port assign will fail consistently).

create_participant() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DomainParticipant with default DomainParticipantQos and no Listener
// The value PARTICIPANT_QOS_DEFAULT is used to denote the default QoS.
DomainParticipant* participant_with_default_attributes =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant_with_default_attributes)
{
    // Error
    return;
}

// A custom DomainParticipantQos can be provided to the creation method
DomainParticipantQos custom_qos;

// Modify QoS attributes
// (...)

DomainParticipant* participant_with_custom_qos =
        DomainParticipantFactory::get_instance()->create_participant(0, custom_qos);
if (nullptr == participant_with_custom_qos)
{
    // Error
    return;
}

// Create a DomainParticipant with default QoS and a custom Listener.
// CustomDomainParticipantListener inherits from DomainParticipantListener.
// The value PARTICIPANT_QOS_DEFAULT is used to denote the default QoS.
```

(continues on next page)

(continued from previous page)

```
CustomDomainParticipantListener custom_listener;
DomainParticipant* participant_with_default_qos_and_custom_listener =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT,
                &custom_listener);
if (nullptr == participant_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a DomainParticipant

Instead of using a DomainParticipantQos, the name of a profile can be used to create a DomainParticipant with the *create_participant_with_profile()* member function on the *DomainParticipantFactory* singleton.

Mandatory arguments are:

- The `DomainId` that identifies the domain where the DomainParticipant will be created. Do not use `DomainId` higher than 200 (see *Creating a DomainParticipant*).
- The name of the profile to be applied to the DomainParticipant.

Optional arguments are:

- A Listener derived from *DomainParticipantListener*, implementing the callbacks that will be triggered in response to events and state changes on the DomainParticipant. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the *DomainParticipantListener*. By default all events are enabled.

create_participant_with_profile() will return a null pointer if there was an error during the operation, e.g if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant using a profile and no Listener
DomainParticipant* participant_with_profile =
        DomainParticipantFactory::get_instance()->create_participant_with_profile(0,
↪"participant_profile");
if (nullptr == participant_with_profile)
{
    // Error
    return;
}

// Create a DomainParticipant using a profile and a custom Listener.
// CustomDomainParticipantListener inherits from DomainParticipantListener.
CustomDomainParticipantListener custom_listener;
DomainParticipant* participant_with_profile_and_custom_listener =
```

(continues on next page)

(continued from previous page)

```
        DomainParticipantFactory::get_instance()->create_participant_with_profile(0,
↪"participant_profile",
                &custom_listener);
if (nullptr == participant_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a DomainParticipant

A DomainParticipant can be deleted with the *delete_participant()* member function on the *DomainParticipantFactory* singleton.

Note: A DomainParticipant can only be deleted if all Entities belonging to the participant (Publisher, Subscriber or Topic) have already been deleted. Otherwise, the function will issue an error and the DomainParticipant will not be deleted. This can be performed by using the *delete_contained_entities()* member function of the *DomainParticipant*.

```
// Create a DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Use the DomainParticipant to communicate
// (...)

// Delete entities created by the DomainParticipant
if (participant->delete_contained_entities() != ReturnCode_t::RETCODE_OK)
{
    // DomainParticipant failed to delete the entities it created.
    return;
}

// Delete the DomainParticipant
if (DomainParticipantFactory::get_instance()->delete_participant(participant) !=␣
↪ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Partitions

Partitions introduce a logical entity isolation level concept inside the physical isolation induced by a *Domain*. They represent another level to separate *Publishers* and *Subscribers* beyond Domain and *Topic*. For a Publisher to communicate with a Subscriber, they have to belong at least to one common partition. In this sense, partitions represent a light mechanism to provide data separation among endpoints:

- Unlike Domain and Topic, Partitions can be changed dynamically during the life cycle of the endpoint with little cost. Specifically, no new threads are launched, no new memory is allocated, and the change history is not affected. Beware that modifying the Partition membership of endpoints will trigger the announcement of the new QoS configuration, and as a result, new endpoint matching may occur, depending on the new Partition configuration. Changes on the memory allocation and running threads may occur due to the matching of remote endpoints.
- Unlike Domain and Topic, an endpoint can belong to several Partitions at the same time. For certain data to be shared over different Topics, there must be a different Publisher for each Topic, each of them sharing its own history of changes. On the other hand, a single Publisher can share the same data over different Partitions using a single topic data change, thus reducing network overload.

The Partition membership of an endpoint can be configured on the *PartitionQosPolicy* data member of the *PublisherQos* or *SubscriberQos* objects. This member holds a list of Partition name strings. If no Partition is defined for an entity, it will be automatically included in the default nameless Partition. Therefore, a Publisher and a Subscriber that specify no Partition will still be able to communicate through the default nameless Partition.

Warning: Partitions are linked to the endpoint and not to the changes. This means that the endpoint history is oblivious to modifications in the Partitions. For example, if a Publisher switches Partitions and afterwards needs to resend some older change again, it will deliver it to the new Partition set, regardless of which Partitions were defined when the change was created. This means that a late joiner Subscriber may receive changes that were created when another set of Partitions was active.

Wildcards in Partitions

Partition name entries can have wildcards following the naming conventions defined by the POSIX `fnmatch` API (1003.2-1992 section B.6). Entries with wildcards can match several names, allowing an endpoint to easily be included in several Partitions. Two Partition names with wildcards will match if either of them matches the other one according to `fnmatch`. That is, the matching is checked both ways. For example, consider the following configuration:

- A Publisher with Partition `part*`
- A Subscriber with Partition `partition*`

Even though `partition*` does not match `part*`, these Publisher and Subscriber will communicate between them because `part*` matches `partition*`.

Note that a Partition with name `*` will match any other partition **except the default Partition**.

Full example

Given a system with the following Partition configuration:

Participant_1	Pub_11	{“Partition_1”, “Partition_2”}
	Pub_12	{“*”}
Participant_2	Pub_21	{}
	Pub_22	{“Partition*”}
Participant_3	Subs_31	{“Partition_1”}
	Subs_32	{“Partition_2”}
	Subs_33	{“Partition_3”}
	Subs_34	{}

The endpoints will finally match the Partitions depicted on the following table. Note that `Pub_12` does not match the default Partition.

	Participant_1		Participant_2		Participant_3			
	Pub_11	Pub_12	Pub_21	Pub_22	Subs_31	Subs_32	Subs_33	Subs_34
Partition_1	✓	✓	×	✓	✓	×	×	×
Partition_2	✓	✓	×	✓	×	✓	×	×
Partition_3	×	✓	×	✓	×	×	✓	×
{default}	×	×	✓	×	×	×	×	✓

The following table provides the communication matrix for the given example:

		Participant_1		Participant_2	
		Pub_11	Pub_12	Pub_21	Pub_22
Participant_3	Subs_31	✓	✓	×	✓
	Subs_32	✓	✓	×	✓
	Subs_33	×	✓	×	✓
	Subs_34	×	×	✓	×

The following piece of code shows the set of parameters needed for the use case depicted in this example.

C++

```
PublisherQos pub_11_qos;
pub_11_qos.partition().push_back("Partition_1");
pub_11_qos.partition().push_back("Partition_2");

PublisherQos pub_12_qos;
pub_12_qos.partition().push_back("*");

PublisherQos pub_21_qos;
//No partitions defined for pub_21

PublisherQos pub_22_qos;
pub_22_qos.partition().push_back("Partition*");

SubscriberQos subs_31_qos;
subs_31_qos.partition().push_back("Partition_1");

SubscriberQos subs_32_qos;
subs_32_qos.partition().push_back("Partition_2");

SubscriberQos subs_33_qos;
subs_33_qos.partition().push_back("Partition_3");

SubscriberQos subs_34_qos;
//No partitions defined for subs_34
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_writer profile_name="pub_11">
        <qos>
            <partition>
                <names>
                    <name>Partition_1</name>
                    <name>Partition_2</name>
                </names>
            </partition>
        </qos>
    </data_writer>

    <data_writer profile_name="pub_12">
        <qos>
            <partition>
                <names>
                    <name>*</name>
                </names>
            </partition>
        </qos>
    </data_writer>

    <data_writer profile_name="pub_22">
        <qos>
            <partition>
                <names>
                    <name>Partition*</name>
                </names>
            </partition>
```

6.17.3 Publisher

A publication is defined by the association of a *DataWriter* to a *Publisher*. To start publishing the values of a data instance, the application creates a new DataWriter in a Publisher. This DataWriter will be bound to the *Topic* that describes the data type that is being transmitted. Remote subscriptions that match with this Topic will be able to receive the data value updates from the DataWriter.

Publisher

The *Publisher* acts on behalf of one or several *DataWriter* objects that belong to it. It serves as a container that allows grouping different DataWriter objects under a common configuration given by the *PublisherQos* of the Publisher.

DataWriter objects that belong to the same Publisher do not have any other relation among each other beyond the PublisherQos of the Publisher and act independently otherwise. Specifically, a Publisher can host DataWriter objects for different *Topics* and data types.

PublisherQos

`PublisherQos` controls the behavior of the `Publisher`. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
PresentationQosPolicy	`presentation()`	Yes
PartitionQosPolicy	`partition()`	Yes
GroupDataQosPolicy	`group_data()`	Yes
EntityFactoryQosPolicy	`entity_factory()`	Yes

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

The QoS value of a previously created Publisher can be modified using the `Publisher::set_qos()` member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher with default PublisherQos
Publisher* publisher =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
```

(continues on next page)

(continued from previous page)

```
PublisherQos qos = publisher->get_qos();

// Modify QoS attributes
// (...)

// Assign the new Qos to the object
publisher->set_qos(qos);
```

Default PublisherQos

The default *PublisherQos* refers to the value returned by the *get_default_publisher_qos()* member function on the DomainParticipant instance. The special value PUBLISHER_QOS_DEFAULT can be used as QoS argument on *create_publisher()* or *Publisher::set_qos()* member functions to indicate that the current default PublisherQos should be used.

When the system starts, the default PublisherQos is equivalent to the default constructed value *PublisherQos()*. The default PublisherQos can be modified at any time using the *set_default_publisher_qos()* member function on the *DomainParticipant* instance. Modifying the default PublisherQos will not affect already existing *Publisher* instances.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
PublisherQos qos_type1 = participant->get_default_publisher_qos();

// Modify QoS attributes
// (...)

// Set as the new default PublisherQos
if (participant->set_default_publisher_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Publisher with the new default PublisherQos.
Publisher* publisher_with_qos_type1 =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher_with_qos_type1)
{
    // Error
    return;
}
```

(continues on next page)

(continued from previous page)

```
// Get the current QoS or create a new one from scratch
PublisherQos qos_type2;

// Modify QoS attributes
// (...)

// Set as the new default PublisherQos
if (participant->set_default_publisher_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Publisher with the new default PublisherQos.
Publisher* publisher_with_qos_type2 =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default PublisherQos to the original default constructed values
if (participant->set_default_publisher_qos(PUBLISHER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (participant->set_default_publisher_qos(PublisherQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

set_default_publisher_qos() member function also accepts the special value PUBLISHER_QOS_DEFAULT as input argument. This will reset the current default PublisherQos to default constructed value *PublisherQos()*.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a custom PublisherQos
```

(continues on next page)

(continued from previous page)

```
PublisherQos custom_qos;

// Modify QoS attributes
// (...)

// Create a publisher with a custom PublisherQos
Publisher* publisher = participant->create_publisher(custom_qos);
if (nullptr == publisher)
{
    // Error
    return;
}

// Set the QoS on the publisher to the default
if (publisher->set_qos(PUBLISHER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (publisher->set_qos(participant->get_default_publisher_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `PUBLISHER_QOS_DEFAULT` has different meaning depending on where it is used:

- On *`create_publisher()`* and *`Publisher::set_qos()`* it refers to the default *PublisherQos*. as returned by *`get_default_publisher_qos()`*.
- On *`set_default_publisher_qos()`* it refers to the default constructed *`PublisherQos()`*.

PublisherListener

`PublisherListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *Publisher*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`PublisherListener` inherits from *DataWriterListener*. Therefore, it has the ability to react to all events that are reported to the *DataWriter*. Since events are always notified to the most specific Entity Listener that can handle the event, callbacks that *`PublisherListener`* inherits from DataWriterListener will only be called if the triggering DataWriter has no Listener attached, or if the callback is disabled by the *`StatusMask`* on the DataWriter.

`PublisherListener` does not add any new callback. Please, refer to the *DataWriterListener* for the list of inherited callbacks and override examples.

Creating a Publisher

A *Publisher* always belongs to a *DomainParticipant*. Creation of a Publisher is done with the *create_publisher()* member function on the DomainParticipant instance, that acts as a factory for the Publisher.

Mandatory arguments are:

- The *PublisherQos* describing the behavior of the Publisher. If the provided value is PUBLISHER_QOS_DEFAULT, the value of the *Default PublisherQos* is used.

Optional arguments are:

- A Listener derived from *PublisherListener*, implementing the callbacks that will be triggered in response to events and state changes on the Publisher. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the PublisherListener. By default all events are enabled.

create_publisher() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher with default PublisherQos and no Listener
// The value PUBLISHER_QOS_DEFAULT is used to denote the default QoS.
Publisher* publisher_with_default_qos =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher_with_default_qos)
{
    // Error
    return;
}

// A custom PublisherQos can be provided to the creation method
PublisherQos custom_qos;

// Modify QoS attributes
// (...)

Publisher* publisher_with_custom_qos =
        participant->create_publisher(custom_qos);
if (nullptr == publisher_with_custom_qos)
{
    // Error
    return;
}

// Create a Publisher with default QoS and a custom Listener.
```

(continues on next page)

(continued from previous page)

```
// CustomPublisherListener inherits from PublisherListener.
// The value PUBLISHER_QOS_DEFAULT is used to denote the default QoS.
CustomPublisherListener custom_listener;
Publisher* publisher_with_default_qos_and_custom_listener =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT, &custom_listener);
if (nullptr == publisher_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a Publisher

Instead of using a *PublisherQos*, the name of a profile can be used to create a Publisher with the *create_publisher_with_profile()* member function on the DomainParticipant instance.

Mandatory arguments are:

- A string with the name that identifies the Publisher.

Optional arguments are:

- A Listener derived from *PublisherListener*, implementing the callbacks that will be triggered in response to events and state changes on the Publisher. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the PublisherListener. By default all events are enabled.

create_publisher_with_profile() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher using a profile and no Listener
Publisher* publisher_with_profile =
        participant->create_publisher_with_profile("publisher_profile");
if (nullptr == publisher_with_profile)
{
    // Error
```

(continues on next page)

(continued from previous page)

```
    return;
}

// Create a Publisher using a profile and a custom Listener.
// CustomPublisherListener inherits from PublisherListener.
CustomPublisherListener custom_listener;
Publisher* publisher_with_profile_and_custom_listener =
        participant->create_publisher_with_profile("publisher_profile", &custom_
↪listener);
if (nullptr == publisher_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a Publisher

A Publisher can be deleted with the *delete_publisher()* member function on the DomainParticipant instance where the Publisher was created.

Note: A Publisher can only be deleted if all Entities belonging to the Publisher (DataWriters) have already been deleted. Otherwise, the function will issue an error and the Publisher will not be deleted. This can be performed by using the *delete_contained_entities()* member function of the *Publisher*.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher
Publisher* publisher =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher)
{
    // Error
    return;
}

// Use the Publisher to communicate
// (...)

// Delete the entities the Publisher created.
if (publisher->delete_contained_entities() != ReturnCode_t::RETCODE_OK)
{
```

(continues on next page)

(continued from previous page)

```
    // Publisher failed to delete the entities it created.
    return;
}

// Delete the Publisher
if (participant->delete_publisher(publisher) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

DataWriter

A *DataWriter* is attached to exactly one *Publisher* that acts as a factory for it. Additionally, each DataWriter is bound to a single *Topic* since its creation. This Topic must exist prior to the creation of the DataWriter, and must be bound to the data type that the DataWriter wants to publish.

The effect of creating a new DataWriter in a Publisher for a specific Topic is to initiate a new publication with the name and data type described by the Topic.

Once the DataWriter is created, the application can inform of changes in the data value using the *write()* member function on the DataWriter. These changes will be transmitted to all subscriptions matched with this publication.

DataWriterQos

DataWriterQos controls the behavior of the DataWriter. Internally it contains the following *QosPolicy* objects:

QosPolicy class	Accessor/Mutator	Mutable
DurabilityQosPolicy	*durability()*	No
DurabilityServiceQosPolicy	*durability_service()*	No
DeadlineQosPolicy	*deadline()*	Yes
LatencyBudgetQosPolicy	*latency_budget()*	Yes
LivelinessQosPolicy	*liveliness()*	No
ReliabilityQosPolicy	*reliability()*	No (*)
DestinationOrderQosPolicy	*destination_order()*	No
HistoryQosPolicy	*history()*	No
ResourceLimitsQosPolicy	*resource_limits()*	No
TransportPriorityQosPolicy	*transport_priority()*	Yes
LifespanQosPolicy	*lifespan()*	Yes
UserDataQosPolicy	*user_data()*	Yes
OwnershipQosPolicy	*ownership()*	No
OwnershipStrengthQosPolicy	*ownership_strength()*	Yes
WriterDataLifecycleQosPolicy	*writer_data_lifecycle()*	Yes
PublishModeQosPolicy	*publish_mode()*	No
DataRepresentationQosPolicy	*representation()*	No
PropertyPolicyQos	*properties()*	No
RTPSReliableWriterQos	*reliable_writer_qos()*	Yes (*)
RTPSEndpointQos	*endpoint()*	No
WriterResourceLimitsQos	*writer_resource_limits()*	No
DataSharingQosPolicy	*data_sharing()*	No

The following non-consolidated property-assigned QoS apply to DataWriters:

Property name	Non-consolidated QoS
fastdds.push_mode	*DataWriter operating mode QoS Policy*
partitions	*Endpoint Partitions*

Refer to the detailed description of each *QosPolicy* class for more information about their usage and default values.

Note: Reliability kind (whether the publication is reliable or best effort) is not mutable. However, the `max_blocking_time` data member of *ReliabilityQosPolicy* can be modified any time.

Note: Not all data members of RTPSReliableWriterQos are mutable, please refer to *RTPSReliableWriterQos* for more information.

The QoS value of a previously created DataWriter can be modified using the *DataWriter::set_qos()* member function.

```
// Create a DataWriter with default DataWriterQos
DataWriter* data_writer =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
DataWriterQos qos = data_writer->get_qos();

// Modify QoS attributes
// (...)

// Assign the new Qos to the object
data_writer->set_qos(qos);
```

Default DataWriterQos

The default *DataWriterQos* refers to the value returned by the *get_default_datawriter_qos()* member function on the Publisher instance. The special value `DATAWRITER_QOS_DEFAULT` can be used as QoS argument on *create_datawriter()* or *DataWriter::set_qos()* member functions to indicate that the current default DataWriterQos should be used.

When the system starts, the default DataWriterQos is equivalent to the default constructed value *DataWriterQos()*. The default DataWriterQos can be modified at any time using the *set_default_datawriter_qos()* member function on the Publisher instance. Modifying the default DataWriterQos will not affect already existing DataWriter instances.

```
// Get the current QoS or create a new one from scratch
DataWriterQos qos_type1 = publisher->get_default_datawriter_qos();
```

(continues on next page)

(continued from previous page)

```
// Modify QoS attributes
// (...)

// Set as the new default DataWriterQos
if (publisher->set_default_datawriter_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DataWriter with the new default DataWriterQos.
DataWriter* data_writer_with_qos_type1 =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer_with_qos_type1)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
DataWriterQos qos_type2;

// Modify QoS attributes
// (...)

// Set as the new default DataWriterQos
if (publisher->set_default_datawriter_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DataWriter with the new default DataWriterQos.
DataWriter* data_writer_with_qos_type2 =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DataWriterQos to the original default constructed values
if (publisher->set_default_datawriter_qos(DATAWRITER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (publisher->set_default_datawriter_qos(DataWriterQos())
```

(continues on next page)

(continued from previous page)

```
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

set_default_datawriter_qos() member function also accepts the special value DATAWRITER_QOS_DEFAULT as input argument. This will reset the current default DataWriterQos to default constructed value *DataWriterQos()*.

```
// Create a custom DataWriterQos
DataWriterQos custom_qos;

// Modify QoS attributes
// (...)

// Create a DataWriter with a custom DataWriterQos
DataWriter* data_writer = publisher->create_datawriter(topic, custom_qos);
if (nullptr == data_writer)
{
    // Error
    return;
}

// Set the QoS on the DataWriter to the default
if (data_writer->set_qos(DATAWRITER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (data_writer->set_qos(publisher->get_default_datawriter_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value DATAWRITER_QOS_DEFAULT has different meaning depending on where it is used:

- On *create_datawriter()* and *DataWriter::set_qos()* it refers to the default DataWriterQos as returned by *get_default_datawriter_qos()*.
- On *set_default_datawriter_qos()* it refers to the default constructed *DataWriterQos()*.

DataWriterListener

DataWriterListener is an abstract class defining the callbacks that will be triggered in response to state changes on the *DataWriter*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

DataWriterListener defines the following callbacks:

- *on_publication_matched()*: The DataWriter has found a *DataReader* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a DataReader that was previously considered to be matched.
- *on_offered_deadline_missed()*: The DataWriter failed to provide data within the deadline period configured on its *DataWriterQos*. It will be called for each deadline period and data instance for which the DataWriter failed to provide data.
- *on_offered_incompatible_qos()*: The DataWriter has found a DataReader that matches the Topic and has a common partition, but with a requested QoS that is incompatible with the one defined on the DataWriter.
- *on_liveliness_lost()*: The DataWriter did not respect the liveliness configuration on its DataWriterQos, and therefore, DataReader entities will consider the DataWriter as no longer *active*.
- *on_unacknowledged_sample_removed()*: The Datawriter has removed a sample that has not been acknowledged by every matched DataReader.

on_unacknowledged_sample_removed callback

on_unacknowledged_sample_removed() non-standard callback notifies the user when a sample has been removed without being sent/received by the matched DataReaders. This could happen in constrained networks or if the publication throughput is too demanding. This callback can be used to detect these situations so the publishing application can apply some solution to ease this issue like reducing the publication rate.

The criteria to consider that a sample has been removed without being acknowledged depends on the *ReliabilityQosPolicy*:

- *BEST_EFFORT_RELIABILITY_QOS* DataWriters will notify that a sample has been removed while unacknowledged if the sample has not been sent through the transport.
- *RELIABLE_RELIABILITY_QOS* DataWriters consider samples to have been removed unacknowledged if not every matched DataReader has confirmed its reception by sending the corresponding meta-traffic `ACK` message. Consequently, a sample that is notified as removed unacknowledged might be received by one or more DataReaders, but not by every matched one, or at least, the `ACK` message has not been received at the moment of sample removal. A race condition is inevitable in this case, because when the sample is removed, the `ACK` from some matched DataReader is missing, but that means that it might have been lost in the transmission or that the message is still coming through and it will be received after the sample removal. Thus, this criteria may include false positives, but from the user's point of view, it is more meaningful to know when the sample has not been acknowledged by every matched DataReader even if some samples are erroneously notified.

A specific case must be considered for reliable DataWriters with *DisablePositiveACKsQosPolicy* enabled. This policy disables the sending of positive `ACK` messages, unless the sample has been lost in which case the matched DataReader notifies the loss with a negative `NACK` message. If no `NACK` has been received in the time defined in the QoS policy, the sample is considered to be received. Again, this is prone to race conditions because the `NACK` message might be on its way or have been lost in the network. For this specific case, where `ACK` messages are not going to be received, the reliable DataWriter uses the same criteria as the best effort DataWriter.

```
class CustomDataWriterListener : public DataWriterListener
{

public:

    CustomDataWriterListener()
        : DataWriterListener()
    {
    }

    virtual ~CustomDataWriterListener()
    {
    }

    void on_publication_matched(
            DataWriter* writer,
            const PublicationMatchedStatus& info) override
    {
        static_cast<void>(writer);
        if (info.current_count_change == 1)
        {
            std::cout << "Matched a remote Subscriber for one of our Topics" <<␣
↪std::endl;
        }
        else if (info.current_count_change == -1)
        {
            std::cout << "Unmatched a remote Subscriber" << std::endl;
        }
    }

    void on_offered_deadline_missed(
            DataWriter* writer,
            const OfferedDeadlineMissedStatus& status) override
    {
        static_cast<void>(writer);
        static_cast<void>(status);
        std::cout << "Some data could not be delivered on time" << std::endl;
    }

    void on_offered_incompatible_qos(
            DataWriter* /*writer*/,
            const OfferedIncompatibleQosStatus& status) override
    {
        std::cout << "Found a remote Topic with incompatible QoS (QoS ID: " << status.
↪last_policy_id <<
            ")" << std::endl;
    }

    void on_liveliness_lost(
            DataWriter* writer,
            const LivelinessLostStatus& status) override
    {
        static_cast<void>(writer);
```

(continues on next page)

(continued from previous page)

```
        static_cast<void>(status);
        std::cout << "Liveliness lost. Matched Subscribers will consider us offline" <<␣
↪std::endl;
    }

    void on_unacknowledged_sample_removed(
            DataWriter* writer,
            const InstanceHandle_t& instance) override
    {
        static_cast<void>(writer);
        static_cast<void>(instance);
        std::cout << "Sample removed unacknowledged" << std::endl;
    }

};
```

Creating a DataWriter

A *DataWriter* always belongs to a *Publisher*. Creation of a DataWriter is done with the *create_datawriter()* member function on the Publisher instance, that acts as a factory for the DataWriter.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- The *DataWriterQos* describing the behavior of the DataWriter. If the provided value is `DATAWRITER_QOS_DEFAULT`, the value of the *Default DataWriterQos* is used.

Optional arguments are:

- A Listener derived from *DataWriterListener*, implementing the callbacks that will be triggered in response to events and state changes on the DataWriter. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the DataWriterListener. By default all events are enabled.

create_datawriter() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DataWriter with default DataWriterQos and no Listener
// The value DATAWRITER_QOS_DEFAULT is used to denote the default QoS.
DataWriter* data_writer_with_default_qos =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer_with_default_qos)
{
    // Error
    return;
}

// A custom DataWriterQos can be provided to the creation method
DataWriterQos custom_qos;

// Modify QoS attributes
// (...)
```

(continues on next page)

(continued from previous page)

```
DataWriter* data_writer_with_custom_qos =
        publisher->create_datawriter(topic, custom_qos);
if (nullptr == data_writer_with_custom_qos)
{
    // Error
    return;
}

// Create a DataWriter with default QoS and a custom Listener.
// CustomDataWriterListener inherits from DataWriterListener.
// The value DATAWRITER_QOS_DEFAULT is used to denote the default QoS.
CustomDataWriterListener custom_listener;
DataWriter* data_writer_with_default_qos_and_custom_listener =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT, &custom_listener);
if (nullptr == data_writer_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a DataWriter

Instead of using a DataWriterQos, the name of a profile can be used to create a DataWriter with the *create_datawriter_with_profile()* member function on the Publisher instance.

Mandatory arguments are:

- A Topic bound to the data type that will be transmitted.
- A string with the name that identifies the DataWriter.

Optional arguments are:

- A Listener derived from DataWriterListener, implementing the callbacks that will be triggered in response to events and state changes on the DataWriter. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the DataWriterListener. By default all events are enabled.

create_datawriter_with_profile() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DataWriter using a profile and no Listener
DataWriter* data_writer_with_profile =
        publisher->create_datawriter_with_profile(topic, "data_writer_profile");
if (nullptr == data_writer_with_profile)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Create a DataWriter using a profile and a custom Listener.
// CustomDataWriterListener inherits from DataWriterListener.
CustomDataWriterListener custom_listener;
DataWriter* data_writer_with_profile_and_custom_listener =
        publisher->create_datawriter_with_profile(topic, "data_writer_profile", &custom_
↪listener);
if (nullptr == data_writer_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Creating a DataWriter with a custom PayloadPool

A custom *PayloadPool* can be passed as an argument during the creation of a *DataWriter*. This allows for customizing the management of the information exchanged between DataWriters and DataReaders. The same configuration can be set in the *opposite endpoint*.

```
// A DataWriterQos must be provided to the creation method
DataWriterQos qos;

// Create PayloadPool
std::shared_ptr<eprosima::fastrtps::rtps::IPayloadPool> payload_pool =
        std::dynamic_pointer_cast<eprosima::fastrtps::rtps::IPayloadPool>(std::make_
↪shared<CustomPayloadPool>());

DataWriter* data_writer = publisher->create_datawriter(topic, qos, nullptr,␣
↪StatusMask::all(), payload_pool);
if (nullptr == data_writer)
{
    // Error
    return;
}
```

This configuration can be performed also in the *RTPS layer*. The *customization example* applies both layers.

Deleting a DataWriter

A DataWriter can be deleted with the *delete_datawriter()* member function on the *Publisher* instance where the DataWriter was created.

```
// Create a DataWriter
DataWriter* data_writer =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Use the DataWriter to communicate
// (...)

// Delete the DataWriter
if (publisher->delete_datawriter(data_writer) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Publishing data

The user informs of a change in the value of a data instance with the *write()* member function on the *DataWriter*. This change will then be communicated to every *DataReader* matched with the DataWriter. As a side effect, this operation asserts liveliness on the DataWriter itself, the *Publisher* and the *DomainParticipant*.

The function takes two arguments:

- A pointer to the data instance with the new values.
- The handler to the instance.

An empty (i.e., default constructed *InstanceHandle_t*) instance handler can be used for the argument handle. This indicates that the identity of the instance should be automatically deduced from the key of the instance data. Alternatively, the member function *write()* is overloaded to take only the pointer to the data instance, which will always deduced the identity from the key of the instance data.

If the handle is not empty, then it must correspond to the value obtained with the *getKey()* of the *TypeSupport* instance. Otherwise the write function will fail with `RETCODE_PRECONDITION_NOT_MET`.

```
// Register the data type in the DomainParticipant.
TypeSupport custom_type_support(new CustomDataType());
custom_type_support.register_type(participant, custom_type_support.get_type_name());

// Create a Topic with the registered type.
Topic* custom_topic =
        participant->create_topic("topic_name", custom_type_support.get_type_name(),␣
↪TOPIC_QOS_DEFAULT);
if (nullptr == custom_topic)
{
    // Error
    return;
}

// Create a DataWriter
DataWriter* data_writer =
        publisher->create_datawriter(custom_topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Get a data instance
void* data = custom_type_support->createData();

// Fill the data values
// (...)

// Publish the new value, deduce the instance handle
if (data_writer->write(data, eprosima::fastrtps::rtps::InstanceHandle_t()) != ReturnCode_
↪t::RETCODE_OK)
{
    // Error
    return;
}

// The data instance can be reused to publish new values,
// but delete it at the end to avoid leaks
custom_type_support->deleteData(data);
```

Blocking of the write operation

If the reliability kind is set to `RELIABLE` on the *DataWriterQos*, the *write()* operation may block. Specifically, if the limits specified in the configured resource limits have been reached, the *write()* operation will block waiting for space to become available. Under these circumstances, the reliability `max_blocking_time` configures the maximum time the write operation may block waiting. If `max_blocking_time` elapses before the DataWriter is able to store the modification without exceeding the limits, the write operation will fail and return `TIMEOUT`.

Borrowing a data buffer

When the user calls *write()* with a new sample value, the data is copied from the given sample to the DataWriter's memory. For large data types this copy can consume significant time and memory resources. Instead, the DataWriter can loan a sample from its memory to the user, and the user can fill this sample with the required values. When *write()* is called with such a loaned sample, the DataWriter does not copy its contents, as it already owns the buffer.

To use loaned data samples in publications, perform the following steps:

1. Get a reference to a loaned sample using *loan_sample()*.
2. Use the reference to build the data sample.
3. Write the sample using *write()*.

Once *write()* has been called with a loaned sample, the loan is considered returned, and it is not safe to make any changes on the contents of the sample.

If function *loan_sample()* is called but the sample is never written, the loan must be returned to the DataWriter using *discard_loan()*. Otherwise the DataWriter may run out of samples.

```
    // Borrow a data instance
    void* data = nullptr;
```

(continues on next page)

(continued from previous page)

```
    if (ReturnCode_t::RETCODE_OK == data_writer->loan_sample(data))
    {
        bool error = false;

        // Fill the data values
        // (...)

        if (error)
        {
            // Return the loan without publishing
            data_writer->discard_loan(data);
            return;
        }

        // Publish the new value
        if (data_writer->write(data, eprosima::fastrtps::rtps::InstanceHandle_t()) !=␣
↪ReturnCode_t::RETCODE_OK)
        {
            // Error
            return;
        }
    }

    // The data instance can be reused to publish new values,
    // but delete it at the end to avoid leaks
    custom_type_support->deleteData(data);
```

6.17.4 Subscriber

A subscription is defined by the association of a *DataReader* to a *Subscriber*. To start receiving updates of a publication, the application creates a new DataReader in a Subscriber. This DataReader will be bound to the *Topic* that describes the data type that is going to be received. The DataReader will then start receiving data value updates from remote publications that match this Topic.

When the Subscriber receives data, it informs the application that new data is available. Then, the application can use the DataReader to get the received data.

Fig. 7: Subscriber class diagram

Subscriber

The *Subscriber* acts on behalf of one or several *DataReader* objects that belong to it. It serves as a container that allows grouping different DataReader objects under a common configuration given by the *SubscriberQos* of the Subscriber.

DataReader objects that belong to the same Subscriber do not have any other relation among each other beyond the *SubscriberQos* of the Subscriber and act independently otherwise. Specifically, a Subscriber can host DataReader objects for different topics and data types.

SubscriberQos

SubscriberQos controls the behavior of the *Subscriber*. Internally it contains the following *QosPolicy* objects:

QosPolicy class	Accessor/Mutator	Mutable
PresentationQosPolicy	*presentation()*	Yes
PartitionQosPolicy	*partition()*	Yes
GroupDataQosPolicy	*group_data()*	Yes
EntityFactoryQosPolicy	*entity_factory()*	Yes

Refer to the detailed description of each *QosPolicy* class for more information about their usage and default values.

The QoS value of a previously created Subscriber can be modified using the *Subscriber::set_qos()* member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Subscriber with default SubscriberQos
Subscriber* subscriber =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
SubscriberQos qos = subscriber->get_qos();

// Modify QoS attributes
qos.entity_factory().autoenable_created_entities = false;

// Assign the new Qos to the object
subscriber->set_qos(qos);
```

Default SubscriberQos

The default *SubscriberQos* refers to the value returned by the *get_default_subscriber_qos()* member function on the *DomainParticipant* instance. The special value SUBSCRIBER_QOS_DEFAULT can be used as QoS argument on *create_subscriber()* or *Subscriber::set_qos()* member functions to indicate that the current default SubscriberQos should be used.

When the system starts, the default SubscriberQos is equivalent to the default constructed value *SubscriberQos()*. The default SubscriberQos can be modified at any time using the *set_default_subscriber_qos()* member function on the DomainParticipant instance. Modifying the default SubscriberQos will not affect already existing Subscriber instances.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
SubscriberQos qos_type1 = participant->get_default_subscriber_qos();

// Modify QoS attributes
// (...)

// Set as the new default SubscriberQos
if (participant->set_default_subscriber_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Subscriber with the new default SubscriberQos.
Subscriber* subscriber_with_qos_type1 =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber_with_qos_type1)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
SubscriberQos qos_type2;

// Modify QoS attributes
// (...)

// Set as the new default SubscriberQos
if (participant->set_default_subscriber_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Create a Subscriber with the new default SubscriberQos.
Subscriber* subscriber_with_qos_type2 =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default SubscriberQos to the original default constructed values
if (participant->set_default_subscriber_qos(SUBSCRIBER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (participant->set_default_subscriber_qos(SubscriberQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

set_default_subscriber_qos() member function also accepts the special value SUBSCRIBER_QOS_DEFAULT as input argument. This will reset the current default SubscriberQos to default constructed value *SubscriberQos()*.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a custom SubscriberQos
SubscriberQos custom_qos;

// Modify QoS attributes
// (...)

// Create a subscriber with a custom SubscriberQos
Subscriber* subscriber = participant->create_subscriber(custom_qos);
if (nullptr == subscriber)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Set the QoS on the subscriber to the default
if (subscriber->set_qos(SUBSCRIBER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (subscriber->set_qos(participant->get_default_subscriber_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `SUBSCRIBER_QOS_DEFAULT` has different meaning depending on where it is used:

- On *create_subscriber()* and *Subscriber::set_qos()* it refers to the default SubscriberQos as returned by *get_default_subscriber_qos()*.
- On *set_default_subscriber_qos()* it refers to the default constructed *SubscriberQos()*.

SubscriberListener

SubscriberListener is an abstract class defining the callbacks that will be triggered in response to state changes on the *Subscriber*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

SubscriberListener inherits from *DataReaderListener*. Therefore, it has the ability to react to all events that are reported to the *DataReader*. Since events are always notified to the most specific Entity Listener that can handle the event, callbacks that SubscriberListener inherits from DataReaderListener will only be called if the triggering DataReader has no Listener attached, or if the callback is disabled by the *StatusMask* on the DataReader.

Additionally, SubscriberListener adds the following callback:

- *on_data_on_readers()*: New data is available on any DataReader belonging to this Subscriber. There is no queuing of invocations to this callback, meaning that if several new data changes are received at once, only one callback invocation may be issued for all of them, instead of one per change. If the application is retrieving the received data on this callback, it must keep *reading data* until no new changes are left.

Important: For more information about callbacks and its hierarchy, please refer to *Listener*.

```
class CustomSubscriberListener : public SubscriberListener
{
```

(continues on next page)

(continued from previous page)

```
public:

    CustomSubscriberListener()
        : SubscriberListener()
    {
    }

    virtual ~CustomSubscriberListener()
    {
    }

    virtual void on_data_on_readers(
            Subscriber* sub)
    {
        static_cast<void>(sub);
        std::cout << "New data available" << std::endl;
    }

};
```

Creating a Subscriber

A *Subscriber* always belongs to a *DomainParticipant*. Creation of a Subscriber is done with the *create_subscriber()* member function on the DomainParticipant instance, that acts as a factory for the Subscriber.

Mandatory arguments are:

- The *SubscriberQos* describing the behavior of the Subscriber. If the provided value is SUBSCRIBER_QOS_DEFAULT, the value of the *Default SubscriberQos* is used.

Optional arguments are:

- A Listener derived from *SubscriberListener*, implementing the callbacks that will be triggered in response to events and state changes on the Subscriber. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the SubscriberListener. By default all events are enabled.

create_subscriber() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Subscriber with default SubscriberQos and no Listener
// The value SUBSCRIBER_QOS_DEFAULT is used to denote the default QoS.
Subscriber* subscriber_with_default_qos =
```

(continues on next page)

(continued from previous page)

```
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber_with_default_qos)
{
    // Error
    return;
}

// A custom SubscriberQos can be provided to the creation method
SubscriberQos custom_qos;

// Modify QoS attributes
// (...)

Subscriber* subscriber_with_custom_qos =
        participant->create_subscriber(custom_qos);
if (nullptr == subscriber_with_custom_qos)
{
    // Error
    return;
}

// Create a Subscriber with default QoS and a custom Listener.
// CustomSubscriberListener inherits from SubscriberListener.
// The value SUBSCRIBER_QOS_DEFAULT is used to denote the default QoS.
CustomSubscriberListener custom_listener;
Subscriber* subscriber_with_default_qos_and_custom_listener =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT, &custom_listener);
if (nullptr == subscriber_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a Subscriber

Instead of using a SubscriberQos, the name of a profile can be used to create a Subscriber with the *create_subscriber_with_profile()* member function on the DomainParticipant instance.

Mandatory arguments are:

- A string with the name that identifies the Subscriber.

Optional arguments are:

- A Listener derived from SubscriberListener, implementing the callbacks that will be triggered in response to events and state changes on the Subscriber. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the SubscriberListener. By default all events are enabled.

create_subscriber_with_profile() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Subscriber using a profile and no Listener
Subscriber* subscriber_with_profile =
        participant->create_subscriber_with_profile("subscriber_profile");
if (nullptr == subscriber_with_profile)
{
    // Error
    return;
}

// Create a Subscriber using a profile and a custom Listener.
// CustomSubscriberListener inherits from SubscriberListener.
CustomSubscriberListener custom_listener;
Subscriber* subscriber_with_profile_and_custom_listener =
        participant->create_subscriber_with_profile("subscriber_profile", &custom_
↪listener);
if (nullptr == subscriber_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a Subscriber

A Subscriber can be deleted with the *delete_subscriber()* member function on the DomainParticipant instance where the Subscriber was created.

Note: A Subscriber can only be deleted if all Entities belonging to the Subscriber (DataReaders) have already been deleted. Otherwise, the function will issue an error and the Subscriber will not be deleted. This can be performed by using the *delete_contained_entities()* member function of the *Subscriber*.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
```

(continues on next page)

(continued from previous page)

```
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Subscriber
Subscriber* subscriber =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}

// Use the Subscriber to communicate
// (...)

// Delete the entities the subscriber created
if (subscriber->delete_contained_entities() != ReturnCode_t::RETCODE_OK)
{
    // Subscriber failed to delete the entities it created
    return;
}

// Delete the Subscriber
if (participant->delete_subscriber(subscriber) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

DataReader

A *DataReader* is attached to exactly one *Subscriber* that acts as a factory for it. Additionally, each DataReader is bound to a single *Topic* since its creation. This Topic must exist prior to the creation of the DataReader, and must be bound to the data type that the DataReader wants to publish.

The effect of creating a new DataReader in a Subscriber for a specific Topic is to initiate a new subscription with the name and data type described by the Topic.

Once the DataReader is created, the application will be informed when changes in the data value are received from remote publications. These changes can then be retrieved using the *DataReader::read_next_sample()* or *DataReader::take_next_sample()* member functions of the DataReader.

DataReaderQos

DataReaderQoS controls the behavior of the *DataReader*. Internally it contains the following *QosPolicy* objects:

QosPolicy class	Accessor/Mutator	Mutable
DurabilityQosPolicy	*durability()*	No
DurabilityServiceQosPolicy	*durability_service()*	Yes
DeadlineQosPolicy	*deadline()*	Yes
LatencyBudgetQosPolicy	*latency_budget()*	Yes
LivelinessQosPolicy	*liveliness()*	No
ReliabilityQosPolicy	*reliability()*	No (*)
DestinationOrderQosPolicy	*destination_order()*	No
HistoryQosPolicy	*history()*	No
ResourceLimitsQosPolicy	*resource_limits()*	No
LifespanQosPolicy	*lifespan()*	Yes
UserDataQosPolicy	*user_data()*	Yes
OwnershipQosPolicy	*ownership()*	No
PropertyPolicyQos	*properties()*	Yes
RTPSEndpointQos	*endpoint()*	Yes
ReaderResourceLimitsQos	*reader_resource_limits()*	Yes
RTPSEndpointTimeBasedFilterQosPolicy	*time_based_filter()*	Yes
ReaderDataLifecycleQosPolicy	*reader_data_lifecycle()*	Yes
RTPSReliableReaderQos	*reliable_reader_qos()*	Yes (*)
TypeConsistencyQos	*type_consistency()*	Yes
DataSharingQosPolicy	*data_sharing()*	No
boolean	*expects_inline_qos()*	Yes

The following non-consolidated property-assigned QoS apply to DataReaders:

Property name	Non-consolidated QoS
partitions	*Endpoint Partitions*

Refer to the detailed description of each *QosPolicy* class for more information about their usage and default values.

Note: Reliability kind (whether the publication is reliable or best effort) is not mutable. However, the *max_blocking_time* data member of *ReliabilityQosPolicy* can be modified any time.

Note: Not all data members of RTPSReliableReaderQos are mutable, please refer to *RTPSReliableReaderQos* for more information.

The QoS value of a previously created DataReader can be modified using the *DataReader::set_qos()* member function.

```
// Create a DataReader with default DataReaderQos
DataReader* data_reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
    // Error
```

(continues on next page)

(continued from previous page)

```
    return;
}

// Get the current QoS or create a new one from scratch
DataReaderQos qos = data_reader->get_qos();

// Modify QoS attributes
// (...)

// Assign the new Qos to the object
data_reader->set_qos(qos);
```

Default DataReaderQos

The default DataReaderQos refers to the value returned by the *get_default_datareader_qos()* member function on the *Subscriber* instance. The special value DATAREADER_QOS_DEFAULT can be used as QoS argument on *create_datareader()* or *DataReader::set_qos()* member functions to indicate that the current default DataReaderQos should be used.

When the system starts, the default DataReaderQos is equivalent to the default constructed value *DataReaderQos()*. The default DataReaderQos can be modified at any time using the *set_default_datareader_qos()* member function on the Subscriber instance. Modifying the default DataReaderQos will not affect already existing *DataReader* instances.

```
// Get the current QoS or create a new one from scratch
DataReaderQos qos_type1 = subscriber->get_default_datareader_qos();

// Modify QoS attributes
// (...)

// Set as the new default DataReaderQos
if (subscriber->set_default_datareader_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DataReader with the new default DataReaderQos.
DataReader* data_reader_with_qos_type1 =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader_with_qos_type1)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
DataReaderQos qos_type2;

// Modify QoS attributes
// (...)
```

(continues on next page)

(continued from previous page)

```
// Set as the new default DataReaderQos
if (subscriber->set_default_datareader_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DataReader with the new default DataReaderQos.
DataReader* data_reader_with_qos_type2 =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DataReaderQos to the original default constructed values
if (subscriber->set_default_datareader_qos(DATAREADER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (subscriber->set_default_datareader_qos(DataReaderQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

set_default_datareader_qos() member function also accepts the special value DATAREADER_QOS_DEFAULT as input argument. This will reset the current default DataReaderQos to default constructed value *DataReaderQos()*.

```
// Create a custom DataReaderQos
DataReaderQos custom_qos;

// Modify QoS attributes
// (...)

// Create a DataWriter with a custom DataReaderQos
DataReader* data_reader = subscriber->create_datareader(topic, custom_qos);
if (nullptr == data_reader)
{
    // Error
    return;
}

// Set the QoS on the DataWriter to the default
if (data_reader->set_qos(DATAREADER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
```

(continues on next page)

(continued from previous page)

```
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (data_reader->set_qos(subscriber->get_default_datareader_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `DATAREADER_QOS_DEFAULT` has different meaning depending on where it is used:

- On *`create_datareader()`* and *`DataReader::set_qos()`* it refers to the default DataReaderQos as returned by *`get_default_datareader_qos()`*.
- On *`set_default_datareader_qos()`* it refers to the default constructed *`DataReaderQos()`*.

DataReaderListener

`DataReaderListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *DataReader*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

DataReaderListener defines the following callbacks:

- *`on_data_available()`*: There is new data available for the application on the DataReader. There is no queuing of invocations to this callback, meaning that if several new data changes are received at once, only one callback invocation may be issued for all of them, instead of one per change. If the application is retrieving the received data on this callback, it must keep *reading data* until no new changes are left.
- *`on_subscription_matched()`*: The DataReader has found a *DataWriter* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a DataWriter that was previously considered to be matched. It is also triggered when a matched DataWriter has changed its *DataWriterQos*.
- *`on_requested_deadline_missed()`*: The DataReader did not receive data within the deadline period configured on its *DataReaderQos*. It will be called for each deadline period and data instance for which the DataReader missed data.
- *`on_requested_incompatible_qos()`*: The DataReader has found a DataWriter that matches the Topic and has a common partition, but with a QoS that is incompatible with the one defined on the DataReader.
- *`on_liveliness_changed()`*: The liveliness status of a matched DataWriter has changed. Either a DataWriter that was *inactive* has become *active* or the other way around.
- *`on_sample_rejected()`*: A received data sample was rejected. See *SampleRejectedStatus* for further information.
- *`on_sample_lost()`*: A data sample was lost and will never be received. See *SampleLostStatus* for further information.

Important: For more information about callbacks and its hierarchy, please refer to *Listener*.

```
class CustomDataReaderListener : public DataReaderListener
{

public:

    CustomDataReaderListener()
        : DataReaderListener()
    {
    }

    virtual ~CustomDataReaderListener()
    {
    }

    void on_data_available(
            DataReader* reader) override
    {
        static_cast<void>(reader);
        std::cout << "Received new data message" << std::endl;
    }

    void on_subscription_matched(
            DataReader* reader,
            const SubscriptionMatchedStatus& info) override
    {
        static_cast<void>(reader);
        if (info.current_count_change == 1)
        {
            std::cout << "Matched a remote DataWriter" << std::endl;
        }
        else if (info.current_count_change == -1)
        {
            std::cout << "Unmatched a remote DataWriter" << std::endl;
        }
    }

    void on_requested_deadline_missed(
            DataReader* reader,
            const eprosima::fastrtps::RequestedDeadlineMissedStatus& info) override
    {
        static_cast<void>(reader);
        static_cast<void>(info);
        std::cout << "Some data was not received on time" << std::endl;
    }

    void on_liveliness_changed(
            DataReader* reader,
            const eprosima::fastrtps::LivelinessChangedStatus& info) override
    {
```

(continues on next page)

(continued from previous page)

```
        static_cast<void>(reader);
        if (info.alive_count_change == 1)
        {
            std::cout << "A matched DataWriter has become active" << std::endl;
        }
        else if (info.not_alive_count_change == 1)
        {
            std::cout << "A matched DataWriter has become inactive" << std::endl;
        }
    }

    void on_sample_rejected(
            DataReader* reader,
            const eprosima::fastrtps::SampleRejectedStatus& info) override
    {
        static_cast<void>(reader);
        static_cast<void>(info);
        std::cout << "A received data sample was rejected" << std::endl;
    }

    void on_requested_incompatible_qos(
            DataReader* /*reader*/,
            const RequestedIncompatibleQosStatus& info) override
    {
        std::cout << "Found a remote Topic with incompatible QoS (QoS ID: " << info.last_
↪policy_id <<
            ")" << std::endl;
    }

    void on_sample_lost(
            DataReader* reader,
            const SampleLostStatus& info) override
    {
        static_cast<void>(reader);
        static_cast<void>(info);
        std::cout << "A data sample was lost and will not be received" << std::endl;
    }

};
```

Creating a DataReader

A *DataReader* always belongs to a *Subscriber*. Creation of a DataReader is done with the *create_datareader()* member function on the Subscriber instance, that acts as a factory for the DataReader.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- The *DataReaderQos* describing the behavior of the DataReader. If the provided value is `DATAREADER_QOS_DEFAULT`, the value of the *Default DataReaderQos* is used.

Optional arguments are:

- A Listener derived from *DataReaderListener*, implementing the callbacks that will be triggered in response to events and state changes on the DataReader. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the DataReaderListener. By default all events are enabled.

create_datareader() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DataReader with default DataReaderQos and no Listener
// The value DATAREADER_QOS_DEFAULT is used to denote the default QoS.
DataReader* data_reader_with_default_qos =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader_with_default_qos)
{
    // Error
    return;
}

// A custom DataReaderQos can be provided to the creation method
DataReaderQos custom_qos;

// Modify QoS attributes
// (...)

DataReader* data_reader_with_custom_qos =
        subscriber->create_datareader(topic, custom_qos);
if (nullptr == data_reader_with_custom_qos)
{
    // Error
    return;
}

// Create a DataReader with default QoS and a custom Listener.
// CustomDataReaderListener inherits from DataReaderListener.
// The value DATAREADER_QOS_DEFAULT is used to denote the default QoS.
CustomDataReaderListener custom_listener;
DataReader* data_reader_with_default_qos_and_custom_listener =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT, &custom_listener);
if (nullptr == data_reader_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a DataReader

Instead of using a DataReaderQos, the name of a profile can be used to create a DataReader with the *create_datareader_with_profile()* member function on the Subscriber instance.

Mandatory arguments are:

- A Topic bound to the data type that will be transmitted.
- A string with the name that identifies the DataReader.

Optional arguments are:

- A Listener derived from DataReaderListener, implementing the callbacks that will be triggered in response to events and state changes on the DataReader. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the DataReaderListener. By default all events are enabled.

create_datareader_with_profile() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DataReader using a profile and no Listener
DataReader* data_reader_with_profile =
        subscriber->create_datareader_with_profile(topic, "data_reader_profile");
if (nullptr == data_reader_with_profile)
{
    // Error
    return;
}

// Create a DataReader using a profile and a custom Listener.
// CustomDataReaderListener inherits from DataReaderListener.
CustomDataReaderListener custom_listener;
DataReader* data_reader_with_profile_and_custom_listener =
        subscriber->create_datareader_with_profile(topic, "data_reader_profile", &custom_
↪listener);
if (nullptr == data_reader_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Creating a DataWriter with a custom PayloadPool

A custom *PayloadPool* can be passed as an argument during the creation of a *DataReader*. This allows for customizing the management of the information exchanged between DataWriters and DataReaders. The same configuration can be set in the *opposite endpoint*.

```
// A DataReaderQos must be provided to the creation method
DataReaderQos qos;

// Create PayloadPool
std::shared_ptr<CustomPayloadPool> payload_pool = std::make_shared<CustomPayloadPool>();

DataReader* data_reader = subscriber->create_datareader(topic, qos, nullptr,␣
↪StatusMask::all(), payload_pool);
if (nullptr == data_reader)
{
    // Error
    return;
}
```

This configuration can be performed also in the *RTPS layer*. The *customization example* applies both layers.

Deleting a DataReader

A DataReader can be deleted with the *delete_datareader()* member function on the *Subscriber* instance where the DataReader was created.

Note: A DataReader can only be deleted if all Entities belonging to the DataReader (QueryConditions) have already been deleted. Otherwise, the function will issue an error and the DataReader will not be deleted. This can be performed by using the *delete_contained_entities()* member function of the *DataReader*.

```
// Create a DataReader
DataReader* data_reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
    // Error
    return;
}

// Use the DataReader to communicate
// (...)

// Delete the entities the DataReader created
if (data_reader->delete_contained_entities() != ReturnCode_t::RETCODE_OK)
{
    // DataReader failed to delete the entities it created.
    return;
}

// Delete the DataReader
```

(continues on next page)

(continued from previous page)

```
if (subscriber->delete_datareader(data_reader) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

SampleInfo

When a sample is retrieved from the *DataReader*, in addition to the sample data, a *SampleInfo* instance is returned. This object contains additional information that complements the returned data value and helps on it interpretation. For example, if the *valid_data* value is `false`, the DataReader is not informing the application about a new value in the data instance, but a change on its status, and the returned data value must be discarded.

Please, refer to the section *Accessing received data* for more information regarding how received data can be accessed on the DataReader.

The following sections describe the data members of *SampleInfo* and the meaning of each one in relation to the returned sample data.

- *sample_state*
- *view_state*
- *instance_state*
- *disposed_generation_count*
- *no_writers_generation_count*
- *sample_rank*
- *generation_rank*
- *absolute_generation_rank*
- *source_timestamp*
- *instance_handle*
- *publication_handle*
- *valid_data*
- *sample_identity*
- *related_sample_identity*

sample_state

sample_state indicates whether or not the corresponding data sample has already been read previously. It can take one of these values:

- **READ**: This is the first time this data sample has been retrieved.
- **NOT_READ**: The data sample has already been *read* or *taken* previously.

view_state

view_state indicates whether or not this is the very first sample of this data instance that the DataReader retrieves. It can take one of these values:

- **NEW**: This is the first time a sample of this instance is retrieved.
- **NOT_NEW**: Other samples of this instance have been retrieved previously.

instance_state

instance_state indicates whether the instance is currently in existence or it has been disposed. In the latter case, it also provides information about the reason for the disposal. It can take one of these values:

- **ALIVE**: The instance is currently in existence.
- **NOT_ALIVE_DISPOSED**: A remote *DataWriter* disposed the instance.
- **NOT_ALIVE_NO_WRITERS**: The DataReader disposed the instance because no remote DataWriter that was publishing the instance is *alive*.

disposed_generation_count

disposed_generation_count indicates the number of times the instance had become alive after it was disposed.

no_writers_generation_count

no_writers_generation_count indicates the number of times the instance had become alive after it was disposed as `NOT_ALIVE_NO_WRITERS`.

sample_rank

sample_rank indicates the number of samples of the same instance that have been received after this one. For example, a value of `5` means that there are 5 newer samples available on the DataReader.

Note: Currently the *sample_rank* is not implemented, and its value is always set to `0`. It will be implemented on a future release of *Fast DDS*.

---

generation_rank

generation_rank indicates the number of times the instance was disposed and become alive again between the time the sample was received and the time the most recent sample of the same instance that is still held in the collection was received.

Note: Currently the *generation_rank* is not implemented, and its value is always set to `0`. It will be implemented on a future release of *Fast DDS*.

---

absolute_generation_rank

`absolute_generation_rank` indicates the number of times the instance was disposed and become alive again between the time the sample was received and the time the most recent sample of the same instance (which may not be in the collection) was received.

Note: Currently the `absolute_generation_rank` is not implemented, and its value is always set to `0`. It will be implemented on a future release of *Fast DDS*.

---

source_timestamp

`source_timestamp` holds the time stamp provided by the DataWriter when the sample was published.

instance_handle

`instance_handle` handles of the local instance.

publication_handle

`publication_handle` handles of the DataWriter that published the data change.

valid_data

`valid_data` is a boolean that indicates whether the data sample contains a change in the value or not. Samples with this value set to false are used to communicate a change in the instance status, e.g., a change in the liveliness of the instance. In this case, the data sample should be dismissed as all the relevant information is in the data members of SampleInfo.

sample_identity

`sample_identity` is an extension for requester-replier configuration. It contains the DataWriter and the sequence number of the current message, and it is used by the replier to fill the *related_sample_identity* when it sends the reply.

related_sample_identity

`related_sample_identity` is an extension for requester-replier configuration. On reply messages, it contains the *sample_identity* of the related request message. It is used by the requester to be able to link each reply to the appropriate request.

Accessing received data

The application can access and consume the data values received on the *DataReader* by *reading* or *taking*.

- **Reading** is done with any of the following member functions:
  - *DataReader::read_next_sample()* reads the next, non-previously accessed data value available on the DataReader, and stores it in the provided data buffer.
  - *DataReader::read()*, *DataReader::read_instance()*, and *DataReader::read_next_instance()* provide mechanisms to get a collection of samples matching certain conditions.
- **Taking** is done with any of the following member functions:
  - *DataReader::take_next_sample()* reads the next, non-previously accessed data value available on the DataReader, and stores it in the provided data buffer.
  - *DataReader::take()*, *DataReader::take_instance()*, and *DataReader::take_next_instance()* provide mechanisms to get a collection of samples matching certain conditions.

  When taking data, the returned samples are also removed from the DataReader, so they are no longer accessible.

When there is no data in the DataReader matching the required conditions, all the operations will return `NO_DATA` and output parameter will remain unchanged.

In addition to the data values, the data access operations also provide SampleInfo instances with additional information that help interpreting the returned data values, like the originating *DataWriter* or the publication time stamp. Please, refer to the *SampleInfo* section for an extensive description of its contents.

Loaning and Returning Data and SampleInfo Sequences

The *DataReader::read()* and *DataReader::take()* operations (and their variants) return information to the application in two sequences:

- Received DDS data samples in a sequence of the data type
- Corresponding information about each DDS sample in a SampleInfo sequence

These sequences are parameters that are passed by the application code into the *DataReader::read()* and *DataReader::take()* operations. When the passed sequences are empty (they are initialized but have a maximum length of 0), the middleware will fill those sequences with memory directly loaned from the receive queue itself. There is no copying of the data or SampleInfo when the contents of the sequences are loaned. This is certainly the most efficient way for the application code to retrieve the data.

When doing so, however, the code must return the loaned sequences back to the middleware, so that they can be reused by the receive queue. If the application does not return the loan by calling the *DataReader::return_loan()* operation, then Fast DDS will eventually run out of memory to store DDS data samples received from the network for that DataReader. See the code below for an example of borrowing and returning loaned sequences.

```
// Sequences are automatically initialized to be empty (maximum == 0)
FooSeq data_seq;
SampleInfoSeq info_seq;

// with empty sequences, a take() or read() will return loaned
// sequence elements
ReturnCode_t ret_code = data_reader->take(data_seq, info_seq,
                LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
                ANY_VIEW_STATE, ANY_INSTANCE_STATE);
```

(continues on next page)

(continued from previous page)

```
    // process the returned data

    // must return the loaned sequences when done processing
    data_reader->return_loan(data_seq, info_seq);
```

Processing returned data

After calling the *DataReader::read()* or *DataReader::take()* operations, accessing the data on the returned sequences is quite easy. The sequences API provides a **length()** operation returning the number of elements in the collections. The application code just needs to check this value and use the **[]** operator to access the corresponding elements. Elements on the DDS data sequence should only be accessed when the corresponding element on the SampleInfo sequence indicate that valid data is present. When using Data Sharing, it is also important to check that the sample is valid (i.e, not replaced, refer to *DataReader and DataWriter history coupling* for further information in this regard).

```
    // Sequences are automatically initialized to be empty (maximum == 0)
    FooSeq data_seq;
    SampleInfoSeq info_seq;

    // with empty sequences, a take() or read() will return loaned
    // sequence elements
    ReturnCode_t ret_code = data_reader->take(data_seq, info_seq,
                    LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
                    ANY_VIEW_STATE, ANY_INSTANCE_STATE);

    // process the returned data
    if (ret_code == ReturnCode_t::RETCODE_OK)
    {
        // Both info_seq.length() and data_seq.length() will have the number of samples␣
↪returned
        for (FooSeq::size_type n = 0; n < info_seq.length(); ++n)
        {
            // Only samples with valid data should be accessed
            if (info_seq[n].valid_data && data_reader->is_sample_valid(&data_seq[n], &
↪info_seq[n]))
            {
                // Do something with data_seq[n]
            }
        }

        // must return the loaned sequences when done processing
        data_reader->return_loan(data_seq, info_seq);
    }
```

Accessing data on callbacks

When the DataReader receives new data values from any matching DataWriter, it informs the application through two Listener callbacks:

- *on_data_available()*.
- *on_data_on_readers()*.

These callbacks can be used to retrieve the newly arrived data, as in the following example.

```
class CustomizedDataReaderListener : public DataReaderListener
{

public:

    CustomizedDataReaderListener()
        : DataReaderListener()
    {
    }

    virtual ~CustomizedDataReaderListener()
    {
    }

    void on_data_available(
            DataReader* reader) override
    {
        // Create a data and SampleInfo instance
        Foo data;
        SampleInfo info;

        // Keep taking data until there is nothing to take
        while (reader->take_next_sample(&data, &info) == ReturnCode_t::RETCODE_OK)
        {
            if (info.valid_data)
            {
                // Do something with the data
                std::cout << "Received new data value for topic "
                          << reader->get_topicdescription()->get_name()
                          << std::endl;
            }
            else
            {
                std::cout << "Remote writer for topic "
                          << reader->get_topicdescription()->get_name()
                          << " is dead" << std::endl;
            }
        }
    }

};
```

Note: If several new data changes are received at once, the callbacks may be triggered just once, instead of once per

change. The application must keep *reading* or *taking* until no new changes are available.

Accessing data with a waiting thread

Wait-sets and DataAvailable status condition

Instead of relying on the Listener to try and get new data values, the application can also dedicate a thread to wait until any new data is available on the DataReader. This can be done using a wait-set to wait for a change on the *DataAvailable* status.

```
// Create a DataReader
DataReader* data_reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
    // Error
    return;
}

// Prepare a wait-set to wait for data on the DataReader
WaitSet wait_set;
StatusCondition& condition = data_reader->get_statuscondition();
condition.set_enabled_statuses(StatusMask::data_available());
wait_set.attach_condition(condition);

// Create a data and SampleInfo instance
Foo data;
SampleInfo info;

//Define a timeout of 5 seconds
eprosima::fastrtps::Duration_t timeout (5, 0);

// Loop reading data as it arrives
// This will make the current thread to be dedicated exclusively to
// waiting and reading data until the remote DataWriter dies
while (true)
{
    ConditionSeq active_conditions;
    if (ReturnCode_t::RETCODE_OK == wait_set.wait(active_conditions, timeout))
    {
        while (ReturnCode_t::RETCODE_OK == data_reader->take_next_sample(&data, &info))
        {
            if (info.valid_data)
            {
                // Do something with the data
                std::cout << "Received new data value for topic "
                          << topic->get_name()
                          << std::endl;
            }
            else
            {
```

(continues on next page)

(continued from previous page)

```
                // If the remote writer is not alive, we exit the reading loop
                std::cout << "Remote writer for topic "
                          << topic->get_name()
                          << " is dead" << std::endl;
                break;
            }
        }
    }
    else
    {
        std::cout << "No data this time" << std::endl;
    }
}
```

DataReader non-blocking calls

The same could be achieved using the `DataReader::wait_for_unread_message()` member function, that blocks until a new data sample is available or the given timeout expires. If no new data was available after the timeout expired, it will return with value `false`. This function returning with value `true` means there is new data available on the *DataReader* ready for the application to retrieve.

```
// Create a DataReader
DataReader* data_reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
    // Error
    return;
}

// Create a data and SampleInfo instance
Foo data;
SampleInfo info;

//Define a timeout of 5 seconds
eprosima::fastrtps::Duration_t timeout (5, 0);

// Loop reading data as it arrives
// This will make the current thread to be dedicated exclusively to
// waiting and reading data until the remote DataWriter dies
while (true)
{
    if (data_reader->wait_for_unread_message(timeout))
    {
        if (ReturnCode_t::RETCODE_OK == data_reader->take_next_sample(&data, &info))
        {
            if (info.valid_data)
            {
                // Do something with the data
                std::cout << "Received new data value for topic "
                          << topic->get_name()
```

(continues on next page)

(continued from previous page)

```
                            << std::endl;
            }
            else
            {
                // If the remote writer is not alive, we exit the reading loop
                std::cout << "Remote writer for topic "
                            << topic->get_name()
                            << " is dead" << std::endl;
                break;
            }
        }
    }
    else
    {
        std::cout << "No data this time" << std::endl;
    }
}
```

6.17.5 Topic

A Topic conceptually fits between publications and subscriptions. Each publication channel must be unambiguously identified by the subscriptions in order to receive only the data flow they are interested in, and not data from other publications. A Topic serves this purpose, allowing publications and subscriptions that share the same Topic to match and start communicating. In that sense, the Topic acts as a description for a data flow.

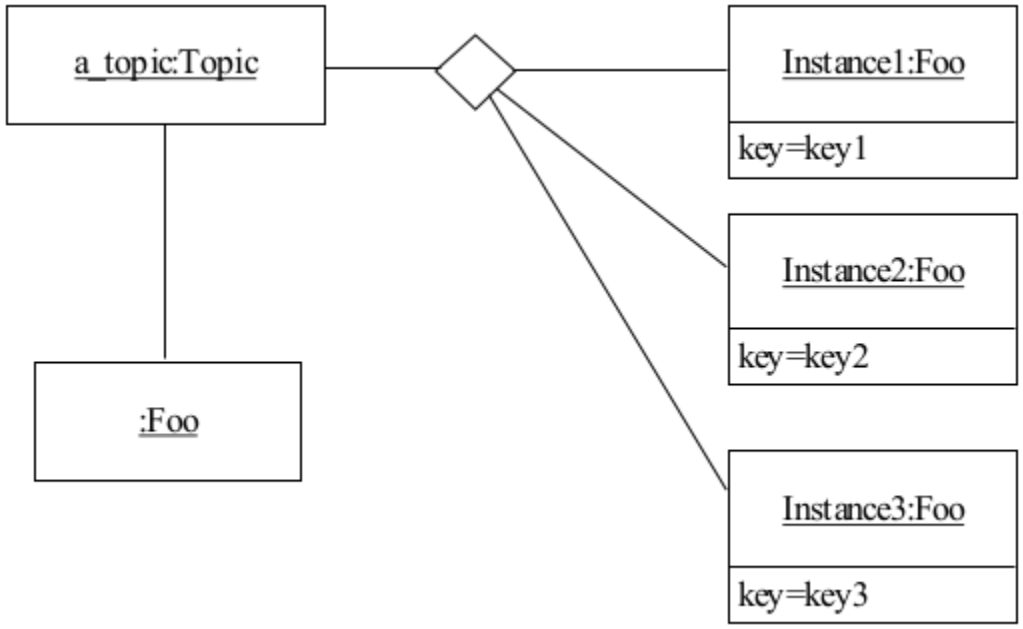
Publications are always linked to a single *Topic*, while subscriptions are linked to a broader concept of *TopicDescription*.

Fig. 8: Topic class diagram

Topics, keys and instances

By definition, a Topic is linked to a single data type, so each data sample related to a Topic could be understood as an update on the information described by the data type. However, it is possible to include a logical separation and have, within the same Topic, several instances referring to the same data type. Thus, the received data sample will be an update for a specific instance of that Topic. Therefore, a Topic identifies data of a single type, ranging from one single instance to a whole collection of instances of that given type, as shown in the figure below.

The different instances gathered under the same topic are distinguishable by means of one or more data fields that form the key to that data set. The key description has to be indicated to the middleware. The rule is simple: different data values with the same key value represent successive data samples for the same instance, while different data values with different keys represent different topic instances. If no key is provided, the data set associated with the Topic is restricted to a single instance. Please refer to *Data types with a key* for more information about how to set the key in *eProsima Fast DDS*.

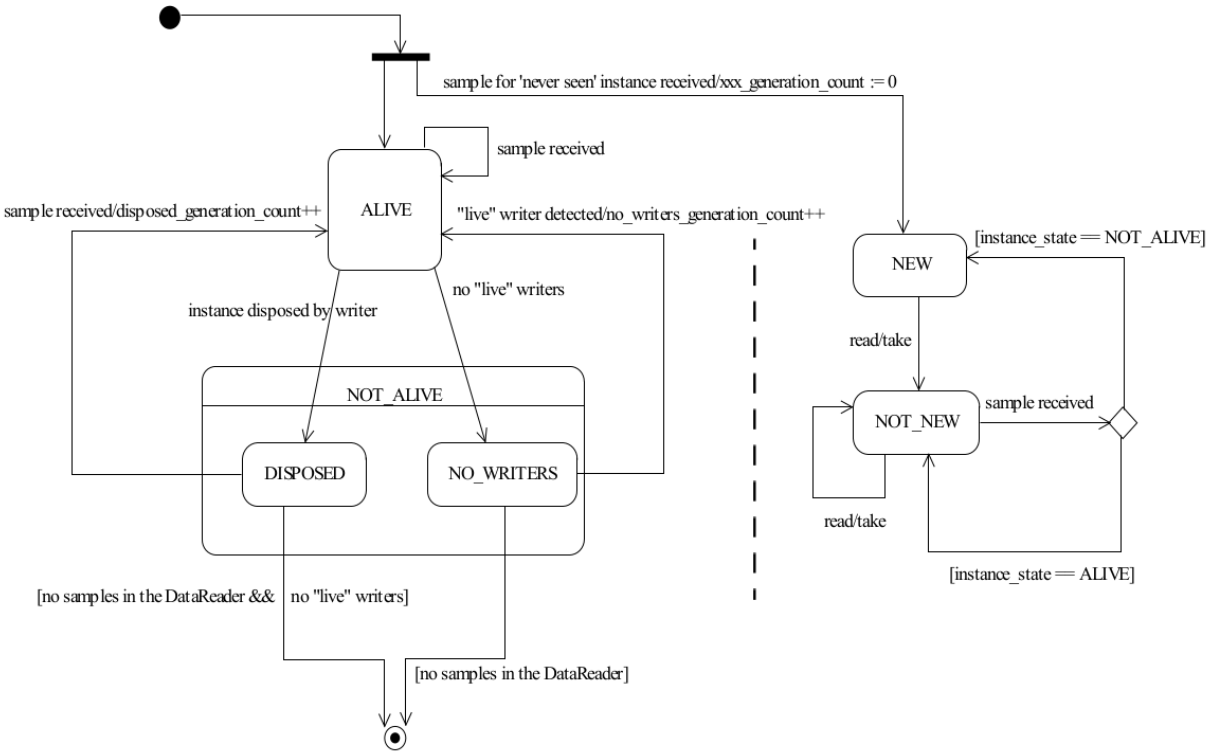


Instance advantages

The advantage of using instances instead of creating a new *DataWriter*, *DataReader*, and *Topic* is that the corresponding entity is already created and discovered. Consequently, there is less memory usage, and no new discovery (with the related metatraffic involved as explained in *Discovery*) is necessary. Another advantage is that several QoS are applied per topic instance; e.g. the *HistoryQosPolicy* is kept for each instance in the DataWriter. Thus, instances could be tuned to a wide range of applications.

Instance lifecycle

When reading or taking data from the *DataReader* (as explained in *Accessing received data*), a *SampleInfo* is also returned. This `SampleInfo` provides additional information about the instance lifecycle, specifically with the *view_state*, *instance_state*, *disposed_generation_count*, and *no_writers_generation_count*. The diagram below shows the statechart of `instance_state` and `view_state` for a single instance.



Practical applications

This section provides a couple of examples to help clarify the use of DDS instances.

Commercial flights tracking

Airspace and the air traffic going through it are typically managed by the air traffic controllers that are in charge of organizing the air traffic, preventing collisions, and providing information. In this scenario, each air traffic control center takes responsibility for a specific flight area and delivers the data to the airspace traffic management system, which unifies the flight information.

Any time an air traffic control center discovers a plane coming into its controlled flight zone, tracking information about that specific flight is notified to the airspace traffic management center. Such a flow of information could be implemented by means of DDS by creating a specific Topic where the information related to the flight location is published. In that case, the management center would be required to create, if not existing previously, the corresponding Topic and DataReader to have access to the flight information, with the corresponding memory consumption and discovery metatraffic required. On the other hand, a cleverer implementation could leverage topic instances to relay the information from the local air traffic control centers to the airspace traffic management center. The topic instances might be identified using the airline name and the flight number (i.e. *IBERIA 1234*) as Topic instance key. The sample data being relayed would be the location of each flight being tracked at any given time. The following IDL defines the data described model:

```
struct FlightPosition
{
    // Unique ID: airline name
    @key string<256> airline_name;

    // Unique ID: flight number
    @key short flight_number;
```


(continues on next page)

(continued from previous page)

```
    // Coordinates
    double latitude;
    double longitude;
    double altitude;
};
```

Once a new flight is discovered by a control center, the corresponding instance is registered into the system:

```
// Create data sample
FlightPosition first_flight_position;

// Specify the flight instance
first_flight_position.airline_name("IBERIA");
first_flight_position.flight_number(1234);

// Register instance
eprosima::fastrtps::rtps::InstanceHandle_t first_flight_handle =
        data_writer->register_instance(&first_flight_position);
```

register_instance() returns an *InstanceHandle_t* which can be used to efficiently call the next operations (i.e. *write()*, *dispose()*, or *unregister_instance()*) over the instance. The returned `InstanceHandle_t` contains the instance keyhash so it does not have to be recalculated again from the data sample. In case of following this approach, the application must take charge of mapping the instance handles to the corresponding instances.

```
// Update position value received from the plane
first_flight_position.latitude(39.08);
first_flight_position.longitude(-84.21);
first_flight_position.altitude(1500);

// Write sample to the instance
data_writer->write(&first_flight_position, first_flight_handle);
```

On the other hand, the user application could directly call the DataWriter instance operations with a `NIL` instance handle. In this case, the instance handle would be calculated every time an operation is done over the instance, which can be time consuming depending on the specific data type being used.

```
// New data sample
FlightPosition second_flight_position;

// New instance
second_flight_position.airline_name("RYANAIR");
second_flight_position.flight_number(4321);

// Update plane location
second_flight_position.latitude(40.02);
second_flight_position.longitude(-84.32);
second_flight_position.altitude(5000);

// Write sample directly without registering the instance
data_writer->write(&second_flight_position);
```

Warning: The correct management of the instance handles in the user application is paramount. Otherwise, a sample corresponding to a different instance could wrongly update the instance which handle the user has passed to the operation (if a non `NIL` instance is provided, the instance handle is not recalculated, trusting that the one passed by the user is the correct one). The following code updates the first instance of this example with the information coming from the second instance.

```
data_writer->write(&second_flight_position, first_flight_handle);
```

Once the plane leaves the controlled area, the air traffic control center may unregister the instance. Unregistering implies that the DataWriter for this specific center has no more information about the unregistered instance, and in this way the matched DataReaders in the management center are notified. The flight is still in the air but out of scope of this particular DataWriter. The instance is alive but no longer tracked by this center.

```
data_writer->unregister_instance(&first_flight_position, first_flight_handle);
data_writer->unregister_instance(&second_flight_position, HANDLE_NIL);
```

Finally, when the flight lands, the instance may be disposed. This means, in this specific example, that as far as the DataWriter knows, the instance no longer exists and should be considered not alive. With this operation, the DataWriter conveys this information to the matched DataReaders.

```
data_writer->dispose(&first_flight_position, first_flight_handle);
data_writer->dispose(&second_flight_position, HANDLE_NIL);
```

From the management center point of view, the samples are read using the same DataReader subscribed to the Topic where the instances are being published. However, *valid_data* must be checked to ensure that the sample received contains a data sample. Otherwise, a change of the instance state is being notified. *Instance lifecycle* contains a diagram showing the instance statechart.

```
if (ReturnCode_t::RETCODE_OK == data_reader->take_next_sample(&data, &info))
{
    if (info.valid_data)
    {
        // Data sample has been received
    }
    else if (info.instance_state == NOT_ALIVE_DISPOSED_INSTANCE_STATE)
    {
        // A remote DataWriter has disposed the instance
    }
    else if (info.instance_state == NOT_ALIVE_NO_WRITERS_INSTANCE_STATE)
    {
        // None of the matched DataWriters are writing in the instance.
        // The instance can be safely disposed.
    }
}
```

Relational databases

Consider now that the air traffic management center wants to keep a database with the flights being tracked. Using DDS instances, maintaining a relational database is almost direct. The instance key (unique identifier of the instance) is analogous to the primary key of the database. Thus, the airspace traffic management center can keep the latest update for each instance in a table like the one below:

Instance handle [PK]	Data
1	Position1
2	Position2
3	Position3
4	Position4
5	Position5

In this case, every time a new sample is received, the corresponding instance entry in the database will be updated with the latest known location. Disposing the instance may translate in erasing the corresponding data from the database. In this scenario, registering and unregistering the instances does not reflect in the database, although if the *instance_state* and *view_state* are also persisted, then the instance lifecycle could be tracked as well. A DataWriter communicating that it is going to be publishing data about a specific instance is of no interest to the database until a new data is received and then an insert is directly done with the new discovered instance.

Historical data can also be stored in the relational database, even though depending on the use case, a time series database might be considered to improve efficiency. In the scenario being considered, the sample timestamp could be used, besides the instance handle, as primary key to be able to access the historical tracking data of an specific flight.

Instance handle [PK]	Source Timestamp [PK]	Data
1	1	Position1
2	1	Position2
1	2	Position3
1	3	Position4
2	2	Position5

In this case, looking for a specific instance handle would return the flight tracking information:

Instance handle [Fixed]	Source Timestamp	Data
1	1	Position1
1	2	Position3
1	3	Position4

Whereas looking for a specific timestamp would allow to have a picture of the different flight locations at a specific time:

Instance handle	Source Timestamp [Fixed]	Data
1	2	Position3
2	2	Position5

TopicDescription

TopicDescription is an abstract class that serves as the base for all classes describing a data flow. Applications will not create instances of *TopicDescription* directly, they must create instances of one of its specializations instead. At the moment, the only specializations implemented are *Topic*, and *ContentFilteredTopic*.

Topic

A *Topic* is a specialization of the broader concept of *TopicDescription*. A Topic represents a single data flow between *Publisher* and *Subscriber*, providing:

- The name to identify the data flow.
- The data type that is transmitted on that flow.
- The QoS values related to the data itself.

The behavior of the Topic can be modified with the QoS values specified on *TopicQos*. The QoS values can be set at the creation of the Topic, or modified later with the *Topic::set_qos()* member function.

Like other Entities, Topic accepts a Listener that will be notified of status changes on the Topic.

Please refer to *Creating a Topic* for more information about how to create a *Topic*.

TopicQos

TopicQos controls the behavior of the Topic. Internally it contains the following *QosPolicy* objects:

QosPolicy class	Accessor	Mutable
TopicDataQosPolicy	*topic_data()*	Yes
DurabilityQosPolicy	*durability()*	Yes
DurabilityServiceQosPolicy	*durability_service()*	Yes
DeadlineQosPolicy	*deadline()*	Yes
LatencyBudgetQosPolicy	*latency_budget()*	Yes
LivelinessQosPolicy	*liveliness()*	Yes
ReliabilityQosPolicy	*reliability()*	Yes
DestinationOrderQosPolicy	*destination_order()*	Yes
HistoryQosPolicy	*history()*	Yes
ResourceLimitsQosPolicy	*resource_limits()*	Yes
TransportPriorityQosPolicy	*transport_priority()*	Yes
LifespanQosPolicy	*lifespan()*	Yes
OwnershipQosPolicy	*ownership()*	Yes
DataRepresentationQosPolicy	*representation()*	Yes

Refer to the detailed description of each QosPolicy-api class for more information about their usage and default values.

The QoS value of a previously created Topic can be modified using the *Topic::set_qos()* member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Create a Topic with default TopicQos
Topic* topic =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
TopicQos qos = topic->get_qos();

// Modify QoS attributes
// (...)

// Assign the new Qos to the object
topic->set_qos(qos);
```

Default TopicQos

The default *TopicQos* refers to the value returned by the *get_default_topic_qos()* member function on the *DomainParticipant* instance. The special value `TOPIC_QOS_DEFAULT` can be used as QoS argument on *create_topic()* or *Topic::set_qos()* member functions to indicate that the current default TopicQos should be used.

When the system starts, the default TopicQos is equivalent to the default constructed value *TopicQos()*. The default TopicQos can be modified at any time using the *get_default_topic_qos()* member function on the DomainParticipant instance. Modifying the default TopicQos will not affect already existing Topic instances.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
TopicQos qos_type1 = participant->get_default_topic_qos();

// Modify QoS attributes
// (...)

// Set as the new default TopicQos
if (participant->set_default_topic_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Create a Topic with the new default TopicQos.
Topic* topic_with_qos_type1 =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic_with_qos_type1)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
TopicQos qos_type2;

// Modify QoS attributes
// (...)

// Set as the new default TopicQos
if (participant->set_default_topic_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Topic with the new default TopicQos.
Topic* topic_with_qos_type2 =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default TopicQos to the original default constructed values
if (participant->set_default_topic_qos(TOPIC_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (participant->set_default_topic_qos(TopicQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

get_default_topic_qos() member function also accepts the value TOPIC_QOS_DEFAULT as input argument. This will reset the current default TopicQos to default constructed value *TopicQos()*.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a custom TopicQos
TopicQos custom_qos;

// Modify QoS attributes
// (...)

// Create a topic with a custom TopicQos
Topic* topic = participant->create_topic("TopicName", "DataTypeName", custom_qos);
if (nullptr == topic)
{
    // Error
    return;
}

// Set the QoS on the topic to the default
if (topic->set_qos(TOPIC_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (topic->set_qos(participant->get_default_topic_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `TOPIC_QOS_DEFAULT` has different meaning depending on where it is used:

- On *create_topic()* and *Topic::set_qos()* it refers to the default TopicQos as returned by *get_default_topic_qos()*.
- On *get_default_topic_qos()* it refers to the default constructed *TopicQos()*.

ContentFilteredTopic

A *ContentFilteredTopic* is a specialization of the broader concept of *TopicDescription*. A ContentFilteredTopic is a Topic with filtering properties. It makes it possible to subscribe to a Topic while at the same time specify interest on a subset of the Topic's data.

Important: Note that a ContentFilteredTopic can only be used to create a DataReader, not a DataWriter.

A ContentFilteredTopic provides a relationship between a *Topic*, called the related topic, and some user-defined filtering properties:

- A **filter expression**, which establishes a logical expression on the content of the related topic. It is similar to the WHERE clause in a SQL statement.
- A list of **expression parameters**, which give values to the parameters present in the filter expression. There must be one parameter string for each parameter in the filter expression.

Note that a ContentFilteredTopic is *not* an Entity, and thus it has neither QoS nor listener. A DataReader created with a ContentFilteredTopic will use the QoS from the related topic. Multiple DataReaders can be created for the same ContentFilteredTopic, and changing the filter properties of a ContentFilteredTopic will affect all DataReaders using it.

Please refer to *Filtering data on a Topic* and *Where is filtering applied: writer vs reader side* for more information about how to use *ContentFilteredTopic*.

TopicListener

TopicListener is an abstract class defining the callbacks that will be triggered in response to state changes on the *Topic*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

TopicListener has the following callback:

- *on_inconsistent_topic()*: A remote Topic is discovered with the same name but different characteristics as another locally created Topic.

Warning: Currently *on_inconsistent_topic()* is not implemented (it will never be called), and will be implemented on a future release of *Fast DDS*.

```
class CustomTopicListener : public TopicListener
{

public:

    CustomTopicListener()
        : TopicListener()
    {
    }

    virtual ~CustomTopicListener()
    {
    }
```

(continues on next page)

(continued from previous page)

```
    void on_inconsistent_topic(
            Topic* topic,
            InconsistentTopicStatus status) override
    {
        static_cast<void>(topic);
        static_cast<void>(status);
        std::cout << "Inconsistent topic received discovered" << std::endl;
    }

};
```

Definition of data types

The definition of the data type exchanged in a *Topic* is divided in two classes: the *TypeSupport* and the *TopicDataType*.

TopicDataType describes the data type exchanged between a publication and a subscription, i.e., the data corresponding to a Topic. The user has to create a specialized class for each specific type that will be used by the application.

Any specialization of TopicDataType must be registered in the *DomainParticipant* before it can be used to create Topic objects. A TypeSupport object encapsulates an instance of TopicDataType, providing the functions needed to register the type and interact with the publication and subscription. To register the data type, create a new TypeSupport with a TopicDataType instance and use the *register_type()* member function on the TypeSupport. Then the Topic can be created with the registered type name.

Note: Registering two different data types on the same DomainParticipant with identical names is not allowed and will issue an error. However, it is allowed to register the same data type within the same DomainParticipant, with the same or different names. If the same data type is registered twice on the same DomainParticipant with the same name, the second registering will have no effect, but will not issue any error.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Register the data type in the DomainParticipant.
// If nullptr is used as name argument, the one returned by the type itself is used
TypeSupport custom_type_support(new CustomDataType());
custom_type_support.register_type(participant, nullptr);

// The previous instruction is equivalent to the following one
// Even if we are registering the same data type with the same name twice, no error will␣
↪be issued
custom_type_support.register_type(participant, custom_type_support.get_type_name());
```

(continues on next page)

(continued from previous page)

```
// Create a Topic with the registered type.
Topic* topic =
        participant->create_topic("topic_name", custom_type_support.get_type_name(),␣
↪TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create an alias for the same data type using a different name.
custom_type_support.register_type(participant, "data_type_name");

// We can now use the aliased name to If no name is given, it uses the name returned by␣
↪the type itself
Topic* another_topic =
        participant->create_topic("other_topic_name", "data_type_name", TOPIC_QOS_
↪DEFAULT);
if (nullptr == another_topic)
{
    // Error
    return;
}
```

Dynamic data types

Instead of directly writing the specialized *TopicDataType* class, it is possible to dynamically define data types following the OMG Extensible and Dynamic Topic Types for DDS interface. Data types can also be described on an XML file that is dynamically loaded.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Load the XML file with the type description
eprosima::fastrtps::xmlparser::XMLProfileManager::loadXMLFile("example_type.xml");

// Retrieve the an instance of the desired type and register it
eprosima::fastrtps::types::DynamicType_ptr dyn_type =
        eprosima::fastrtps::xmlparser::XMLProfileManager::getDynamicTypeByName(
↪"DynamicType")->build();
TypeSupport dyn_type_support(new eprosima::fastrtps::types::DynamicPubSubType(dyn_type));
dyn_type_support.register_type(participant, nullptr);

// Create a Topic with the registered type.
```

(continues on next page)

(continued from previous page)

```
Topic* topic =
        participant->create_topic("topic_name", dyn_type_support.get_type_name(), TOPIC_
↪QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}
```

A complete description of the dynamic definition of types can be found on the *Dynamic Topic Types* section.

Data types with a key

Data types that define a set of fields to form a unique key can distinguish different data sets within the same data type.

To define a keyed Topic, the *getKey()* member function on the *TopicDataType* has to be overridden to return the appropriate key value according to the data fields. Additionally, the *m_isGetKeyDefined* data member needs to be set to `true` to let the entities know that this is a keyed Topic and that *getKey()* should be used. Types that do not define a key will have *m_isGetKeyDefined* set to false.

There are three ways to implement keys on the TopicDataType:

- Adding a `@Key` annotation to the members that form the key in the IDL file when using *Fast DDS-Gen*.
- Adding the attribute `Key` to the member and its parents when using *Dynamic Topic Types*.
- Manually implementing the *getKey()* member function on the TopicDataType and setting the *m_isGetKeyDefined* data member value to `true`.

Data types with key are used to define data sub flows on a single Topic. Data values with the same key on the same Topic represent data from the same sub-flow, while data values with different keys on the same Topic represent data from different sub-flows. The middleware keeps these sub-flows separated, but all will be restricted to the same QoS values of the Topic. If no key is provided, the data set associated with the Topic is restricted to a single flow.

Creating a Topic

A *Topic* always belongs to a *DomainParticipant*. Creation of a Topic is done with the *create_topic()* member function on the *DomainParticipant* instance, that acts as a factory for the *Topic*.

Mandatory arguments are:

- A string with the name that identifies the Topic.
- The name of the registered *data type* that will be transmitted.
- The *TopicQos* describing the behavior of the Topic. If the provided value is `TOPIC_QOS_DEFAULT`, the value of the *Default TopicQos* is used.

Optional arguments are:

- A Listener derived from *TopicListener*, implementing the callbacks that will be triggered in response to events and state changes on the Topic. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the TopicListener. By default all events are enabled.

create_topic() will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic with default TopicQos and no Listener
// The symbol TOPIC_QOS_DEFAULT is used to denote the default QoS.
Topic* topic_with_default_qos =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic_with_default_qos)
{
    // Error
    return;
}

// A custom TopicQos can be provided to the creation method
TopicQos custom_qos;

// Modify QoS attributes
// (...)

Topic* topic_with_custom_qos =
        participant->create_topic("TopicName", "DataTypeName", custom_qos);
if (nullptr == topic_with_custom_qos)
{
    // Error
    return;
}

// Create a Topic with default QoS and a custom Listener.
// CustomTopicListener inherits from TopicListener.
// The symbol TOPIC_QOS_DEFAULT is used to denote the default QoS.
CustomTopicListener custom_listener;
Topic* topic_with_default_qos_and_custom_listener =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT, &
↪custom_listener);
if (nullptr == topic_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a Topic

Instead of using a TopicQos, the name of a profile can be used to create a Topic with the *create_topic_with_profile()* member function on the DomainParticipant instance.

Mandatory arguments are:

- A string with the name that identifies the Topic.
- The name of the registered *data type* that will be transmitted.
- The name of the profile to be applied to the Topic.

Optional arguments are:

- A Listener derived from TopicListener, implementing the callbacks that will be triggered in response to events and state changes on the Topic. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the TopicListener. By default all events are enabled.

`create_topic_with_profile()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic using a profile and no Listener
Topic* topic_with_profile =
        participant->create_topic_with_profile("TopicName", "DataTypeName", "topic_
↪profile");
if (nullptr == topic_with_profile)
{
    // Error
    return;
}

// Create a Topic using a profile and a custom Listener.
// CustomTopicListener inherits from TopicListener.
CustomTopicListener custom_listener;
Topic* topic_with_profile_and_custom_listener =
        participant->create_topic_with_profile("TopicName", "DataTypeName", "topic_
↪profile", &custom_listener);
```

(continues on next page)

(continued from previous page)

```
if (nullptr == topic_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a Topic

A Topic can be deleted with the *delete_topic()* member function on the DomainParticipant instance where the Topic was created.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic
Topic* topic =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Use the Topic to communicate
// (...)

// Delete the Topic
if (participant->delete_topic(topic) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Filtering data on a Topic

Creating a ContentFilteredTopic

A *ContentFilteredTopic* always belongs to a *DomainParticipant*. Creation of a ContentFilteredTopic is done with the *create_contentfilteredtopic()* member function on the *DomainParticipant* instance, that acts as a factory for the *ContentFilteredTopic*.

Mandatory arguments are:

- A string with the name that identifies the ContentFilteredTopic.

- The related *Topic* being filtered.
- A string with the filter expression indicating the conditions for a sample to be returned.
- A list of strings with the value of the parameters present on the filter expression.

Note: The number of parameter values cannot exceed the maximum set by the *expression_parameters* QoS configuration. The default (and absolute) maximum allowed as set by the OMG DDS Standard is 100.

Optional arguments are:

- A string with the name of the filter class to use for the filter creation. This allows the user to create filters different from the standard SQL like one (please refer to *Using custom filters*). Defaults to *FASTDDS_SQLFILTER_NAME* (`DDSSQL`).

Important: Setting an empty string as filter expression results in the disabling of the filtering. This can be used to enable/disable the DataReader filtering capabilities at any given time by simply *updating the filter expression*.

---

create_contentfilteredtopic() will return a null pointer if there was an error during the operation, e.g. if the related Topic belongs to a different DomainParticipant, a Topic with the same name already exists, syntax errors on the filter expression, or missing parameter values. It is advisable to check that the returned value is a valid pointer.

Note: Different filter classes may impose different requirements on the related Topic, the expression, or the parameters. The default filter class, in particular, requires that a TypeObject for the related Topic's type has been registered. When using *fastddsgen* to generate your type support code, remember to include the `-typeobject` option so the TypeObject registration code is generated.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create the Topic.
/* IDL
 *
 * struct HelloWorld
 * {
 *     long index;
 *     string message;
 * }
 *
 */
Topic* topic =
        participant->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Create a ContentFilteredTopic using an expression with no parameters
std::string expression = "message like 'Hello*'";
std::vector<std::string> parameters;
ContentFilteredTopic* filter_topic =
        participant->create_contentfilteredtopic("HelloWorldFilteredTopic1", topic,␣
↪expression, parameters);
if (nullptr == filter_topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic using an expression with parameters
expression = "message like %0 or index > %1";
parameters.push_back("'*world*'");
parameters.push_back("20");
ContentFilteredTopic* filter_topic_with_parameters =
        participant->create_contentfilteredtopic("HelloWorldFilteredTopic2", topic,␣
↪expression, parameters);
if (nullptr == filter_topic_with_parameters)
{
    // Error
    return;
}

// The ContentFilteredTopic instances can then be used to create DataReader objects.
Subscriber* subscriber =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}

DataReader* reader_on_filter = subscriber->create_datareader(filter_topic, DATAREADER_
↪QOS_DEFAULT);
if (nullptr == reader_on_filter)
{
    // Error
    return;
}

DataReader* reader_on_filter_with_parameters =
        subscriber->create_datareader(filter_topic_with_parameters, DATAREADER_QOS_
↪DEFAULT);
if (nullptr == reader_on_filter_with_parameters)
{
    // Error
```

(continues on next page)

(continued from previous page)

```
    return;
}
```

Updating the filter expression and parameters

A ContentFilteredTopic provides several member functions for the management of the filter expression and the expression parameters:

- The filter expression can be retrieved with the *get_filter_expression()* member function.
- The expression parameters can be retrieved with the *get_expression_parameters()* member function.
- The expression parameters can be modified using the *set_expression_parameters()* member function. The same constraints as when creating a ContentFilteredTopic apply.
- The filter expression can be modified along with the expression parameters using the *set_filter_expression()* member function.

```
// This lambda prints all the information of a ContentFilteredTopic
auto print_filter_info = [](
    const ContentFilteredTopic* filter_topic)
        {
            std::cout << "ContentFilteredTopic info for '" << filter_topic->get_name() <
↪< "':" << std::endl;
            std::cout << "  - Related Topic: " << filter_topic->get_related_topic()->get_
↪name() << std::endl;
            std::cout << "  - Expression:    " << filter_topic->get_filter_expression() <
↪< std::endl;
            std::cout << "  - Parameters:" << std::endl;

            std::vector<std::string> parameters;
            filter_topic->get_expression_parameters(parameters);
            size_t i = 0;
            for (const std::string& parameter : parameters)
            {
                std::cout << "    " << i++ << ": " << parameter << std::endl;
            }
        };

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic
/* IDL
 *
 * struct HelloWorld
```

(continues on next page)

(continued from previous page)

```
 * {
 *     long index;
 *     string message;
 * }
 *
 */
Topic* topic =
        participant->create_topic("HelloWorldTopic", "HelloWorldTopic", TOPIC_QOS_
↪DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic
ContentFilteredTopic* filter_topic =
        participant->create_contentfilteredtopic("HelloWorldFilteredTopic", topic,
↪"index > 10", {});
if (nullptr == filter_topic)
{
    // Error
    return;
}

// Print the information
print_filter_info(filter_topic);

// Use the ContentFilteredTopic on DataReader objects.
// (...)

// Update the expression
if (ReturnCode_t::RETCODE_OK !=
        filter_topic->set_filter_expression("message like %0 or index > %1", {"'Hello*'",
↪ "15"}))
{
    // Error
    return;
}

// Print the updated information
print_filter_info(filter_topic);

// Update the parameters
if (ReturnCode_t::RETCODE_OK !=
        filter_topic->set_expression_parameters({"'*world*'", "222"}))
{
    // Error
    return;
}

// Print the updated information
```

(continues on next page)

(continued from previous page)

```
print_filter_info(filter_topic);
```

Deleting a ContentFilteredTopic

A ContentFilteredTopic can be deleted with the *delete_contentfilteredtopic()* member function on the DomainParticipant instance where the ContentFilteredTopic was created.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic
/* IDL
 *
 * struct HelloWorld
 * {
 *     long index;
 *     string message;
 * }
 *
 */
Topic* topic =
        participant->create_topic("HelloWorldTopic", "HelloWorldTopic", TOPIC_QOS_
↪DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic
ContentFilteredTopic* filter_topic =
        participant->create_contentfilteredtopic("HelloWorldFilteredTopic", topic,
↪"index > 10", {});
if (nullptr == filter_topic)
{
    // Error
    return;
}

// Use the ContentFilteredTopic on DataReader objects.
// (...)

// Delete the ContentFilteredTopic
if (ReturnCode_t::RETCODE_OK != participant->delete_contentfilteredtopic(filter_topic))
```

(continues on next page)

(continued from previous page)

```
{
    // Error
    return;
}
```

The default SQL-like filter

Filter expressions used by *ContentFilteredTopic* API may use a subset of SQL syntax, extended with the possibility to use program variables in the SQL expression. This section shows this default SQL-like syntax and how to use it.

- *Grammar*
- *Like condition*
- *Match condition*
- *Type comparisons*
- *Example*

Grammar

The allowed SQL expressions are defined with the BNF-grammar below.

The following conventions are made:

- "Terminals" are quoted.
- TOKENS are typeset in code block with black font color.

```
Expression           ::=   FilterExpression
FilterExpression     ::=   Condition
Condition            ::=   Predicate |
                           Condition "AND" Condition |
                           Condition "OR" Condition |
                           "NOT" Condition |
                           "(" Condition ")"
Predicate            ::=   ComparisonPredicate |
                           BetweenPredicate
ComparisonPredicate  ::=   FIELDNAME RelOp Parameter |
                           Parameter RelOp FIELDNAME |
                           FIELDNAME RelOp FIELDNAME
BetweenPredicate     ::=   FIELDNAME "BETWEEN" Range |
                           FIELDNAME "NOT BETWEEN" Range
RelOp                ::=   "=" | ">" | ">=" | "<" | "<=" |
                           "<>" | "!=" | like | match
Range                ::=   Parameter "AND" Parameter
Parameter            ::=   BOOLEANVALUE |
                           INTEGERVALUE |
                           CHARVALUE |
                           FLOATVALUE |
                           STRINGVALUE |
                           ENUMERATEDVALUE |
```

```
PARAMETER
```

"Terminals" and TOKENS are case sensitive but both uppercase and lowercase are supported.

The syntax and meaning of the tokens used in the SQL grammar is described as follows:

- **FIELDNAME**: is a reference to a field in the data-structure. The dot `.` is used to navigate through nested structures. The number of dots that may be used in a FIELDNAME is unlimited. The FIELDNAME can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the corresponding structure.

```
FIELDNAME      ::=   FieldNamePart ( "." FieldNamePart )*
FieldNamePart  ::=   Identifier ( "[" Integer "]" )?
```

An example of FIELDNAMEs:

Filter expression

```
"points[0] = 0 AND color.red < 100"
```

Associated IDL

```
struct Color
{
    octet red;
    octet green;
    octet blue;
};

struct Shape
{
    long points[4];
    Color color;
};
```

- **BOOLEANVALUE**: Can either be *true* of *false*, case sensitive.

```
BOOLEANVALUE  ::=   ["TRUE", "true", "FALSE", "false"]
```

- **INTEGERVALUE**: Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. A hexadecimal number is preceded by `0x` and must be a valid hexadecimal expression.

```
INTEGERVALUE  ::=   (["+","-"])? Integer
Integer       ::=   (["0"-"9"])+ | ["0x","0X"](["0"-"9", "A"-"F", "a"-"f"])+
```

An example of INTEGERVALUE:

```
value = -10
```

- **CHARVALUE**: A single character enclosed between single quotes.

```
CHARVALUE  ::=   "'" Character "'"
Character  ::=   ~["\n"]
```

An example of CHARVALUE:

```
value = 'c'
```

- **FLOATVALUE**: Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point (.). A power-of-ten expression may be postfixed, which has the syntax e:sup:*n*, where n is a number, optionally preceded by a plus or minus sign.

```
FLOATVALUE  ::=   (["+"], "-"])? (Integer Exponent | Integer Fractional | Integer Fractional Exponent)
Fractional  ::=   "." Integer
Exponent    ::=   ["e","E"] (["+"], "-"])? Integer
```

An example of FLOATVALUE:

```
value = 10.1e-10
```

- **STRINGVALUE**: Any series of characters encapsulated in single quotes, except a new-line character or a right quote. A string starts with a left or right quote, but ends with a right quote.

```
STRINGVALUE  ::=   ["'"] ~["'", "\r", "\n"] ["'"]
```

An example of STRINGVALUE:

```
value = 'This is a string'
```

- **ENUMERATEDVALUE**: An enumerated value is a reference to a value declared within an enumeration. Enumerated values consist of the name of the enumeration label enclosed in single quotes. The name used for the enumeration label must correspond to the label names specified in the IDL definition of the enumeration.

```
ENUMERATEDVALUE  ::=   ["'"] ~["'", "\r", "\n"] ["'"]
```

An example of ENUMERATEDVALUE:

Filter expression

```
value = 'ENUM_VALUE_1'
```

Associated IDL

```
enum MyEnum
{
    ENUM_VALUE_1,
    ENUM_VALUE_2,
    ENUM_VALUE_3
};

struct Enumerators
{
    MyEnum value;
```

(continues on next page)

(continued from previous page)

```
};
```

- **PARAMETER**: A parameter is of the form `%n`, where `n` represents a natural number (zero included) smaller than 100. It refers to the `n + 1 th` argument in the given context.

```
PARAMETER  ::=   ["%"] ["0"-"9"] (["0"-"9"])?
```

An example of PARAMETER:

```
value = %1
```

Like condition

The like operator is similar as the one defined by SQL. This operator can only be used with strings. There are two wildcards that could be used in conjunction with this operator

- The percent sign `%` (or its alias *) represents zero, one, or multiple characters.
- The underscore sign _ (or its alias ?) represents one single character.

All wildcards can also be used in combinations.

An example of `like` operator

Filter expression

```
"str like '%bird%'"
```

Associated IDL

```
struct Like
{
    string str;
};
```

where string `There are birds flying` will return `true`.

Match condition

The `match` operator performs a full-text search using a regular expression. This operator can only be used with strings. It uses the Basic Regular Expression (BRE) defined by POSIX.

An example of `match` operator

Filter expression

```
"str match '^The'"
```

Associated IDL

```
struct Like
{
```

(continues on next page)

(continued from previous page)

```
    string str;
};
```

where string `There are birds flying` will return `true`.

Type comparisons

For the supported operators in the grammar, next table shows the type compatibility.

Operator1 | Operator2	BOOLEAN	INTEGER	FLOAT	CHAR	STRING	ENUM
BOOLEAN						
INTEGER						
FLOAT						
CHAR						
STRING						
ENUM						*

(*) Only for the same enumerated type.

Example

Assuming Topic `Shape` has next IDL definition.

```
struct Shape
{
    long x,
    long y,
    long z,
    long width,
    long height
};
```

An example of filter expression would be:

```
"x < 23 AND y > 50 AND width BETWEEN %0 AND %1"
```

A *ContentFilteredTopic* may be created using this filter expression as explained in section *Creating a ContentFilteredTopic*.

```
ContentFilteredTopic* sql_filter_topic =
        participant->create_contentfilteredtopic("Shape", topic,
                "x < 23 AND y > 50 AND width BETWEEN %0 AND %1",
                {"10", "20"});
```

In this example parameters are used. Internally the *ContentFilteredTopic* will be created with the filter expression below, after setting the provided parameters.

```
"x < 23 AND y > 50 AND width BETWEEN 10 AND 20"
```

Using custom filters

Fast DDS API supports the creation and later registration of user's custom filters to be used in the creation of a *ContentFilteredTopic*. Required steps for using a Custom Filter are:

- *Creating the Custom Filter*
- *Creating the Factory for the Custom Filter*
- *Registering the Factory*
- *Creating a ContentFilteredTopic using the Custom Filter*

Creating the Custom Filter

A custom filter must be implemented by a class which inherits from *IContentFilter*. Only one function must be implemented, overriding *evaluate()*. Each time a sample is received by a *DataReader*, this function is called with next arguments.

- `payload` - The serialized payload of the sample which the custom filter has to evaluate.
- `sample_info` - The extra information which accompanies the sample.
- `reader_guid` - The GUID of the reader for which the filter is being evaluated.

The function returns a boolean where `true` implies the sample is accepted and `false` rejects the sample.

Next snippet code shows an example of Custom Filter which deserialize the `index` field from a serialized sample and rejects samples where `index` > `low_mark_` and `index` < `high_mark_`.

```
class MyCustomFilter : public IContentFilter
{
public:

    MyCustomFilter(
            int low_mark,
            int high_mark)
        : low_mark_(low_mark)
        , high_mark_(high_mark)
    {
    }

    bool evaluate(
            const SerializedPayload& payload,
            const FilterSampleInfo& sample_info,
            const GUID_t& reader_guid) const override
    {
        // Deserialize the `index` field from the serialized sample.
        /* IDL
         *
         * struct HelloWorld
         * {
         *     long index;
         *     string message;
         * }
         */
        eprosima::fastcdr::FastBuffer fastbuffer(reinterpret_cast<char*>(payload.data),␣
↪payload.length);
```

(continues on next page)

(continued from previous page)

```
        eprosima::fastcdr::Cdr deser(fastbuffer);
        // Deserialize encapsulation.
        deser.read_encapsulation();
        int index = 0;

        // Deserialize `index` field.
        try
        {
            deser >> index;
        }
        catch (eprosima::fastcdr::exception::NotEnoughMemoryException& /*exception*/)
        {
            return false;
        }

        // Custom filter: reject samples where index > low_mark_ and index < high_mark_.
        if (index > low_mark_ && index < high_mark_)
        {
            return false;
        }

        return true;
    }

private:

    int low_mark_ = 0;
    int high_mark_ = 0;

};
```

Creating the Factory for the Custom Filter

Fast DDS creates filters through a factory. Therefore a factory which provides instantiating of a Custom Filter must be implemented.

A Custom Filter's factory has to inherit from *IContentFilterFactory*. This interface requires two functions to be implemented.

Each time a Custom Filter has to be created or updated, *create_contentfilteredtopic()* calls internally *create_content_filter()* with these arguments:

- `filter_class_name` - Filter class name for which the factory is being called. It allows using the same factory for different filter classes.
- `type_name` - Type name of the topic being filtered.
- `data_type` - Type support object of the topic being filtered.
- `filter_expression` - Custom filter expression.
- `filter_parameters` - Values to set for the filter parameters (where custom filter expression has its pattern to substitute them).

- `filter_instance` - When a filter is being created, it will be `nullptr` on input, and will have the pointer to the created filter instance on output. When a filter is being updated, it will have a previously returned pointer on input.

This function should return the result of the operation.

When a Custom Filter should be removed, *delete_contentfilteredtopic()* calls internally *delete_content_filter()*. The factory must remove the provided Custom Filter's instance.

Next snippet code shows an example of Custom Filter's factory which manages instances of the Custom Filter implemented in the previous section.

```
class MyCustomFilterFactory : public IContentFilterFactory
{
public:

    ReturnCode_t create_content_filter(
            const char* filter_class_name, // My custom filter class name is 'MY_CUSTOM_
↪FILTER'.
            const char* type_name, // This custom filter only supports one type:
↪'HelloWorld'.
            const TopicDataType* /*data_type*/, // Not used in this implementation.
            const char* filter_expression, // This Custom Filter doesn't implement a␣
↪filter expression.
            const ParameterSeq& filter_parameters, // Always need two parameters to be␣
↪set: low_mark and high_mark.
            IContentFilter*& filter_instance) override
    {
        // Check the ContentFilteredTopic should be created by my factory.
        if (0 != strcmp(filter_class_name, "MY_CUSTOM_FILTER"))
        {
            return ReturnCode_t::RETCODE_BAD_PARAMETER;
        }

        // Check the ContentFilteredTopic is created for the unique type this Custom␣
↪Filter supports.
        if (0 != strcmp(type_name, "HelloWorld"))
        {
            return ReturnCode_t::RETCODE_BAD_PARAMETER;
        }

        // Check that the two mandatory filter parameters are set.
        if (2 != filter_parameters.length())
        {
            return ReturnCode_t::RETCODE_BAD_PARAMETER;
        }

        // If there is an update, delete previous instance.
        if (nullptr != filter_instance)
        {
            delete(dynamic_cast<MyCustomFilter*>(filter_instance));
        }

        // Instantiation of the Custom Filter.
        filter_instance = new MyCustomFilter(std::stoi(filter_parameters[0]),␣
↪std::stoi(filter_parameters[1]));
```

(continues on next page)

(continued from previous page)

```
        return ReturnCode_t::RETCODE_OK;
    }

    ReturnCode_t delete_content_filter(
            const char* filter_class_name,
            IContentFilter* filter_instance) override
    {
        // Check the ContentFilteredTopic should be created by my factory.
        if (0 != strcmp(filter_class_name, "MY_CUSTOM_FILTER"))
        {
            return ReturnCode_t::RETCODE_BAD_PARAMETER;
        }

        // Deletion of the Custom Filter.
        delete(dynamic_cast<MyCustomFilter*>(filter_instance));

        return ReturnCode_t::RETCODE_OK;
    }

};
```

Registering the Factory

To be able to use the Custom Filter in an application, the Custom Filter's factory must be registered in the *DomainParticipant*. Next snippet code shows how to register a factory through API function *register_content_filter_factory()*.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create Custom Filter Factory
MyCustomFilterFactory* factory = new MyCustomFilterFactory();


// Registration of the factory
if (ReturnCode_t::RETCODE_OK !=
        participant->register_content_filter_factory("MY_CUSTOM_FILTER", factory))
{
    // Error
    return;
}
```

Creating a ContentFilteredTopic using the Custom Filter

Creating a ContentFilteredTopic explains how to create a *ContentFilteredTopic*. In the case of using a Custom Filter, *create_contentfilteredtopic()* has an overload adding an argument to select the Custom Filter.

Next snippet code shows how to create a *ContentFilteredTopic* using the Custom Filter.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create the Topic.
Topic* topic =
        participant->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic selecting the Custom Filter and using no expression␣
↪with two parameters
// Filter expression cannot be an empty one even when it is not used by the custom␣
↪filter, as that effectively
// disables any filtering
std::string expression = " ";
std::vector<std::string> parameters;
parameters.push_back("10"); // Parameter for low_mark
parameters.push_back("20"); // Parameter for low_mark
ContentFilteredTopic* filter_topic =
        participant->create_contentfilteredtopic("HelloWorldFilteredTopic1", topic,␣
↪expression, parameters,
                "MY_CUSTOM_FILTER");
if (nullptr == filter_topic)
{
    // Error
    return;
}

// The ContentFilteredTopic instances can then be used to create DataReader objects.
Subscriber* subscriber =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}
```

(continues on next page)

(continued from previous page)

```
DataReader* reader_on_filter = subscriber->create_datareader(filter_topic, DATAREADER_
↪QOS_DEFAULT);
if (nullptr == reader_on_filter)
{
    // Error
    return;
}
```

Important: Even though this specific custom filtering example is not using the filter expression, mind that the expression cannot be an empty string as that disables filtering as explained in *Creating a ContentFilteredTopic*.

Note: Deleting a ContentFilteredTopic which uses a Custom Filter is done exactly in the same manner explained in *Deleting a ContentFilteredTopic*.

Where is filtering applied: writer vs reader side

Content filters may be evaluated on either side, as the DataWriter obtains the filter expression from the DataReader during discovery. Filtering on the writer side can save network bandwidth at the cost of increasing CPU usage on the writer.

Conditions for writer side filtering

A DataWriter will perform filter evaluation in the DataReader stead whenever all of the following conditions are met; filtering will otherwise be performed by the DataReader.

- The DataWriter has infinite liveliness. See *LivelinessQosPolicy*.
- Communication with the DataReader is neither *intra-process* nor *data-sharing*.
- The DataReader is not using multicast.
- The DataWriter is filtering for no more DataReaders than the maximum value set on `reader_filters_allocation`.
  - There is a *resource-limit policy* on *DataWriterQos* that controls the allocation behavior of writer-side filtering resources. Setting a maximum value of 0 disables filter evaluation on the writer side. A maximum value of 32 (the default value) means the writer will perform filter evaluation for up to 32 readers.
  - If the DataWriter is evaluating filters for `writer_resource_limits.reader_filters_allocation.maximum` DataReaders, and a new filtered DataReader is created, then the filter for the newly created DataReader will be evaluated on the reader side.

Discovery race condition

On applications where the filter expression and/or the expression parameters are updated, there may be a situation where the DataWriter will apply the old version of the filter until it receives updated information through discovery. This may imply that a publication made a short time after the DataReader updated the filter, but before the updated discovery information is received by the DataWriter, may not be sent to the DataReader, even if the new filter would have told otherwise. Publications made after the updated discovery information is received will use the updated filter.

If some critical application considers this race condition issue unbearable, filtering on the writer side can be disabled by setting the maximum value on `reader_filters_allocation` to 0.

Fast DDS-Gen for data types source code generation

eProsima Fast DDS comes with a built-in source code generation tool, *Fast DDS-Gen*, which eases the process of translating an IDL specification of a data type to a functional implementation. Thus, this tool automatically generates the source code of a data type defined using IDL. A basic use of the tool is described below. To learn about all the features that *Fast DDS* offers, please refer to *Fast DDS-Gen* section.

Basic usage

Fast DDS can be executed by calling *fastddsgen* on Linux or *fastddsgen.bat* on Windows. The IDL file containing the data type definition is given with the `<IDLfile>` argument.

Linux
`fastddsgen [<options>] <IDLfile> [<IDLfile> ...]`
Windows
`fastddsgen.bat [<options>] <IDLfile> [<IDLfile> ...]`

Among the available arguments defined in *Usage*, the main *Fast DDS-Gen* options for data type source code generation are the following:

- `-replace`: It replaces existing files in case the data type files have been previously generated.
- `-help`: It lists the currently supported platforms and Visual Studio versions.
- `-typeobject`: It builds additional files for `TypeObject` generation and management (see *TypeObject*).
- `-example`: It generates a basic example of a DDS application and the files to build it for the given `platform`. Thus, *Fast DDS-Gen* tool can generate a sample application using the provided data type, together with a *Makefile*, to compile it on Linux distributions, and a Visual Studio project for Windows. To see an example of this please refer to tutorial *Building a publish/subscribe application*.

Output files

Fast DDS-Gen outputs several files. Assuming the IDL file had the name *"Mytype"*, and none of the above options have been defined, these files are:

- MyType.cxx/.h: Type definition.
- MyTypePubSubType.cxx/.h: Serialization and deserialization source code for the data type. It also defines the *getKey()* member function of the `MyTypePubSubType` class in case the topic implements keys (see *Data types with a key*).

If the `-typeobject` argument was used, *MyType.cxx* is modified to register the `TypeObject` representation in the `TypeObjectFactory`, and these files will also be generated:

- MyTypeTypeObject.cxx/.h: TypeObject representation for *MyType* IDL.

6.18 RTPS Layer

The lower level RTPS Layer of *eprosima Fast DDS* serves an implementation of the protocol defined in the RTPS standard. This layer provides more control over the internals of the communication protocol than the *DDS Layer*, so advanced users have finer control over the library's functionalities.

6.18.1 Relation to the DDS Layer

Elements of this layer map one-to-one with elements from the *DDS Layer*, with a few additions. This correspondence is shown in the following table:

DDS Layer	*RTPS Layer*
Domain	RTPSDomain
DomainParticipant	RTPSParticipant
DataWriter	RTPSWriter
DataReader	RTPSReader

6.18.2 How to use the RTPS Layer

We will now go over the use of the RTPS Layer like we did with the *DDS Layer* one, explaining the new features it presents.

We recommend you to look at the two examples describing how to use the RTPS layer that come with the distribution while reading this section. They are located in examples/cpp/rtps/AsSocket and examples/cpp/rtps/Registered

Managing the Participant

Creating a *RTPSParticipant* is done with *RTPSDomain::createParticipant()*. *RTPSParticipantAttributes* structure is used to configure the `RTPSParticipant` upon creation.

```
RTPSParticipantAttributes participant_attr;
participant_attr.setName("participant");
RTPSParticipant* participant = RTPSDomain::createParticipant(0, participant_attr);
```

Managing the Writers and Readers

As the RTPS standard specifies, *RTPSWriters* and *RTPSReaders* are always associated with a *History* element. In the *DDS Layer*, its creation and management is hidden, but in the *RTPS Layer*, you have full control over its creation and configuration.

Writers are created with *RTPSDomain::createRTPSWriter()* and configured with a *WriterAttributes* structure. They also need a *WriterHistory* which is configured with a *HistoryAttributes* structure.

```
HistoryAttributes history_attr;
WriterHistory* history = new WriterHistory(history_attr);
WriterAttributes writer_attr;
RTPSWriter* writer = RTPSDomain::createRTPSWriter(participant, writer_attr, history);
```

Similar to the creation of Writers, Readers are created with *RTPSDomain::createRTPSReader()* and configured with a *ReaderAttributes* structure. A *HistoryAttributes* structure is used to configure the required *ReaderHistory*. Note that in this case, you can provide a specialization of *ReaderListener* class that implements your callbacks:

```
class MyReaderListener : public ReaderListener
{
    // Callbacks override
};
MyReaderListener listener;
HistoryAttributes history_attr;
ReaderHistory* history = new ReaderHistory(history_attr);
ReaderAttributes reader_attr;
RTPSReader* reader = RTPSDomain::createRTPSReader(participant, reader_attr, history, &
↪listener);
```

Using the History to Send and Receive Data

In the RTPS Protocol, Readers and Writers save the data about a topic in their associated Histories. Each piece of data is represented by a Change, which *eprosima Fast DDS* implements as *CacheChange_t*. Changes are always managed by the History.

You can add a new `CacheChange_t` to the History of the Writer to send data. The procedure is as follows:

1. Request a `CacheChange_t` from the Writer with *RTPSWriter::new_change()*. In order to allocate enough memory, you need to provide a callback that returns the maximum number bytes in the payload.
2. Fill the `CacheChange_t` with the data.
3. Add it to the History with *WriterHistory::add_change()*.

The Writer will take care of everything to communicate the data to the Readers.

```
//Request a change from the writer
CacheChange_t* change = writer->new_change([]() -> uint32_t
                {
                    return 255;
                }, ALIVE);
//Write serialized data into the change
change->serializedPayload.length = sprintf((char*) change->serializedPayload.data, "My␣
↪example string %d", 2) + 1;
//Insert change into the history. The Writer takes care of the rest.
history->add_change(change);
```

If your topic data type has several fields, you will have to provide functions to serialize and deserialize your data in and out of the `CacheChange_t`. *Fast DDS-Gen* does this for you.

You can receive data from within the *`ReaderListener::onNewCacheChangeAdded`* callback, as we did in the *DDS Layer*:

1. The callback receives a `CacheChange_t` parameter containing the received data.
2. Process the data within the received `CacheChange_t`.
3. Inform the Reader's History that the change is not needed anymore.

```
class MyReaderListener : public ReaderListener
{
public:

    MyReaderListener()
    {
    }

    ~MyReaderListener()
    {
    }

    void onNewCacheChangeAdded(
            RTPSReader* reader,
            const CacheChange_t* const change)
    {
        // The incoming message is enclosed within the `change` in the function parameters
        printf("%s\n", change->serializedPayload.data);
        // Once done, remove the change
        reader->getHistory()->remove_change((CacheChange_t*)change);
    }

};
```

Managing the Builtin Transports

DDS uses the *Transport Layer* to allow communication between DDS entities. *eProsima Fast DDS* comes with five transports already implemented. However, these transports are not always exclusive between them and in some cases they can be used simultaneously.

You can choose what transports you want to use by disabling the use of builtin transports and manually adding them (see *TransportConfigQos*) or using the default builtin transports behavior and selecting one of the configuration options listed below. Each option modifies the kind of transports that will be instantiated.

Builtin Transports Options	Description
`NONE`	No transport will be instantiated. Hence, the user must manually add the desired transports. Otherwise, the participant creation will fail.
`DEFAULT`	UDPv4 and SHM transports will be instantiated. SHM transport has priority over the UDPv4 transport. Meaning that SHM will always be used when possible.
`DEFAULTv6`	UDPv6 and SHM transports will be instantiated. SHM transport has priority over the UDPv4 transport. Meaning that SHM will always be used when possible.
`SHM`	Only a SHM transport will be instantiated.
`UDPv4`	Only a UDPv4 transport will be instantiated.
`UDPv6`	Only a UDPv6 transport will be instantiated.
`LARGE_DATA`	UDPv4, TCPv4, and SHM transports will be instantiated. However, UDP will only be used for multicast announcements during the participant discovery phase (see *Discovery phases*) while the participant liveliness and the application data delivery occurs over TCP or SHM. This configuration is useful when working with large data.(See *Large Data mode and Fast DDS over TCP*).

```
RTPSParticipantAttributes participant_attr;
participant_attr.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_
↪DATA);
RTPSParticipant* participant = RTPSDomain::createParticipant(0, participant_attr);
```

The same result can also be obtained using the *setup_transports()* wrapper function of the *DomainParticipantQos*, XML profiles (see *RTPS element type*) or the `FASTDDS_BUILTIN_TRANSPORTS` environment variable (see *FASTDDS_BUILTIN_TRANSPORTS*).

Note: TCPv4 transport is initialized with the following configuration:

- *calculate_crc*, *check_crc* and *apply_security* are set to false.
- *enable_tcp_nodelay* is set to true.

Warning: To obtain a better performance when working with large data messages it is extremely recommended to set through *DomainParticipantQos* the send and receive buffer sizes of the TCP transport to the maximum default system value. See *Finding out system maximum values* to learn how to check it.

6.18.3 Configuring Readers and Writers

One of the benefits of using the *RTPS Layer* is that it provides new configuration possibilities while maintaining the options from the DDS layer. For example, you can set a Writer or a Reader as a Reliable or Best-Effort endpoint as previously:

```
writer_attr.endpoint.reliabilityKind = BEST_EFFORT;
```

Setting the data durability kind

The Durability parameter defines the behavior of the Writer regarding samples already sent when a new Reader matches. *eProsima Fast DDS* offers three Durability options:

- VOLATILE (default): Messages are discarded as they are sent. If a new Reader matches after message *n*, it will start received from message *n+1*.
- TRANSIENT_LOCAL: The Writer saves a record of the last *k* messages it has sent. If a new reader matches after message *n*, it will start receiving from message *n-k*
- TRANSIENT: As TRANSIENT_LOCAL, but the record of messages will be saved to persistent storage, so it will be available if the writer is destroyed and recreated, or in case of an application crash.

To choose your preferred option:

```
writer_attr.endpoint.durabilityKind = TRANSIENT_LOCAL;
```

Because in the *RTPS Layer* you have control over the History, in TRANSIENT_LOCAL and TRANSIENT modes the Writer sends all changes you have not explicitly released from the History.

6.18.4 Configuring the History

The History has its own configuration structure, the *HistoryAttributes*.

Changing the maximum size of the payload

You can choose the maximum size of the Payload that can go into a `CacheChange_t`. Be sure to choose a size that allows it to hold the biggest possible piece of data:

```
history_attr.payloadMaxSize  = 250;//Defaults to 500 bytes
```

Changing the size of the History

You can specify a maximum amount of changes for the History to hold and an initial amount of allocated changes:

```
history_attr.initialReservedCaches = 250; //Defaults to 500
history_attr.maximumReservedCaches = 500; //Defaults to 0 = Unlimited Changes
```

When the initial amount of reserved changes is lower than the maximum, the History will allocate more changes as they are needed until it reaches the maximum size.

6.18.5 Using a custom Payload Pool

A *Payload* is defined as the data the user wants to transmit between a Writer and a Reader. RTPS needs to add some metadata to this Payload in order to manage the communication between the endpoints. Therefore, this Payload is encapsulated inside the `SerializedPayload_t` field of the `CacheChange_t`, while the rest of the fields of the `CacheChange_t` provide the required metadata.

WriterHistory and *ReaderHistory* provide an interface for the user to interact with these changes: Changes to be transmitted by the Writer are added to its WriterHistory, and changes already processed on the Reader can be removed from the ReaderHistory. In this sense, the History acts as a buffer for changes that are not fully processed yet.

During a normal execution, new changes are added to the History and old ones are removed from it. In order to manage the lifecycle of the Payloads contained in these changes, Readers and Writers use a pool object, an implementation of the *IPayloadPool* interface. Different pool implementations allow for different optimizations. For example, Payloads of different size could be retrieved from different preallocated memory chunks.

Writers and Readers can automatically select a default Payload pool implementation that best suits the configuration given in *HistoryAttributes*. However, a custom Payload pool can be given to *RTPSDomain::createRTPSWriter()* and *RTPSDomain::createRTPSReader()* functions. Writers and Readers will use the provided pool when a new `CacheChange_t` is requested or released.

IPayloadPool interface

- *IPayloadPool::get_payload* overload with size parameter:

  Ties an empty Payload of the requested size to a `CacheChange_t` instance. The Payload can then be filled with the required data.

- *IPayloadPool::get_payload* overload with SerializadPayload parameter:

  Copies the given Payload data to a new Payload from the pool and ties it to the `CacheChange_t` instance. This overload also takes a pointer to the pool that owns the original Payload. This allows certain optimizations, like sharing the Payload if the original one comes form this same pool, therefore avoiding the copy operation.

- *IPayloadPool::release_payload*:

  Returns the Payload tied to a `CacheChange_t` to the pool, and breaks the tie.

Important: When implementing a custom Payload pool, make sure that the allocated Payloads fulfill the requirements of standard RTPS serialization. Specifically, the Payloads must be large enough to accommodate the serialized user data plus the 4 octets of the *SerializedPayloadHeader* as specified in section 10.2 of the RTPS standard.

For example, if we know the upper bound of the serialized user data, we may consider implementing a pool that always allocates Payloads of a fixed size, large enough to hold any of this data. If the serialized user data has at most N octets, then the allocated Payloads must have at least N+4 octets.

Note that the size requested to *IPayloadPool::get_payload* already considers this 4 octet header.

Default Payload pool implementation

If no custom Payload pool is provided to the Writer or Reader, *Fast DDS* will automatically use the default implementation that best matches the *memoryPolicy* configuration of the History.

PREALLOCATED_MEMORY_MODE

All payloads will have a data buffer of fixed size, equal to the value of *payloadMaxSize*, regardless of the size requested to *IPayloadPool::get_payload*. Released Payloads can be reused for another `CacheChange_t`. This reduces memory allocation operations at the cost of higher memory usage.

During the initialization of the History, *initialReservedCaches* Payloads are preallocated for the initially allocated `CacheChange_t`.

PREALLOCATED_WITH_REALLOC_MEMORY_MODE

Payloads are guaranteed to have a data buffer at least as large as the maximum between the requested size and *payloadMaxSize*. Released Payloads can be reused for another `CacheChange_t`. If there is at least one free Payload with a buffer size equal or larger to the requested one, no memory allocation is done.

During the initialization of the History, *initialReservedCaches* Payloads are preallocated for the initially allocated CacheChange_t.

DYNAMIC_RESERVE_MEMORY_MODE

Every time a Payload is requested, a new one is allocated in memory with the appropriate size. *payloadMaxSize* is ignored. The memory of released Payloads is always deallocated, so there are never free Payloads in the pool. This reduces memory usage at the cost of frequent memory allocations.

No preallocation of Payloads is done in the initialization of the History,

DYNAMIC_REUSABLE_MEMORY_MODE

Payloads are guaranteed to have a data buffer at least as large as the requested size. *payloadMaxSize* is ignored.

Released Payloads can be reused for another CacheChange_t. If there is at least one free Payload with a buffer size equal or larger to the requested one, no memory allocation is done.

Example using a custom Payload pool

```
// A simple payload pool that reserves and frees memory each time
class CustomPayloadPool : public IPayloadPool
{
    bool get_payload(
            uint32_t size,
            CacheChange_t& cache_change) override
    {
        // Reserve new memory for the payload buffer
        octet* payload = new octet[size];

        // Assign the payload buffer to the CacheChange and update sizes
        cache_change.serializedPayload.data = payload;
        cache_change.serializedPayload.length = size;
        cache_change.serializedPayload.max_size = size;

        // Tell the CacheChange who needs to release its payload
        cache_change.payload_owner(this);

        return true;
    }

    bool get_payload(
            SerializedPayload_t& data,
            IPayloadPool*& /*data_owner*/,
            CacheChange_t& cache_change)
    {
        // Reserve new memory for the payload buffer
        octet* payload = new octet[data.length];

        // Copy the data
        memcpy(payload, data.data, data.length);

        // Tell the CacheChange who needs to release its payload
        cache_change.payload_owner(this);
```

(continues on next page)

(continued from previous page)

```
        // Assign the payload buffer to the CacheChange and update sizes
        cache_change.serializedPayload.data = payload;
        cache_change.serializedPayload.length = data.length;
        cache_change.serializedPayload.max_size = data.length;

        return true;
    }

    bool release_payload(
            CacheChange_t& cache_change) override
    {
        // Ensure precondition
        if (this != cache_change.payload_owner())
        {
            std::cerr << "Trying to release a payload buffer allocated by a different␣
↪PayloadPool." << std::endl;
            return false;
        }

        // Dealloc the buffer of the payload
        delete[] cache_change.serializedPayload.data;

        // Reset sizes and pointers
        cache_change.serializedPayload.data = nullptr;
        cache_change.serializedPayload.length = 0;
        cache_change.serializedPayload.max_size = 0;

        // Reset the owner of the payload
        cache_change.payload_owner(nullptr);

        return true;
    }

};

std::shared_ptr<CustomPayloadPool> payload_pool = std::make_shared<CustomPayloadPool>();

// A writer using the custom payload pool
HistoryAttributes writer_history_attr;
WriterHistory* writer_history = new WriterHistory(writer_history_attr);
WriterAttributes writer_attr;
RTPSWriter* writer = RTPSDomain::createRTPSWriter(participant, writer_attr, payload_pool,
↪ writer_history);

// A reader using the same instance of the custom payload pool
HistoryAttributes reader_history_attr;
ReaderHistory* reader_history = new ReaderHistory(reader_history_attr);
ReaderAttributes reader_attr;
RTPSReader* reader = RTPSDomain::createRTPSReader(participant, reader_attr, payload_pool,
↪ reader_history);

// Write and Read operations work as usual, but take the Payloads from the pool.
```

(continues on next page)

(continued from previous page)

```
// Requesting a change to the Writer will provide one with an empty Payload taken from␣
↪the pool
CacheChange_t* change = writer->new_change([]() -> uint32_t
                {
                    return 255;
                }, ALIVE);

// Write serialized data into the change and add it to the history
change->serializedPayload.length = sprintf((char*) change->serializedPayload.data, "My␣
↪example string %d", 2) + 1;
writer_history->add_change(change);
```

6.19 Discovery

Fast DDS, as a Data Distribution Service (DDS) implementation, provides discovery mechanisms that allow for automatically finding and matching *DataWriters* and *DataReaders* across *DomainParticipants* so they can start sharing data. This discovery is performed, for all the mechanisms, in two phases.

6.19.1 Discovery phases

1. **Participant Discovery Phase (PDP)**: During this phase the *DomainParticipants* acknowledge each other's existence. To do that, each DomainParticipant sends periodic announcement messages, which specify, among other things, unicast addresses (IP and port) where the DomainParticipant is listening for incoming meta and user data traffic. Two given DomainParticipants will match when they exist in the same DDS Domain. By default, the announcement messages are sent using well-known multicast addresses and ports (calculated using the `DomainId`). Furthermore, it is possible to specify a list of addresses to send announcements using unicast (see in *Initial peers*). Moreover, is is also possible to configure the periodicity of such announcements (see *Discovery Configuration*).
2. **Endpoint Discovery Phase (EDP)**: During this phase, the *DataWriters* and *DataReaders* acknowledge each other. To do that, the DomainParticipants share information about their DataWriters and DataReaders with each other, using the communication channels established during the PDP. This information contains, among other things, the *Topic* and data type (see *Topic*). For two endpoints to match, their topic and data type must coincide. Once DataWriter and DataReader have matched, they are ready for sending/receiving user data traffic.

Important: It is possible to use the PDP phase to transmit information about the host, user, and process (physical information) in which the *DomainParticipant* is running. Please refer to *Physical Data in Discovery Information* for more information about how to configure the transmitted physical data.

6.19.2 Discovery mechanisms

Fast DDS provides the following discovery mechanisms:

- *Simple Discovery*: This is the default mechanism. It upholds the RTPS standard for both PDP and EDP, and therefore provides compatibility with any other DDS and RTPS implementations.
- *Static Discovery*: This mechanisms uses the Simple Participant Discovery Protocol (SPDP) for the PDP phase (as specified by the RTPS standard), but allows for skipping the Simple Endpoint Discovery Protocol (SEDP) phase when all the DataWriters' and DataReaders' IPs and ports, data types, and Topics are known beforehand.
- *Discovery Server*: This discovery mechanism uses a centralized discovery architecture, where a DomainParticipant, referred as Server, acts as a hub for meta traffic discovery.
- **Manual Discovery**: This mechanism is only compatible with the RTPS layer. It disables the PDP, letting the user to manually match and unmatch `RTPSParticipants`, `RTPSReaders`, and `RTPSWriters` using whatever external meta-information channel of its choice. Therefore, the user must access the RTPSParticipant implemented by the DomainParticipant and directly match the RTPS Entities.

6.19.3 Discovery settings

The following sections list and describe the settings available for each of the previously defined discovery mechanisms, as well as how to define the *DomainParticipantListener* discovery callbacks.

General Discovery Settings

Some discovery settings are shared across the different discovery mechanisms. These settings are defined under the `builtin` public data member of the `WireProtocolConfigQos` class. These are:

Name	Description	Type	Default
Discovery Protocol	The discovery protocol to use (see *Discovery mechanisms*).	`DiscoveryProtocol`	`SIMPLE`
Ignore Participant flags	Filter discovery traffic for *DomainParticipants* in the same process, in different processes, or in different hosts.	`ParticipantFilteringFlags`	`NO_FILTER`
Lease Duration	Indicates for how much time should a remote DomainParticipant consider the local DomainParticipant to be alive.	`Duration_t`	20 s
Announcement Period	The period for the DomainParticipant to send PDP announcements.	`Duration_t`	3 s

Discovery Protocol

Specifies the discovery protocol to use (see *Discovery mechanisms*). The possible values are:

Discovery Mechanism	Possible values	Description
Simple	*SIMPLE*	Simple discovery protocol as specified in the RTPS standard.
Static	STATIC	SPDP with manual EDP specified in XML files.
Discovery Server	*SERVER*	The DomainParticipant acts as a hub for discovery traffic, receiving and distributing discovery information.
	CLIENT	The DomainParticipant acts as a client for discovery traffic. It sends its discovery information to the server, and it receives only the information that is relevant to it.
	SUPER_CLIENT	The DomainParticipant acts as a client for discovery traffic. It sends its discovery information to the server, and it receives all other discovery information from the server.
	BACKUP	Creates a SERVER DomainParticipant which has a persistent `sqlite` database. A BACKUP server can load the a database on start. This type of sever makes the Discovery Server architecture resilient to server destruction.
Manual	*NONE*	Disables PDP phase, therefore there is no EDP phase. All matching must be done manually through the `addReaderLocator`, `addReaderProxy`, `addWriterProxy` RTPS layer methods.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        DiscoveryProtocol_t::SIMPLE;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="participant_discovery_protocol">
            <rtps>
                <builtin>
                    <discovery_config>
                        <discoveryProtocol>SIMPLE</discoveryProtocol>
                    </discovery_config>
                </builtin>
            </rtps>
        </participant>
    </profiles>
</dds>
```

Ignore Participant flags

Defines a filter to ignore some discovery traffic when received. This is useful to add an extra level of DomainParticipant isolation. The possible values are:

Possible values	Description
NO_FILTER	All Discovery traffic is processed.
FILTER_DIFFERENT_HOST	Discovery traffic from another host is discarded.
FILTER_DIFFERENT_PROCESS	Discovery traffic from another process on the same host is discarded.
FILTER_SAME_PROCESS	Discovery traffic from DomainParticipant's own process is discarded.
FILTER_DIFFERENT_PROCESS | *FILTER_SAME_PROCESS*	Discovery traffic from DomainParticipant's own host is discarded.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.ignoreParticipantFlags =
        static_cast<eprosima::fastrtps::rtps::ParticipantFilteringFlags_t>(
    ParticipantFilteringFlags_t::FILTER_DIFFERENT_PROCESS |
    ParticipantFilteringFlags_t::FILTER_SAME_PROCESS);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="participant_discovery_ignore_flags">
            <rtps>
                <builtin>
                    <discovery_config>
                        <ignoreParticipantFlags>FILTER_DIFFERENT_PROCESS | FILTER_SAME_
↪PROCESS</ignoreParticipantFlags>
                    </discovery_config>
                </builtin>
            </rtps>
        </participant>
    </profiles>
</dds>
```

Note: To configure a *DomainParticipant* to not receive data from its own *DataWriters*, please refer to *Ignore Local Endpoints*.

Lease Duration

Indicates for how much time should a remote DomainParticipant consider the local DomainParticipant to be alive. If the liveliness of the local DomainParticipant has not being asserted within this time, the remote DomainParticipant considers the local DomainParticipant dead and destroys all the information regarding the local DomainParticipant and all its endpoints.

The local DomainParticipant's liveliness is asserted on the remote DomainParticipant any time the remote DomainParticipant receives any kind of traffic from the local DomainParticipant.

The lease duration is specified as a time expressed in seconds and nanosecond using a *Duration_t*.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.leaseDuration = Duration_t(10, 20);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="participant_discovery_lease_duration">
            <rtps>
                <builtin>
                    <discovery_config>
                        <leaseDuration>
                            <sec>10</sec>
                            <nanosec>20</nanosec>
                        </leaseDuration>
                    </discovery_config>
                </builtin>
            </rtps>
        </participant>
    </profiles>
</dds>
```

Announcement Period

It specifies the periodicity of the DomainParticipant's PDP announcements. For liveliness' sake it is recommend that the announcement period is shorter than the lease duration, so that the DomainParticipant's liveliness is asserted even when there is no data traffic. It is important to note that there is a trade-off involved in the setting of the announcement period, i.e. too frequent announcements will bloat the network with meta traffic, but too scarce ones will delay the discovery of late joiners.

DomainParticipant's announcement period is specified as a time expressed in seconds and nanosecond using a *Duration_t*.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.leaseDuration_announcementperiod =␣
↪Duration_t(1, 2);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="participant_discovery_lease_announcement">
            <rtps>
                <builtin>
                    <discovery_config>
                        <leaseAnnouncement>
                            <sec>1</sec>
                            <nanosec>2</nanosec>
                        </leaseAnnouncement>
                    </discovery_config>
                </builtin>
            </rtps>
        </participant>
    </profiles>
</dds>
```

SIMPLE Discovery Settings

The SIMPLE discovery protocol resolves the establishment of the end-to-end connection between various DDS Entities. *eProsima Fast DDS* implements the SIMPLE discovery protocol to provide compatibility with the RTPS standard. The specification splits up the SIMPLE discovery protocol into two independent protocols:

- **Simple Participant Discovery Protocol (SPDP):** specifies how DomainParticipants discover each other in the network; it announces and detects the presence of DomainParticipants within the same domain.
- **Simple Endpoint Discovery Protocol (SEDP):** defines the protocol adopted by the discovered DomainParticipants for the exchange of information in order to discover the DDS Entities contained in each of them, i.e. the *DataWriter* and *DataReader*.

Name	Description
Initial Announcements	It defines the behavior of the DomainParticipants initial announcements.
Simple EDP Attributes	It defines the use of the SIMPLE protocol as a discovery protocol.
Initial peers	A list of DomainParticipant's IP/port pairs to which the SPDP announcements are sent.

Initial Announcements

RTPS standard simple discovery mechanism requires the DomainParticipants to send announcements of their presence in the domain. These announcements are not delivered in a reliable fashion, and can be disposed of by the network. In order to avoid the discovery delay induced by message disposal, the initial announcement can be set up to make several shots, in order to increase proper reception chances. See `InitialAnnouncementConfig`.

Initial announcements only take place upon participant creation. Once this phase is over, the only announcements enforced are the standard ones based on the `leaseDuration_announcementperiod` period (not the `period`).

Name	Description	Type	Default
count	It defines the number of announcements to send at start-up.	`uint32_t`	5
period	It defines the specific period for initial announcements.	`Duration_t`	100ms

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.initial_announcements.count = 5;
pqos.wire_protocol().builtin.discovery_config.initial_announcements.period = Duration_
↪t(0, 100000000u);
```

XML

```
<participant profile_name="participant_profile_simple_discovery">
    <rtps>
        <builtin>
            <discovery_config>
                <initialAnnouncements>
                    <count>5</count>
                    <period>
                        <nanosec>100000000</nanosec>
                    </period>
                </initialAnnouncements>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Simple EDP Attributes

Name	Description	Type	Default
SIMPLE EDP	It defines the use of the SIMPLE protocol as a discovery protocol for the EDP phase. A DomainParticipant may create DataWriters, DataReaders, both or neither.	`bool`	true
Publication writer and Subscription reader	It is intended for DomainParticipants that implement only one or more DataWriters, i.e. do not implement DataReaders. It allows the creation of only DataReader discovery related EDP endpoints.	`bool`	true
Publication reader and Subscription writer	It is intended for DomainParticipants that implement only one or more DataReaders, i.e. do not implement DataWriters. It allows the creation of only DataWriter discovery related EDP endpoints.	`bool`	true

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol =␣
↪true;
pqos.wire_protocol().builtin.discovery_config.m_simpleEDP.use_
↪PublicationWriterANDSubscriptionReader = true;
pqos.wire_protocol().builtin.discovery_config.m_simpleEDP.use_
↪PublicationReaderANDSubscriptionWriter = false;
```

XML

```
<participant profile_name="participant_profile_qos_discovery_edp">
    <rtps>
        <builtin>
            <discovery_config>
                <EDP>SIMPLE</EDP>
                <simpleEDP>
                    <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
                    <PUBREADER_SUBWRITER>false</PUBREADER_SUBWRITER>
                </simpleEDP>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Initial peers

According to the RTPS standard (Section 9.6.1.1), each *RTPSParticipant* must listen for incoming Participant Discovery Protocol (PDP) discovery metatraffic in two different ports, one linked to a multicast address and another one linked to a unicast address. *Fast DDS* allows for the configuration of an initial peers list which contains one or more such IP-port address pairs corresponding to remote DomainParticipants PDP discovery listening resources, so that the local DomainParticipant will not only send its PDP traffic to the default multicast address-port specified by its domain, but also to all the IP-port address pairs specified in the initial peers list.

A DomainParticipant's initial peers list contains the list of IP-port address pairs of all other DomainParticipants with which it will communicate. It is a list of addresses that a DomainParticipant will use in the unicast discovery mechanism, together or as an alternative to multicast discovery. Therefore, this approach also applies to those scenarios in which multicast functionality is not available.

According to the RTPS standard (Section 9.6.1.1), the RTPSParticipants' discovery traffic unicast listening ports are calculated using the following equation: 7400 + 250 * *domainID* + 10 + 2 * *participantID*. Thus, if for example a RTPSParticipant operates in Domain 0 (default domain) and its ID is 1, its discovery traffic unicast listening port would be: 7400 + 250 * 0 + 10 + 2 * 1 = 7412. By default *eProsima Fast DDS* uses as initial peers the Metatraffic Multicast Locators.

The following constitutes an example configuring an Initial Peers list with one peer on host 192.168.10.13 with DomainParticipant ID 1 in domain 0.

Note: There is also the possibility of not defining the initial peer port. In this case, the discovery information would be sent to every port ranging from *participantID* zero to the `maxInitialPeersRange` value set in the *TransportDe-*

scriptorInterface. Consequently, setting this value to at least the maximum expected number of DomainParticipants will ensure discovery and communication.

C++

```
DomainParticipantQos qos;

// configure an initial peer on host 192.168.10.13.
// The port number corresponds to the well-known port for metatraffic unicast
// on participant ID `1` and domain `0`.
Locator_t initial_peer;
IPLocator::setIPv4(initial_peer, "192.168.10.13");
initial_peer.port = 7412;
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer);
```

XML

```
<!--
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
-->
    <participant profile_name="initial_peers_example_profile" is_default_profile="true
↪">
        <rtps>
            <builtin>
                <initialPeersList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.13</address>
                            <port>7412</port>
                        </udpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
```

STATIC Discovery Settings

Fast DDS allows for the substitution of the SEDP protocol for the EDP phase with a static version that completely eliminates EDP meta traffic. This can become useful when dealing with limited network bandwidth and a well-known schema of *DataWriters* and *DataReaders*. If all DataWriters and DataReaders, and their *Topics* and data types, are known beforehand, the EDP phase can be replaced with a static configuration of peers. It is important to note that by doing this, no EDP discovery meta traffic will be generated, and only those peers defined in the configuration will be able to communicate. The STATIC discovery related settings are:

Name	Description
STATIC EDP	It activates the STATIC discovery protocol.
STATIC EDP XML Configuration Specification	Specifies an XML content with a description of the remote DataWriters and DataReaders.
Initial Announcements	It defines the behavior of the DomainParticipant initial announcements (PDP phase).

STATIC EDP

To activate the STATIC EDP, the SEDP must be disabled on the *WireProtocolConfigQos*. This can be done either by code or using an XML configuration file:

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol =␣
↪false;
pqos.wire_protocol().builtin.discovery_config.use_STATIC_EndpointDiscoveryProtocol =␣
↪true;
```

XML

```
<participant profile_name="participant_profile_static_edp">
    <rtps>
        <builtin>
            <discovery_config>
                <EDP>STATIC</EDP>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Currently two different formats of exchanging information in the Participant Discovery Phase (PDP) are supported: the default one and another that reduces the network bandwidth used. *Static Discovery's Exchange Format* explains how to change this.

STATIC EDP XML Configuration Specification

Since activating STATIC EDP suppresses all EDP meta traffic, the information about the remote entities (DataWriters and DataReaders) must be statically specified, which is done using dedicated XML files. A *DomainParticipant* may load several of such configuration files so that the information about different entities can be contained in one file, or split into different files to keep it more organized. *Fast DDS* provides a Static Discovery example that implements this EDP discovery protocol.

The following table describes all the possible elements of a STATIC EDP XML configuration file. A full example of such file can be found in *STATIC EDP XML Example*.

Name	Description	Values	Default
<userId>	Mandatory. Uniquely identifies the DataReader/DataWriter.	uint16_t	0
<entityID>	EntityId of the DataReader/DataWriter.	uint16_t	0
<expectsInlineQos>	Indicates if QOS is expected inline (DataReader **only**).	bool	false
<topicName>	Mandatory. The topic of the remote DataReader/DataWriter. Should match with one of the topics of the local DataReaders/DataWriters.	string_255	
<topicDataType>	Mandatory. The data type of the topic.	string_255	
<topicKind>	The kind of topic.	NO_KEY WITH_KEY	NO_KEY
<partitionQos>	The name of a partition of the remote peer. Repeat to configure several partitions.	string	
<unicastLocator>	Unicast locator of the DomainParticipant. See *Locators definition*.		
<multicastLocator>	Multicast locator of the DomainParticipant. See *Locators definition*.		
<reliabilityQos>	See *ReliabilityQosPolicy* section.	*BEST_EFFORT_RELIABILITY_QOS* *RELIABLE_RELIABILITY_QOS*	*BEST_EFFORT_RELIABILITY_*
<durabilityQos>	See *DurabilityQosPolicy* section.	*VOLATILE_DURABILITY_QOS* *TRANSIENT_LOCAL_DURABILITY_QOS* *TRANSIENT_DURABILITY_QOS*	*VOLATILE_DURABILITY_QOS*
<ownershipQos>	See *Ownership QoS*.		
<livelinessQos>	Defines the liveliness of the remote peer. See *Liveliness QoS*.		
<disablePositiveACKs>	See *DisablePositiveACKsQosPolicy*.	See *DisablePositiveAcks*	

Locators definition

Locators for remote peers are configured using <unicastLocator> and <multicastLocator> tags. These take no value, and the locators are defined using tag elements. Locators defined with <unicastLocator> and <multicastLocator> are accumulative, so they can be repeated to assign several remote endpoints locators to the same peer.

- address: a mandatory string representing the locator address.
- port: an optional uint16_t representing a port on that address.

Ownership QoS

The ownership of the topic can be configured using <ownershipQos> tag. It takes no value, and the configuration is done using tag elements:

- kind: can be one of *SHARED_OWNERSHIP_QOS* or *EXCLUSIVE_OWNERSHIP_QOS*. This element is mandatory withing the tag.
- strength: an optional uint32_t specifying how strongly the remote DomainParticipant owns the *Topic*. This QoS can be set on DataWriters **only**. If not specified, default value is zero.

Liveliness QoS

The *LivelinessQosPolicy* of the remote peer is configured using `<livelinessQos>` tag. It takes no value, and the configuration is done using tag elements:

- `kind`: can be any of *AUTOMATIC_LIVELINESS_QOS*, *MANUAL_BY_PARTICIPANT_LIVELINESS_QOS* or *MANUAL_BY_TOPIC_LIVELINESS_QOS*. This element is mandatory withing the tag.
- `leaseDuration_ms`: an optional `uint32` specifying the lease duration for the remote peer. The special value `INF` can be used to indicate infinite lease duration. If not specified, default value is `INF`

Checking STATIC EDP XML Files

Before loading a static EDP XML file, it would be useful to check its validity and make sure the file will be successfully loaded. This verification can be performed on *DomainParticipantFactory* using *DomainParticipantFactory::check_xml_static_discovery()*, using either XML files or the configuration directly, as in the examples below.

File

```
// The (file://) flag is optional.
std::string file = "file://static_Discovery.xml";
DomainParticipantFactory* factory = DomainParticipantFactory::get_instance();
if (ReturnCode_t::RETCODE_OK != factory->check_xml_static_discovery(file))
{
    std::cout << "Error parsing xml file " << file << std::endl;
}
```

Data

```
// The (data://) flag is required to load the configuration directly.
std::string fileData = "data://<?xml version=\"1.0\" encoding=\"utf-8\"?>" \
        "<staticdiscovery>" \
        "<participant>" \
        "<name>HelloWorldPublisher</name>" \
        "<writer>" \
        "<userId>1</userId>" \
        "<entityID>2</entityID>" \
        "<topicName>HelloWorldTopic</topicName>" \
        "<topicDataType>HelloWorld</topicDataType>" \
        "</writer>" \
        "</participant>" \
        "</staticdiscovery>";
if (ReturnCode_t::RETCODE_OK != factory->check_xml_static_discovery(fileData))
{
    std::cout << "Error parsing xml file data:" << std::endl << fileData << std::endl;
}
```

STATIC EDP XML Example

The following is a complete example of a configuration XML file for two remote DomainParticipant, a DataWriter and a DataReader. This configuration **must** agree with the configuration used to create the remote DataReader/DataWriter. Otherwise, communication between DataReaders and DataWriters may be affected. If any non-mandatory element is missing, it will take the default value. As a rule of thumb, all the elements that were specified on the remote DataReader/DataWriter creation should be configured.

XML

```
<staticdiscovery>
    <participant>
        <name>HelloWorldSubscriber</name>
        <reader>
            <userId>3</userId>
            <entityID>4</entityID>
            <expectsInlineQos>true</expectsInlineQos>
            <topicName>HelloWorldTopic</topicName>
            <topicDataType>HelloWorld</topicDataType>
            <topicKind>WITH_KEY</topicKind>
            <partitionQos>HelloPartition</partitionQos>
            <partitionQos>WorldPartition</partitionQos>
            <unicastLocator address="192.168.0.128" port="5000"/>
            <unicastLocator address="10.47.8.30" port="6000"/>
            <multicastLocator address="239.255.1.1" port="7000"/>
            <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
            <ownershipQos kind="SHARED_OWNERSHIP_QOS"/>
            <livelinessQos kind="AUTOMATIC_LIVELINESS_QOS" leaseDuration_ms="1000"/>
            <disablePositiveAcks>
                <enabled>true</enabled>
            </disablePositiveAcks>
        </reader>
    </participant>
    <participant>
        <name>HelloWorldPublisher</name>
        <writer>
            <unicastLocator address="192.168.0.120" port="9000"/>
            <unicastLocator address="10.47.8.31" port="8000"/>
            <multicastLocator address="239.255.1.1" port="7000"/>
            <userId>5</userId>
            <entityID>6</entityID>
            <topicName>HelloWorldTopic</topicName>
            <topicDataType>HelloWorld</topicDataType>
            <topicKind>WITH_KEY</topicKind>
            <partitionQos>HelloPartition</partitionQos>
            <partitionQos>WorldPartition</partitionQos>
            <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
            <ownershipQos kind="SHARED_OWNERSHIP_QOS" strength="50"/>
            <livelinessQos kind="AUTOMATIC_LIVELINESS_QOS" leaseDuration_ms="1000"/>
            <disablePositiveAcks>
                <enabled>true</enabled>
                <duration>
                    <sec>300</sec>
                </duration>
            </disablePositiveAcks>
        </writer>
    </participant>
</staticdiscovery>
```

Loading STATIC EDP XML Files

Statically discovered remote DataReaders/DataWriters **must** define a unique *userID* on their profile, whose value **must** agree with the one specified in the discovery configuration XML. This is done by setting the user ID on the *DataReaderQos*/*DataWriterQos*:

C++

```
// Configure the DataWriter
DataWriterQos wqos;
wqos.endpoint().user_defined_id = 1;

// Configure the DataReader
DataReaderQos rqos;
rqos.endpoint().user_defined_id = 3;
```

XML

```
<data_writer profile_name="writer_xml_conf_static_discovery">
    <userDefinedID>3</userDefinedID>
</data_writer>

<data_reader profile_name="reader_xml_conf_static_discovery">
    <userDefinedID>5</userDefinedID>
</data_reader>
```

On the local DomainParticipant, you can load STATIC EDP configuration content specifying the file containing it.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.static_edp_xml_config("file://
↪RemotePublisher.xml");
pqos.wire_protocol().builtin.discovery_config.static_edp_xml_config("file://
↪RemoteSubscriber.xml");
```

XML

```
<participant profile_name="participant_profile_static_load_xml">
    <rtps>
        <builtin>
            <discovery_config>
                <static_edp_xml_config>file://RemotePublisher.xml</static_edp_xml_
↪config>
                <static_edp_xml_config>file://RemoteSubscriber.xml</static_edp_xml_
↪config>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Or you can specify the STATIC EDP configuration content directly.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.static_edp_xml_config(
    "data://<?xml version=\"1.0\" encoding=\"utf-8\"?>" \
    "<staticdiscovery><participant><name>RTPSParticipant</name></participant></
↪staticdiscovery>");
```

Discovery Server Settings

This mechanism is based on a client-server discovery paradigm, i.e. the metatraffic (message exchange among *DomainParticipants* to identify each other) is managed by one or several server DomainParticipants (left figure), as opposed to simple discovery (right figure), where metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol. A Discovery-Server tool is available to ease Discovery Server setup and testing.

Note: *DDS Domain* concept does not apply when enabling the Discovery Server mechanism.

- *Key concepts*
- *Choosing between Client and Server*
- *The GuidPrefix as the server unique identifier*
- *The server locator list*
- *Fine tuning discovery server handshake*
- *Modifying remote servers list at run time*
- *Configure Discovery Server locators using names*
- *Full example*
- *Security*

Fig. 9: Comparison of Discovery Server and Simple discovery mechanisms

Key concepts

In this architecture there are several key concepts to understand:

- The Discovery Server mechanism reuses the RTPS discovery messages structure, as well as the standard DDS *DataWriters* and *DataReaders*.
- Discovery Server DomainParticipants may be *clients* or *servers*. The only difference between them is how they handle discovery traffic. The user traffic, that is, the traffic among the DataWriters and DataReaders they create, is role-independent.

- All *server* and *client* discovery information will be shared with linked *clients*. Note that a *server* may act as a *client* for other *servers*.
- A *SERVER* is a participant to which the *clients* (and maybe other *servers*) send their discovery information. The role of the *server* is to re-distribute the *clients* (and *servers*) discovery information to their known *clients* and *servers*. A *server* may connect to other *servers* to receive information about their *clients*. Known *servers* will receive all the information known by the *server*. Known *clients* will only receive the information they need to establish communication, i.e. the information about the DomainParticipants, DataWriters, and DataReaders to which they match. This means that the *server* runs a "matching" algorithm to sort out which information is required by which *client*.
- A *BACKUP server* is a *server* that persists its discovery database into a file. This type of *server* can load the network graph from a file on start-up without the need of receiving any *client's* information. It can be used to persist the *server* knowledge about the network between runs, thus securing the *server's* information in case of unexpected shutdowns. It is important to note that the discovery times will be negatively affected when using this type of *server*, since periodically writing to a file is an expensive operation.
- A *CLIENT* is a participant that connects to one or more *servers* from which it receives only the discovery information they require to establish communication with matching endpoints.
- *Clients* require prior knowledge of the *servers* to which they want to link. Basically it is reduced to the *servers* identity (henceforth called *GuidPrefix_t*) and a list of locators where the *servers* are listening. These locators also define the transport protocol (UDP or TCP) the client will use to contact the *server*.
  - The *GuidPrefix_t* is the RTPS standard RTPSParticipant unique identifier, a 12-byte chain. This identifier allows *clients* to assess whether they are receiving messages from the right *server*, as each standard RTPS message contains this piece of information.

    The *GuidPrefix_t* is used because the *server's* IP address may not be a reliable enough server identifier, since several *servers* can be hosted in the same machine, thus having the same IP, and also because multicast addresses are acceptable addresses.
- A *SUPER_CLIENT* is a *client* that receives the discovery information known by the *server*, in opposition to *clients*, which only receive the information they need.

Note: A *SUPER_CLIENT* does not behave as a *Server* as it only receives the discovery information through the *Server* to which it is connected. Any DomainParticipant discovered by the *Server* with no endpoints will not be known by the *SUPER_CLIENT*.

- *Servers* do not require any prior knowledge of their *clients*, but their *GuidPrefix_t* and locator list (where they are listening) must match the one provided to the *clients*. *Clients* send discovery messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement. From then on, the *server* knows about the *client* and will inform it of the relevant discovery information. The same principle applies to a *server* connecting to another *server*.

Choosing between Client and Server

It is set by the *Discovery Protocol* general setting. A participant can only play one role (despite the fact that a *server* may connect to other *servers*). It is mandatory to fill this value because it defaults to *SIMPLE*. The examples below shows how to set this parameter both programmatically and using XML.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        DiscoveryProtocol_t::CLIENT;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        DiscoveryProtocol_t::SUPER_CLIENT;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        DiscoveryProtocol_t::SERVER;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        DiscoveryProtocol_t::BACKUP;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_discovery_protocol_alt" >
        <rtps>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>CLIENT</discoveryProtocol>
                    <!-- alternatives
                    <discoveryProtocol>SERVER</discoveryProtocol>
                    <discoveryProtocol>SUPER_CLIENT</discoveryProtocol>
                    <discoveryProtocol>BACKUP</discoveryProtocol>
                    -->
                    </discovery_config>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

The GuidPrefix as the server unique identifier

The *GuidPrefix_t* attribute belongs to the RTPS specification and univocally identifies each RTPSParticipant. It consists on 12 bytes, and in Fast DDS is a key for the DomainParticipant used in the DDS domain. Fast DDS defines the DomainParticipant *GuidPrefix_t* as a public data member of the *WireProtocolConfigQos* class. In the Discovery Server, it has the purpose to link a *server* to its *clients*. It must be specified in *server* and *client* setups.

Server side setup

The examples below show how to manage the corresponding enum data member and XML tag.

C++ - Option 1: Manual setting of the `unsigned char` in ASCII format.

```
eprosima::fastrtps::rtps::GuidPrefix_t serverGuidPrefix;
serverGuidPrefix.value[0] = eprosima::fastrtps::rtps::octet(0x44);
serverGuidPrefix.value[1] = eprosima::fastrtps::rtps::octet(0x53);
serverGuidPrefix.value[2] = eprosima::fastrtps::rtps::octet(0x00);
serverGuidPrefix.value[3] = eprosima::fastrtps::rtps::octet(0x5f);
serverGuidPrefix.value[4] = eprosima::fastrtps::rtps::octet(0x45);
serverGuidPrefix.value[5] = eprosima::fastrtps::rtps::octet(0x50);
serverGuidPrefix.value[6] = eprosima::fastrtps::rtps::octet(0x52);
serverGuidPrefix.value[7] = eprosima::fastrtps::rtps::octet(0x4f);
serverGuidPrefix.value[8] = eprosima::fastrtps::rtps::octet(0x53);
serverGuidPrefix.value[9] = eprosima::fastrtps::rtps::octet(0x49);
serverGuidPrefix.value[10] = eprosima::fastrtps::rtps::octet(0x4d);
serverGuidPrefix.value[11] = eprosima::fastrtps::rtps::octet(0x41);

DomainParticipantQos serverQos;
serverQos.wire_protocol().prefix = serverGuidPrefix;
```

C++ - Option 2: Using the >> operator and the `std::istringstream` type.

```
DomainParticipantQos serverQos;
std::istringstream("44.53.00.5f.45.50.52.4f.53.49.4d.41") >> serverQos.wire_protocol().
↪prefix;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_server_guidprefix" >
        <rtps>
            <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
        </rtps>
    </participant>
</profiles>
```

Note that a *server* can connect to other *servers*. Thus, the following section may also apply.

Important: When selecting a GUID prefix for the *server*, it is important to take into account that Fast DDS also uses this parameter to identify participants in the same process and enable intra-process communications. Setting two DomainParticipant GUID prefixes as intra-process compatible will result in no communication if the DomainParticipants run in separate processes. For more information, please refer to *GUID Prefix considerations for intra-process delivery*.

Warning: Launching more than one server using the same GUID prefix is undefined behavior.

Client side setup

Each *client* must keep a list of the *servers* to which it wants to link. Each single element represents an individual server, and a *GuidPrefix_t* must be provided. The *server* list must be populated with `RemoteServerAttributes` objects with a valid *GuidPrefix_t* data member. In XML the server list and its elements are simultaneously specified. Note that `prefix` is an element of the `RemoteServer` tag.

C++

```
RemoteServerAttributes server;
server.ReadguidPrefix("44.53.00.5f.45.50.52.4f.53.49.4d.41");

DomainParticipantQos clientQos;
clientQos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(server);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_discovery_client_prefix">
        <rtps>
            <builtin>
                <discovery_config>
                    <discoveryServersList>
                        <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
                            <!-- Metatraffic locators -->
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

The server locator list

Each *server* must specify valid locators where it can be reached. Any *client* must be given proper locators to reach each of its *servers*. As in the *above section*, here there is a *server* and a *client* side setup.

Server side setup

The examples below show how to setup the server locator list and XML tag.

C++

```
Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 1, 133);
locator.port = 64863;

DomainParticipantQos serverQos;
serverQos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_discovery_server_server_metatraffic
↪">
        <rtps>
            <builtin>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <!-- placeholder server UDP address -->
                            <address>192.168.1.113</address>
                            <port>64863</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

Note that a *server* can connect to other *servers*, thus, the following section may also apply.

Client side setup

Each *client* must keep a list of locators associated to the *servers* to which it wants to link. Each *server* specifies its own locator list which must be populated with `RemoteServerAttributes` objects with a valid `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList`. In XML the server list and its elements are simultaneously specified. Note the `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList` are elements of the `RemoteServer` tag.

C++

```
Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 1, 133);
locator.port = 64863;
RemoteServerAttributes server;
server.metatrafficUnicastLocatorList.push_back(locator);

DomainParticipantQos clientQos;
clientQos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(server);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_discovery_server_client_metatraffic
↪">
        <rtps>
            <builtin>
                <discovery_config>
                    <discoveryServersList>
                        <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <!-- placeholder server UDP address -->
                                        <address>192.168.1.113</address>
                                        <port>64863</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

Fine tuning discovery server handshake

As explained *above* the *clients* send discovery messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement. Mind that this period also applies for those *servers* which connect to other *servers*. The default value for this period is 450 ms.

C++

```
DomainParticipantQos participant_qos;
participant_qos.wire_protocol().builtin.discovery_config.discoveryServer_client_
↪syncperiod =
        Duration_t(0, 250000000);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_ping" >
        <rtps>
            <builtin>
                <discovery_config>
                    <clientAnnouncementPeriod>
                        <!-- change default to 250 ms -->
                        <nanosec>250000000</nanosec>
                    </clientAnnouncementPeriod>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

Modifying remote servers list at run time

Once a *server* or *client* is running, it is possible to programmatically modify the participant's list of remote *servers* to which the running *server* or *client* should connect. This is done by calling *DomainParticipant::set_qos()* with a *DomainParticipantQos* which has a modified *WireProtocolConfigQos* (see *WireProtocolConfigQos*). This feature allows to include a new remote server into the Discovery Server network or modify the remote server locator in case that the remote server is relaunched with a different listening locator.

Important: The list of remote *servers* can only be modified to either add more *servers*, or modify the remote server locator, but not to remove any of the existing ones. This means that the new list passed to *DomainParticipant::set_qos()* must be a superset of the existing one.

Note: The remote server list can also be modified using the `ROS_DISCOVERY_SERVER` environment variable. Please refer to *FASTDDS_ENVIRONMENT_FILE* for more information.

Warning: It is strongly advised to use either the API or the environment file. Using both at the same time may cause undefined behavior.

C++

```
// Get existing QoS for the server or client
DomainParticipantQos client_or_server_qos;
client_or_server->get_qos(client_or_server_qos);

/* Create a new server entry to which the client or server should connect */
RemoteServerAttributes remote_server_att;

// Set server's GUID prefix
remote_server_att.ReadguidPrefix("44.53.00.5f.45.50.52.4f.53.49.4d.42");

// Set server's listening locator for PDP
Locator_t locator;
IPLocator::setIPv4(locator, 127, 0, 0, 1);
locator.port = 11812;
remote_server_att.metatrafficUnicastLocatorList.push_back(locator);

/* Update list of remote servers for this client or server */
client_or_server_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_att);
if (ReturnCode_t::RETCODE_OK != client_or_server->set_qos(client_or_server_qos))
{
    // Error
    return;
}
```

Configure Discovery Server locators using names

All the examples provided in *Discovery Server Settings* use IPv4 addresses to specify the servers' listening locators. However, *Fast DDS* also allows to *specify locator addresses using names*.

Full example

The following constitutes a full example on how to configure *server* and *client* both programmatically and using XML. You may also have a look at the *eProsima Fast DDS* Github repository, which contains an example similar to the one discussed in this section, as well as multiple other examples for different use cases.

Server side setup

C++

```
// Get default participant QoS
DomainParticipantQos server_qos = PARTICIPANT_QOS_DEFAULT;

// Set participant as SERVER
server_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        DiscoveryProtocol_t::SERVER;

// Set SERVER's GUID prefix
std::istringstream("44.53.00.5f.45.50.52.4f.53.49.4d.41") >> server_qos.wire_
↪protocol().prefix;

// Set SERVER's listening locator for PDP
Locator_t locator;
IPLocator::setIPv4(locator, 127, 0, 0, 1);
locator.port = 11811;
server_qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);

/* Add a remote serve to which this server will connect */
// Set remote SERVER's GUID prefix
RemoteServerAttributes remote_server_att;
remote_server_att.ReadguidPrefix("44.53.01.5f.45.50.52.4f.53.49.4d.41");

// Set remote SERVER's listening locator for PDP
Locator_t remote_locator;
IPLocator::setIPv4(remote_locator, 127, 0, 0, 1);
remote_locator.port = 11812;
remote_server_att.metatrafficUnicastLocatorList.push_back(remote_locator);

// Add remote SERVER to SERVER's list of SERVERs
server_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_att);

// Create SERVER
DomainParticipant* server =
        DomainParticipantFactory::get_instance()->create_participant(0, server_qos);
if (nullptr == server)
{
    // Error
    return;
}
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_server_full_example">
        <rtps>
            <!-- Set SERVER's GUID prefix -->
            <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
            <builtin>
                <!-- Set participant as SERVER -->
                <discovery_config>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                    <!--
                        Set a list of remote servers to which this server connects.
```

Client side setup

C++

```
// Get default participant QoS
DomainParticipantQos client_qos = PARTICIPANT_QOS_DEFAULT;

// Set participant as CLIENT
client_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        DiscoveryProtocol_t::CLIENT;

// Set SERVER's GUID prefix
RemoteServerAttributes remote_server_att;
remote_server_att.ReadguidPrefix("44.53.00.5f.45.50.52.4f.53.49.4d.41");

// Set SERVER's listening locator for PDP
Locator_t locator;
IPLocator::setIPv4(locator, 127, 0, 0, 1);
locator.port = 11811;
remote_server_att.metatrafficUnicastLocatorList.push_back(locator);

// Add remote SERVER to CLIENT's list of SERVERs
client_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_att);

// Set ping period to 250 ms
client_qos.wire_protocol().builtin.discovery_config.discoveryServer_client_syncperiod =
        Duration_t(0, 250000000);

// Create CLIENT
DomainParticipant* client =
        DomainParticipantFactory::get_instance()->create_participant(0, client_qos);
if (nullptr == client)
{
    // Error
    return;
}
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_client_full_example">
        <rtps>
            <builtin>
                <discovery_config>
                    <!-- Set participant as CLIENT -->
                    <discoveryProtocol>CLIENT</discoveryProtocol>
                    <!--
                        Set list of remote servers. This list may contain one or
                        more <RemoteServer> tags
                    -->
                    <discoveryServersList>
                        <!--
                            Set remote server configuration:
                                - Prefix
                                - PDP listening locator
                        -->
                        <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
```

Security

Configuring *Security* on *servers* and *clients* is done the same way as for any other participant. This section depicts the limitations imposed by the security enforcement on the communication between *clients* and *servers*, and which discovery information is propagated by a *server* depending on the security configuration of the *clients* and *servers* to which it is connected.

It is important to note that for enabling a secure discovery when using Discovery Server, *Fast DDS* must be compiled with security support (see *CMake options*), and the *Domain Governance Document* must explicitly encrypt the discovery.

As in SDP, when using this feature, the Domain Governance Document of all *clients* and *servers* connecting to a *server* must match that of the *server*, which implies that all *DomainParticipants* belonging to the same Discovery Sever network must configure the discovery protection in the same manner.

Although the *server* mediates the discovery process and creates connections between *clients*, the *clients* themselves still go through the PKI (Public Key Infrastructure) exchange in order to have a secure communication between them.

Important: In order to keep the behavior consistent with the QoS Policies, the *server* does not check the *DomainParticipant Permissions Document* of the *DomainParticipants* that it is connecting.

Important: Security support for Discovery Server is only supported from Fast DDS v2.10.0 onward.

DomainParticipantListener Discovery Callbacks

As stated in *DomainParticipantListener*, the `DomainParticipantListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the DomainParticipant. Fast DDS defines four callbacks attached to events that may occur during discovery: `on_participant_discovery()`, `on_subscriber_discovery()`, `on_publisher_discovery()`, `on_type_discovery()`. Further information about the DomainParticipantListener is provided in the *DomainParticipantListener* section. The following is an example of the implementation of DomainParticipantListener discovery callbacks.

```
class DiscoveryDomainParticipantListener : public DomainParticipantListener
{
    /* Custom Callback on_participant_discovery */
    void on_participant_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::ParticipantDiscoveryInfo&& info) override
    {
        static_cast<void>(participant);
        switch (info.status){
            case eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOVERED_
↪PARTICIPANT:
                /* Process the case when a new DomainParticipant was found in the domain␣
↪*/
                std::cout << "New DomainParticipant '" << info.info.m_participantName <<
                    "' with ID '" << info.info.m_guid.entityId << "' and GuidPrefix '" <<
                    info.info.m_guid.guidPrefix << "' discovered." << std::endl;
                break;
            case eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::CHANGED_QOS_
↪PARTICIPANT:
```

(continues on next page)

(continued from previous page)

```
                /* Process the case when a DomainParticipant changed its QOS */
                break;
            case eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::REMOVED_PARTICIPANT:
                /* Process the case when a DomainParticipant was removed from the domain␣
↪*/
                std::cout << "New DomainParticipant '" << info.info.m_participantName <<
                    "' with ID '" << info.info.m_guid.entityId << "' and GuidPrefix '" <<
                    info.info.m_guid.guidPrefix << "' left the domain." << std::endl;
                break;
        }
    }

    /* Custom Callback on_subscriber_discovery */
    void on_subscriber_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info) override
    {
        static_cast<void>(participant);
        switch (info.status){
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERED_READER:
                /* Process the case when a new subscriber was found in the domain */
                std::cout << "New DataReader subscribed to topic '" << info.info.
↪topicName() <<
                    "' of type '" << info.info.typeName() << "' discovered";
                break;
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::CHANGED_QOS_READER:
                /* Process the case when a subscriber changed its QOS */
                break;
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::REMOVED_READER:
                /* Process the case when a subscriber was removed from the domain */
                std::cout << "New DataReader subscribed to topic '" << info.info.
↪topicName() <<
                    "' of type '" << info.info.typeName() << "' left the domain.";
                break;
        }
    }

    /* Custom Callback on_publisher_discovery */
    void on_publisher_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info) override
    {
        static_cast<void>(participant);
        switch (info.status){
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERED_WRITER:
                /* Process the case when a new publisher was found in the domain */
                std::cout << "New DataWriter publishing under topic '" << info.info.
↪topicName() <<
                    "' of type '" << info.info.typeName() << "' discovered";
                break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::CHANGED_QOS_WRITER:
                /* Process the case when a publisher changed its QOS */
```

(continues on next page)

(continued from previous page)

```
                break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::REMOVED_WRITER:
                /* Process the case when a publisher was removed from the domain */
                std::cout << "New DataWriter publishing under topic '" << info.info.
↪topicName() <<
                    "' of type '" << info.info.typeName() << "' left the domain.";
                break;
        }
    }

    /* Custom Callback on_type_discovery */
    void on_type_discovery(
            DomainParticipant* participant,
            const eprosima::fastrtps::rtps::SampleIdentity& request_sample_id,
            const eprosima::fastrtps::string_255& topic,
            const eprosima::fastrtps::types::TypeIdentifier* identifier,
            const eprosima::fastrtps::types::TypeObject* object,
            eprosima::fastrtps::types::DynamicType_ptr dyn_type) override
    {
        static_cast<void>(participant);
        static_cast<void>(request_sample_id);
        static_cast<void>(topic);
        static_cast<void>(identifier);
        static_cast<void>(object);
        static_cast<void>(dyn_type);
        std::cout << "New data type of topic '" << topic << "' discovered." << std::endl;
    }

};
```

To use the previously implemented discovery callbacks in `DiscoveryDomainParticipantListener` class, which inherits from the DomainParticipantListener, an object of this class is created and registered as a listener of the DomainParticipant.

```
// Create the participant QoS and configure values
DomainParticipantQos pqos;

// Create a custom user DomainParticipantListener
DiscoveryDomainParticipantListener* plistener = new DiscoveryDomainParticipantListener();
// Pass the listener on DomainParticipant creation.
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(
    0, pqos, plistener);
```

Important: Read more about callbacks and its hierarchy *here*

6.20 Transport Layer

The transport layer provides communication services between DDS entities, being responsible of actually sending and receiving messages over a physical transport. The DDS layer uses this service for both user data and discovery traffic communication. However, the DDS layer itself is transport independent, it defines a transport API and can run over any transport plugin that implements this API. This way, it is not restricted to a specific transport, and applications can choose the one that best suits their requirements, or create their own.

eProsima Fast DDS comes with five transports already implemented:

- **UDPv4**: UDP Datagram communication over IPv4. This transport is created by default on a new *DomainParticipant* if no specific transport configuration is given (see *UDP Transport*).
- **UDPv6**: UDP Datagram communication over IPv6 (see *UDP Transport*).
- **TCPv4**: TCP communication over IPv4 (see *TCP Transport*).
- **TCPv6**: TCP communication over IPv6 (see *TCP Transport*).
- **SHM**: Shared memory communication among entities running on the same host. This transport is created by default on a new *DomainParticipant* if no specific transport configuration is given (see *Shared Memory Transport*).

Although it is not part of the transport module, *intraprocess data delivery* and *data sharing delivery* are also available to send messages between entities on some settings. The figure below shows a comparison between the different transports available in *Fast DDS*.

6.20.1 Transport API

The following diagram presents the classes defined on the transport API of *eProsima Fast DDS*. It shows the abstract API interfaces, and the classes required to implement a transport.

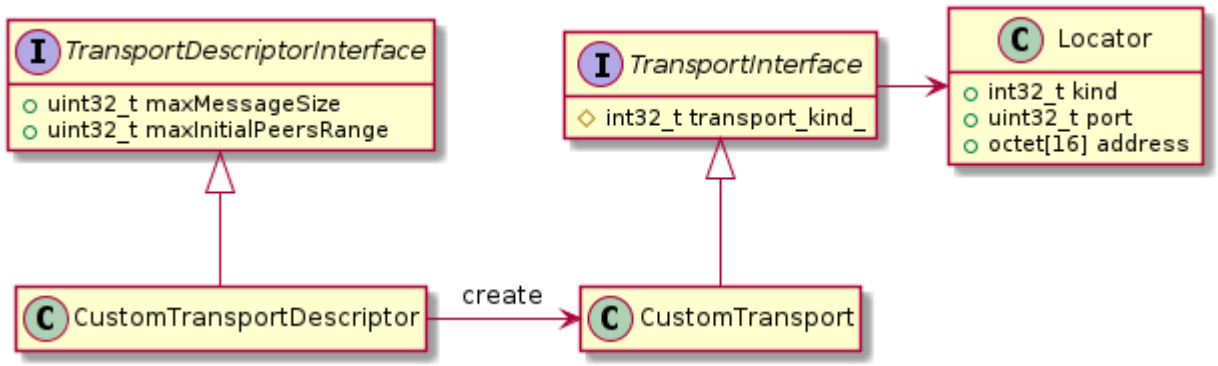


Fig. 10: Transport API diagram

- *TransportDescriptorInterface*
- *TransportInterface*
- *Locator*
- *Chaining of transports*

TransportDescriptorInterface

Any class that implements the *TransportDescriptorInterface* is known as a `TransportDescriptor`. It acts as a *builder* for a given transport, meaning that is allows to configure the transport, and then a new *Transport* can be built according to this configuration using its *create_transport* factory member function.

Data members

The TransportDescriptorInterface defines the following data members:

Member	Data type	Description
maxMessageSize	`uint32_t`	Maximum size of a single message in the transport.
maxInitialPeersRange	`uint32_t`	Number of channels opened with each initial remote peer.

Any implementation of *TransportDescriptorInterface* should add as many data members as required to full configure the transport it describes.

TransportInterface

A `Transport` is any class that implements the *TransportInterface*. It is the object that actually performs the message distribution over a physical transport.

Each `Transport` class defines its own *kind*, a unique identifier that is used to check the compatibility of a *Locator* with a Transport, i.e., determine whether a Locator refers to a Transport or not.

Applications do not create the `Transport` instance themselves. Instead, applications use a `TransportDescriptor` instance to configure the desired transport, and add this configured instance to the list of user-defined transports of the *DomainParticipant*. The DomainParticipant will use the factory function on the `TransportDescriptor` to create the `Transport` when required.

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto udp_transport = std::make_shared<UDPv4TransportDescriptor>();
udp_transport->sendBufferSize = 9216;
udp_transport->receiveBufferSize = 9216;
udp_transport->non_blocking_send = true;

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(udp_transport);

// Avoid using the default transport
qos.transport().use_builtin_transports = false;
```

Data members

The TransportInterface defines the following data members:

Member	Data type	Description
`transport_kind_`	`int32_t`	Unique identifier of the transport type.

Note: `transport_kind_` is a protected data member for internal use. It cannot be accessed nor modified from the public API. However, users that are implementing a custom Transport need to fill it with a unique constant value in the new implementation.

Currently the following identifiers are used in *Fast DDS*:

Identifier	Value	Transport type
LOCATOR_KIND_RESERVED	0	None. Reserved value for internal use.
LOCATOR_KIND_UDPv4	1	*UDP Transport* over IPv4.
LOCATOR_KIND_UDPv6	2	*UDP Transport* over IPv6.
LOCATOR_KIND_TCPv4	4	*TCP Transport* over IPv4.
LOCATOR_KIND_TCPv6	8	*TCP Transport* over IPv6.
LOCATOR_KIND_SHM	16	*Shared Memory Transport*.

Locator

A `Locator_t` uniquely identifies a communication channel with a remote peer for a particular transport. For example, on UDP transports, the Locator will contain the information of the IP address and port of the remote peer.

The Locator class is not abstract, and no specializations are implemented for each transport type. Instead, transports should map the data members of the Locator class to their own channel identification concepts. For example, on *Shared Memory Transport* the `address` contains a unique ID for the local host, and the `port` represents the shared ring buffer used to communicate buffer descriptors.

Please refer to *Listening Locators* for more information about how to configure DomainParticipant to listen to incoming traffic.

Data members

The Locator defines the following data members:

Member	Data type	Description
kind	`int32_t`	Unique identifier of the transport type.
port	`uint32_t`	The channel *port*.
address	`octet[16]`	The channel *address*.

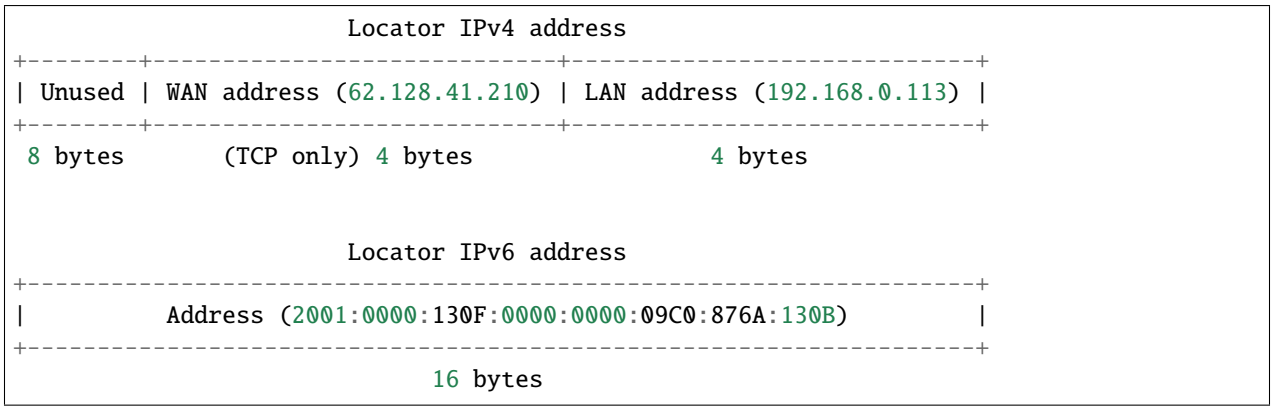
In TCP, the port of the locator is divided into a physical and a logical port.

- The *physical port* is the port used by the network device, the real port that the operating system understands. It is stored in the two least significant bytes of the member *port*.
- The *logical port* is the RTPS port. It is used by the RTPS protocol to distinguish different entities. It is stored in the two most significant bytes of the member *port*.

In TCP, this distinction allows for several DDS applications using different RTPS ports (*logical ports*) to share the same *physical port*, thus only requiring for a single port to be opened for all communications. In UDP there is only the *physical port*, which is also the RTPS port, and is stored in the two least significant bytes of the member `port`.

The locator address, represented in 16 bytes, is managed differently depending on whether the protocol used is IPv4 or IPv6.

- The IPv6 address uses the 16 available bytes to represent a unique and global address.
- The IPv4 address splits those 16 bytes in the following three sections, ordered from least to greatest significance:
  - 4 bytes LAN IP: Local subnet identification (UDP and TCP).
  - 4 bytes WAN IP: Public IP (TCP only).
  - 8 bytes unused.



Check how to manipulate the WAN address in the *TCP IPv4 transport descriptor api* section.

Configuring IP locators with IPLocator

`IPLocator` is an auxiliary static class that offers methods to manipulate IP based locators. It is convenient when setting up a new *UDP Transport* or *TCP Transport*, as it simplifies setting IPv4 and IPv6 addresses, or manipulating ports.

For example, normally users configure the physical port and do not need to worry about logical ports. However, `IPLocator` allows to manage them if needed.

```
// We will configure a TCP locator with IPLocator
Locator_t locator;

// Get & set the physical port
uint16_t physical_port = IPLocator::getPhysicalPort(locator);
IPLocator::setPhysicalPort(locator, 5555);

// On TCP locators, we can get & set the logical port
uint16_t logical_port = IPLocator::getLogicalPort(locator);
IPLocator::setLogicalPort(locator, 7400);

// Set WAN address
IPLocator::setWan(locator, "80.88.75.55");
```

Fast DDS also allows to specify locator addresses using names. When an address is specified by a name, *Fast DDS* will query the known hosts and available DNS servers to try to resolve the IP address. This address will in turn be used

to create the listening locator in the case of *server*, or as the address of the remote *server* in the case of *clients* (and *servers* that connect to other *servers*).

C++

```
Locator_t locator;
auto response = eprosima::fastrtps::rtps::IPLocator::resolveNameDNS("localhost");
// Get the first returned IPv4
if (response.first.size() > 0)
{
    IPLocator::setIPv4(locator, response.first.begin()->data());
    locator.port = 11811;
}
// Use the locator to create server or client
```

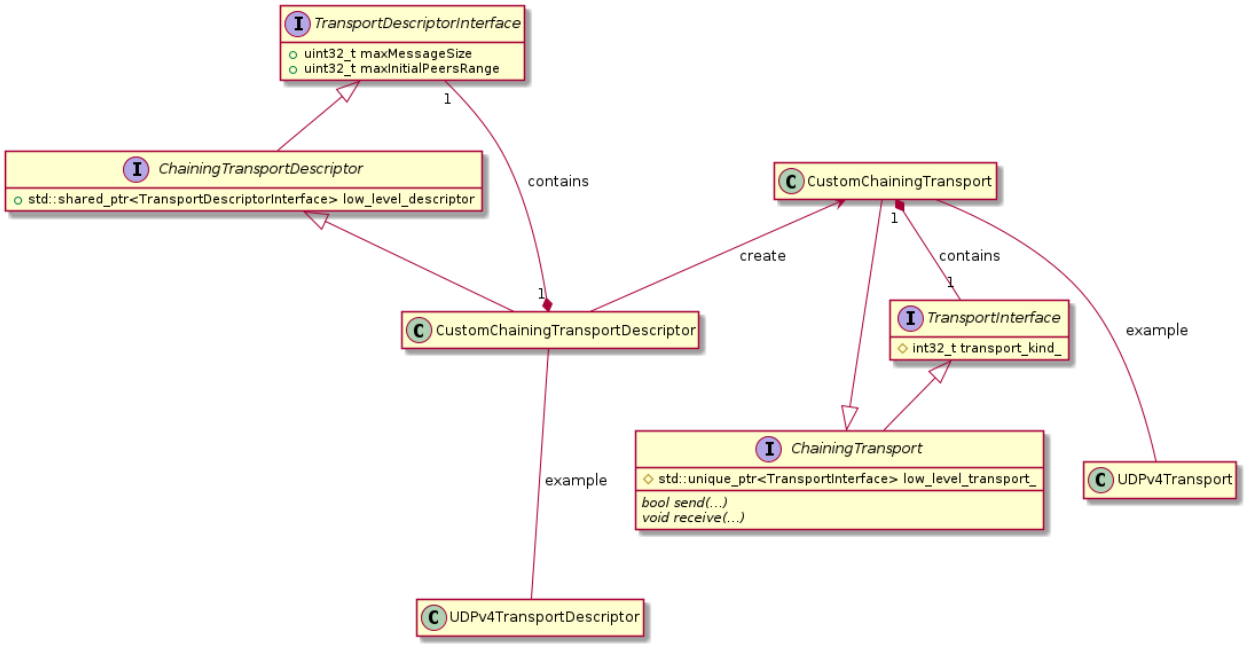
XML

```
<locator>
    <udpv4>
        <port>11811</port>
        <address>localhost</address>
    </udpv4>
</locator>
```

Warning: Currently, XML only supports loading IP addresses by name for UDP transport.

Chaining of transports

There are use cases where the user needs to pre-process out-coming information before being sent to network and also the incoming information after being received. *Transport API* offers two interfaces for implementing this kind of functionality: *ChainingTransportDescriptor* and *ChainingTransport*.



These extensions allow to implement a new Transport which depends on another one (called here as `low_level_transport_`). The user can override the *send()* function, pre-processing the out-coming buffer before calling the associated `low_level_transport_`. Also, when a incoming buffer arrives to the `low_level_transport_`, this one calls the overridden *receive()* function to allow to pre-process the buffer.

ChainingTransportDescriptor

Implementing *ChainingTransportDescriptor* allows to configure the new Transport and set the `low_level_transport_` on which it depends. The associated `low_level_transport_` can be any transport which inherits from *TransportInterface* (including another *ChainingTransport*).

The *ChainingTransportDescriptor* defines the following data members:

Member	Data type	Description
low_level_descriptor	std::shared_ptr<TransportDescriptorInterface>	Transport descriptor of the low_level_transport_.

User has to specify the `low_level_transport_` in the definition of its new custom transport.

```
        DomainParticipantQos qos;

        auto udp_transport = std::make_shared<UDPv4TransportDescriptor>();

        // Create a descriptor for the new transport.
        // The low level transport will be a UDPv4Transport.
        auto custom_transport = std::make_shared<CustomChainingTransportDescriptor>(udp_
↪transport);

        // Link the Transport Layer to the Participant.
        qos.transport().user_transports.push_back(custom_transport);

        // Avoid using the default transport
        qos.transport().use_builtin_transports = false;
```

ChainingTransport

This interface forces the user to implement *send()* and *receive()* functions. The idea is to pre-process the buffer and after, call to the next level.

```
class CustomChainingTransport : public eprosima::fastdds::rtps::ChainingTransport
{

public:

    CustomChainingTransport(
            const CustomChainingTransportDescriptor& descriptor)
        : ChainingTransport(descriptor)
        , descriptor_(descriptor)
    {
    }
```

(continues on next page)

(continued from previous page)

```
    eprosima::fastdds::rtps::TransportDescriptorInterface* get_configuration()
    {
        return &descriptor_;
    }

    bool send(
            eprosima::fastrtps::rtps::SenderResource* low_sender_resource,
            const eprosima::fastrtps::rtps::octet* send_buffer,
            uint32_t send_buffer_size,
            eprosima::fastrtps::rtps::LocatorsIterator* destination_locators_begin,
            eprosima::fastrtps::rtps::LocatorsIterator* destination_locators_end,
            const std::chrono::steady_clock::time_point& timeout) override
    {
        //
        // Preprocess outcoming buffer.
        //

        // Call low level transport
        return low_sender_resource->send(send_buffer, send_buffer_size, destination_
↪locators_begin,
                        destination_locators_end, timeout);
    }

    void receive(
            eprosima::fastdds::rtps::TransportReceiverInterface* next_receiver,
            const eprosima::fastrtps::rtps::octet* receive_buffer,
            uint32_t receive_buffer_size,
            const eprosima::fastrtps::rtps::Locator_t& local_locator,
            const eprosima::fastrtps::rtps::Locator_t& remote_locator) override
    {
        //
        // Preprocess incoming buffer.
        //

        // Call upper level
        next_receiver->OnDataReceived(receive_buffer, receive_buffer_size, local_locator,
↪ remote_locator);
    }

private:

    CustomChainingTransportDescriptor descriptor_;
};
```

6.20.2 UDP Transport

UDP is a connectionless transport, where the receiving *DomainParticipant* must open a UDP port listening for incoming messages, and the sending DomainParticipant sends messages to this port.

Warning: This documentation assumes the reader has basic knowledge of UDP/IP concepts, since terms like Time To Live (TTL), socket buffers, and port numbering are not explained in detail. However, it is possible to configure a basic UDP transport on *Fast DDS* without this knowledge.

UDPTransportDescriptor

eProsima Fast DDS implements UDP transport for both UDPv4 and UDPv6. Each of these transports is independent from the other, and has its own `TransportDescriptorInterface`. However, all their `TransportDescriptorInterface` data members are common.

The following table describes the common data members for both UDPv4 and UDPv6.

Member	Data type	Default	Description
`sendBufferSize`	`uint32_t`	0	Size of the sending buffer of the socket (octets).
`receiveBufferSize`	`uint32_t`	0	Size of the receiving buffer of the socket (octets).
`interfaceWhiteList`	`vector<string>`	Empty vector	List of allowed interfaces. See *Interface Whitelist*.
`TTL`	`uint8_t`	1	Time to live, in number of hops.
`m_output_udp_socket`	`uint16_t`	0	Port number for the outgoing messages.
`non_blocking_send`	`bool`	`false`	Do not block on send operations (*).

Note: When `non_blocking_send` is set to true, send operations will return immediately if the buffer is full, but no error will be returned to the upper layer. This means that the application will behave as if the datagram is sent and lost. This value is specially useful on high-frequency best-effort writers.

When set to `false`, send operations will block until the network buffer has space for the datagram. This may hinder performance on high-frequency writers.

UDPv4TransportDescriptor

`UDPv4TransportDescriptor` has no additional data members from the common ones described in *UDPTransport-Descriptor*.

Note: The `kind` value for a `UDPv4TransportDescriptor` is given by the value `LOCATOR_KIND_UDPv4`.

UDPv6TransportDescriptor

UDPv6TransportDescriptor has no additional data members from the common ones described in *UDPTransportDescriptor*.

Note: The *kind* value for a *UDPv6TransportDescriptor* is given by the value *LOCATOR_KIND_UDPv6*.

Enabling UDP Transport

Fast DDS enables a UDPv4 transport by default. Nevertheless, the application can enable other UDP transports if needed. To enable a new UDP transport in a *DomainParticipant*, first create an instance of *UDPv4TransportDescriptor* (for UDPv4) or *UDPv6TransportDescriptor* (for UDPv6), and add it to the user transport list of the *DomainParticipant*.

The examples below show this procedure in both C++ code and XML file.

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto udp_transport = std::make_shared<UDPv4TransportDescriptor>();
udp_transport->sendBufferSize = 9216;
udp_transport->receiveBufferSize = 9216;
udp_transport->non_blocking_send = true;

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(udp_transport);

// Avoid using the default transport
qos.transport().use_builtin_transports = false;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>udp_transport</transport_id>
            <type>UDPv4</type>
            <sendBufferSize>9216</sendBufferSize>
            <receiveBufferSize>9216</receiveBufferSize>
            <non_blocking_send>true</non_blocking_send>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="UDPParticipant">
        <rtps>
            <userTransports>
                <transport_id>udp_transport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
        </rtps>
```

(continues on next page)

(continued from previous page)

```
    </participant>
</profiles>
```

6.20.3 TCP Transport

TCP is a connection oriented transport, so the *DomainParticipant* must establish a TCP connection to the remote peer before sending data messages. Therefore, one of the communicating DomainParticipants (the one acting as *server*) must open a TCP port listening for incoming connections, and the other one (the one acting as *client*) must connect to this port.

Note: The *server* and *client* concepts are independent from the DDS concepts of *Publisher*, *Subscriber*, *DataWriter*, and *DataReader*. Also, these concepts are independent from the *eProsima Discovery Server servers* and *clients* (*Discovery Server Settings*). Any of them can act as a *TCP Server* or *TCP Client* when establishing the connection, and the DDS communication will work over this connection.

Warning: This documentation assumes the reader has basic knowledge of TCP/IP concepts, since terms like Time To Live (TTL), Cyclic Redundancy Check (CRC), Transport Layer Security (TLS), socket buffers, and port numbering are not explained in detail. However, it is possible to configure a basic TCP transport on *Fast DDS* without this knowledge.

TCPTransportDescriptor

eProsima Fast DDS implements TCP transport for both TCPv4 and TCPv6. Each of these transports is independent from the other, and has its own `TransportDescriptorInterface`. However, they share many of their features, and most of the `TransportDescriptorInterface` data members are common.

The following table describes the common data members for both TCPv4 and TCPv6.

Member	Data type	Default	Description
sendBufferSize	uint32_t	0	Size of the sending buffer of the socket (octets).
receiveBufferSize	uint32_t	0	Size of the receiving buffer of the socket (octets).
interfaceWhiteList	vector<string>	Empty vector	List of allowed interfaces See *Interface Whitelist*.
TTL	uint8_t	1	Time to live, in number of hops.
listening_ports	vector<uint16_t>	Empty vector	List of ports to listen as *server*. If a port is set to 0, an available port will be automatically assigned.
keep_alive_frequency	uint32_t	5000	Frequency of RTCP keep alive requests (in ms).
keep_alive_timeout	uint32_t	15000	Time since sending the last keep alive request to consider a connection as broken (in ms).
max_logical_port	uint16_t	100	Maximum number of logical ports to try during RTCP negotiation.
logical_port_range	uint16_t	20	Maximum number of logical ports per request to try during RTCP negotiation.
logical_port_increment	uint16_t	2	Increment between logical ports to try during RTCP negotiation.
enable_tcp_nodelay	bool	false	Enables the TCP_NODELAY socket option.
calculate_crc	bool	true	True to calculate and send CRC on message headers.
check_crc	bool	true	True to check the CRC of incoming message headers.
apply_security	bool	false	True to use TLS. See *TLS over TCP*.
tls_config	*TLSConfig*		Configuration for TLS. See *TLS over TCP*.

Note: If *listening_ports* is left empty, the participant will not be able to receive incoming connections but will be able to connect to other participants that have configured their listening ports.

TCPv4TransportDescriptor

The following table describes the data members that are exclusive for *TCPv4TransportDescriptor*.

Member	Data type	Default	Description
wan_addr	octet[4]	[0, 0, 0, 0]	Configuration for WAN. See *WAN or Internet Communication over TCPv4*.

Note: The *kind* value for a *TCPv4TransportDescriptor* is given by the value *LOCATOR_KIND_TCPv4*.

TCPv6TransportDescriptor

TCPv6TransportDescriptor has no additional data members from the common ones described in *TCPTransportDescriptor*.

Note: The *kind* value for a *TCPv6TransportDescriptor* is given by the value *LOCATOR_KIND_TCPv6*.

Enabling TCP Transport

There are several ways of enabling TCP transport in *eprosima Fast DDS*. According to the facet of each scenario, one method might suit better than the others.

Configuration of Builtin Transports

The first option is to modify the builtin transports that are responsible of the creation of the DomainParticipant transports. The existing configuration that enables TCP transports is `LARGE_DATA`. This option instantiates a UDPv4, a TCPv4 and a SHM transport, respectively. UDP protocol will be used for multicast announcements during the participant discovery phase (see *Discovery phases*) while the participant liveliness and the application data delivery occurs over TCP or SHM. This configuration enables auto discovery and does not require to manually set up each participant IP and listening port. Hence, avoiding the typical Server-Client configuration.

Builtin Transports can be configured using the `FASTDDS_BUILTIN_TRANSPORTS` environment variable (see *FASTDDS_BUILTIN_TRANSPORTS*), XML profiles (see *RTPS element type*) or via code.

Environment Variable

```
export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <!--
            UDP transport for PDP and SHM/TCPv4 transport for both EDP and application␣
↪data
        -->
        <participant profile_name="large_data_builtin_transports" is_default_profile=
↪"true">
            <rtps>
                <builtinTransports>LARGE_DATA</builtinTransports>
            </rtps>
        </participant>
    </profiles>
</dds>
```

C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;
qos.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_DATA);
```

Note: Note that `LARGE_DATA` configuration of the builtin transports will also create a SHM transport along the UDP and TCP transports. Shared Memory will be used whenever it is possible. Manual configuration will be required if a TCP communication is required when SHM is feasible. (See *Large Data Mode*).

Server-Client Configuration

To set up a Server-Client configuration you need to create an instance of *TCPv4TransportDescriptor* (for TCPv4) or *TCPv6TransportDescriptor* (for TCPv6), and add it to the user transport list of the DomainParticipant.

Depending on the TCP transport descriptor settings and network locators defined, the DomainParticipant can act as a TCP Server or TCP Client.

- **TCP Server**: If you provide `listening_ports` on the descriptor, the DomainParticipant will act as *TCP server*, listening for incoming remote connections on the given ports. The examples below show this procedure in both C++ code and XML file.

  C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared
↪<eprosima::fastdds::rtps::TCPv4TransportDescriptor>();
tcp_transport->add_listener_port(5100);

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tcp_transport);

// Avoid using the default transport
qos.transport().use_builtin_transports = false;

// [OPTIONAL] Set unicast locators
eprosima::fastrtps::rtps::Locator_t locator;
locator.kind = LOCATOR_KIND_TCPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, "192.168.1.10");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(locator, 5100);
// [OPTIONAL] Logical port default value is 0, automatically assigned.
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(locator, 5100);

qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
qos.wire_protocol().default_unicast_locator_list.push_back(locator);
```

  XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tcp_server_transport</transport_id>
            <type>TCPv4</type>
            <listening_ports>
                <port>5100</port>
            </listening_ports>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="tcp_server_participant">
        <rtps>
            <userTransports>
```

(continues on next page)

(continued from previous page)

```
                <transport_id>tcp_server_transport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
            <!-- Optional unicast locator set -->
            <defaultUnicastLocatorList>
                <locator>
                    <tcpv4>
                        <address>192.168.1.10</address>
                        <physical_port>5100</physical_port>
                        <!-- Optional logical port set -->
                        <port>5100</port>
                    </tcpv4>
                </locator>
            </defaultUnicastLocatorList>
            <!-- Optional metatraffic unicast locator set -->
            <builtin>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <tcpv4>
                            <address>192.168.1.10</address>
                            <physical_port>5100</physical_port>
                            <!-- Optional logical port set -->
                            <port>5100</port>
                        </tcpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

- **TCP Client**: If you provide `initialPeersList` to the DomainParticipant, it will act as *TCP client*, trying to connect to the remote *servers* at the given addresses and ports. The examples below show this procedure in both C++ code and XML file. See *Initial peers* for more information about their configuration.

  C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;

// Disable the built-in Transport Layer.
qos.transport().use_builtin_transports = false;

// Create a descriptor for the new transport.
// Do not configure any listener port
auto tcp_transport = std::make_shared
↪<eprosima::fastdds::rtps::TCPv4TransportDescriptor>();
qos.transport().user_transports.push_back(tcp_transport);

// Set initial peers.
eprosima::fastrtps::rtps::Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(initial_peer_locator, "192.168.1.10");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(initial_peer_locator, 5100);
```

(continues on next page)

(continued from previous page)

```
// If the logical port is set in the server side, it must be also set here with the␣
↪same value.
// If not set in the server side in a unicast locator, do not set it here.
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(initial_peer_locator, 5100);

qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer_locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tcp_client_transport</transport_id>
            <type>TCPv4</type>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="tcp_client_participant">
        <rtps>
            <userTransports>
                <transport_id>tcp_client_transport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
            <builtin>
                <initialPeersList>
                    <locator>
                        <tcpv4>
                            <address>192.168.1.10</address>
                            <physical_port>5100</physical_port>
                            <!-- To be set if set in server -->
                            <port>5100</port>
                        </tcpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

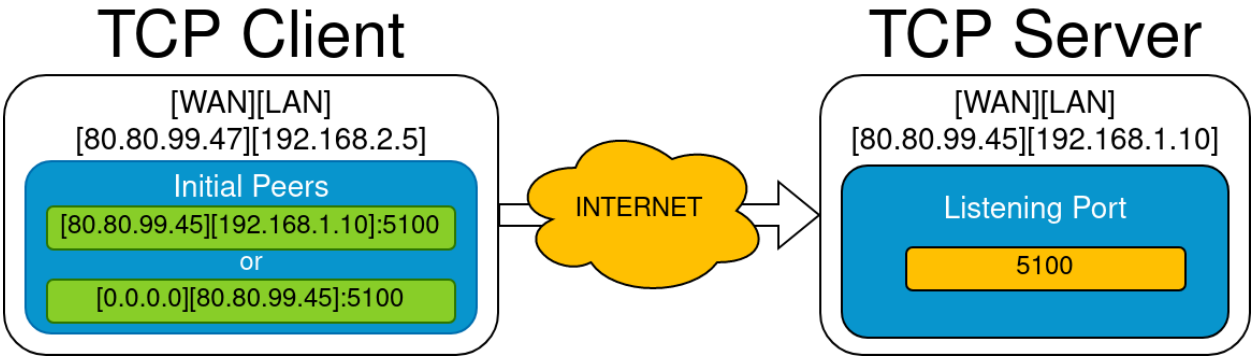
Note: Manually setting unicast locators is optional. If not setting them or setting them with a logical port of `0`, the client's initial peer shouldn't set its logical port (or set it to `0`). Otherwise, initial peer's logical port must match server's unicast logical port.

HelloWorldExampleTCP shows how to use and configure a TCP transport.

WAN or Internet Communication over TCPv4

Fast DDS is able to connect through the Internet or other WAN networks when configured properly. To achieve this kind of scenarios, the involved network devices such as routers and firewalls must add the rules to allow the communication.

For example, imagine we have the scenario represented on the following figure:



- A DomainParticipant acts as a *TCP server* listening on port `5100` and is connected to the WAN through a router with public IP `80.80.99.45`.
- Another DomainParticipant acts as a *TCP client* and has configured the server's IP address and port in its *Initial peers* list.

By using `set_WAN_address(wan_ip)`, the WAN IP address is set on the participant's locators that are communicated during the discovery phase.

Like in the LAN case, manually setting unicast locators is optional. However, in this case, there are some considerations to take into account when setting its IP addresses:

- Setting the WAN IP address using the `setWAN()` method in unicast locators is ineffective because it gets overridden by the `set_WAN_address()` call.
- For assigning IP addresses to unicast locators, use only the `setIPv4()` or `setIPv6()` methods, which are LAN IP setters. There are some configurations which allow using these setters with a WAN IP address.

Depending on whether the server has manually set its metatraffic unicast locators and default unicast locators, the client needs to adjust its initial peer list accordingly:

- If the server's unicast locators are configured with the LAN IP address:
  - The initial peer can be set up with only the server's WAN IP using the LAN IP setter.
  - Alternatively, it can be configured with both the server's LAN and WAN IP addresses using the LAN setter for the LAN IP and the WAN setter for the WAN IP.
- If the server's unicast locators are configured with the WAN IP address:
  - The initial peer must be set up with only the server's WAN IP using the LAN setter.
  - Alternatively, it can be configured with the WAN IP address using both the LAN and WAN setters.
- If the server has not set any unicast locators:
  - The initial peer can be configured with only the server's WAN IP using the LAN setter.
  - Alternatively, it can be configured with both the server's LAN and WAN IP addresses using the LAN setter for the LAN IP and the WAN setter for the WAN IP.

Note: Manually setting unicast locators is optional. If not setting them or setting them with a logical port of `0`, the client's initial peer shouldn't set its logical port (or set it to `0`). Otherwise, initial peer's logical port must match server's unicast logical port.

On the server side, the router must be configured to forward to the *TCP server* all traffic incoming to port `5100`. Typically, a NAT routing of port `5100` to our machine is enough. Any existing firewall should be configured as well.

In addition, to allow incoming connections through a WAN, the *TCPv4TransportDescriptor* must indicate its **public** IP address in the *wan_addr* data member. The following examples show how to configure the DomainParticipant both in C++ and XML.

C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↪();
tcp_transport->add_listener_port(5100);
tcp_transport->set_WAN_address("80.80.99.45");

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tcp_transport);

// Avoid using the default transport
qos.transport().use_builtin_transports = false;

// [OPTIONAL] Set unicast locators (do not use setWAN(), set_WAN_address() overwrites it)
eprosima::fastrtps::rtps::Locator_t locator;
locator.kind = LOCATOR_KIND_TCPv4;
// [RECOMMENDED] Use the LAN address of the server
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, "192.168.1.10");
// [ALTERNATIVE] Use server's WAN address. In that case, initial peers must be configured
// only with server's WAN address.
// eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, "80.80.99.45");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(locator, 5100);
// [OPTIONAL] Logical port default value is 0, automatically assigned.
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(locator, 5100);

qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
qos.wire_protocol().default_unicast_locator_list.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
-->
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tcp_server_wan_transport</transport_id>
            <type>TCPv4</type>
            <listening_ports>
                <port>5100</port>
```

(continues on next page)

(continued from previous page)

```
            </listening_ports>
            <wan_addr>80.80.99.45</wan_addr>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="tcp_server_wan_participant">
        <rtps>
            <userTransports>
                <transport_id>tcp_server_wan_transport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
            <!-- Optional unicast locator set -->
            <defaultUnicastLocatorList>
                <locator>
                    <tcpv4>
                        <address>192.168.1.10</address>
                        <!-- Alternatively use WAN address -->
                        <!-- <address>80.80.99.45</address> -->
                        <physical_port>5100</physical_port>
                        <!-- Optional logical port set -->
                        <port>5100</port>
                    </tcpv4>
                </locator>
            </defaultUnicastLocatorList>
            <!-- Optional metatraffic unicast locator set -->
            <builtin>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <tcpv4>
                            <address>192.168.1.10</address>
                            <!-- Alternatively use WAN address -->
                            <!-- <address>80.80.99.45</address> -->
                            <physical_port>5100</physical_port>
                            <!-- Optional logical port set -->
                            <port>5100</port>
                        </tcpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

On the client side, the DomainParticipant must be configured with the **public** IP address and `listening_ports` of the *TCP server* as *Initial peers*.

C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;

// Disable the built-in Transport Layer.
qos.transport().use_builtin_transports = false;
```

(continues on next page)

(continued from previous page)

```
// Create a descriptor for the new transport.
// Do not configure any listener port
auto tcp_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↪();
// [RECOMMENDED] Use client's WAN address if there are more clients in other local␣
↪networks.
tcp_transport->set_WAN_address("80.80.99.47");
qos.transport().user_transports.push_back(tcp_transport);

// Set initial peers.
eprosima::fastrtps::rtps::Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
// [RECOMMENDED] Use both WAN and LAN server addresses
eprosima::fastrtps::rtps::IPLocator::setIPv4(initial_peer_locator, "192.168.1.10");
eprosima::fastrtps::rtps::IPLocator::setWan(initial_peer_locator, "80.80.99.45");
// [ALTERNATIVE] Use server's WAN address only. Valid if server specified its unicast␣
↪locators
// with its LAN or WAN address.
// eprosima::fastrtps::rtps::IPLocator::setIPv4(initial_peer_locator, "80.80.99.45");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(initial_peer_locator, 5100);
// If the logical port is set in the server side, it must be also set here with the same␣
↪value.
// If not set in the server side in a unicast locator, do not set it here.
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(initial_peer_locator, 5100);

qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer_locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
-->
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tcp_client_wan_transport</transport_id>
            <type>TCPv4</type>
            <!-- Recommended client's WAN set -->
            <wan_addr>80.80.99.47</wan_addr>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="tcp_client_wan_participant">
        <rtps>
            <userTransports>
                <transport_id>tcp_client_wan_transport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
            <builtin>
                <initialPeersList>
                    <locator>
                        <tcpv4>
                            <!-- Recommended use of both WAN and LAN server addresses -->
```

(continues on next page)

(continued from previous page)

```
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.10</address>
                        <!-- Alternatively use server's WAN addresses only -->
                        <!-- <address>80.80.99.45</address> -->
                        <physical_port>5100</physical_port>
                        <!-- To be set if set in server -->
                        <port>5100</port>
                    </tcpv4>
                </locator>
            </initialPeersList>
        </builtin>
    </rtps>
</participant>
</profiles>
```

HelloWorldExampleTCP

A TCP version of helloworld example can be found in the HelloWorldExampleTCP folder. It shows a publisher and a subscriber that communicate through TCP. The publisher is configured as *TCP server* while the Subscriber is acting as *TCP client*.

6.20.4 Shared Memory Transport

The shared memory (SHM) transport enables fast communications between entities running in the same processing unit/machine, relying on the shared memory mechanisms provided by the host operating system.

Note: Fast DDS utilizes the *DomainParticipant*'s `GuidPrefix_t` to identify peers running in the same host. Two participants with identical 4 first bytes on the `GuidPrefix_t` are considered to be running in the same host. `is_on_same_host_as()` API is provided to check this condition. Please, take also into account the caveats included in *GUID Prefix considerations for intra-process delivery*.

SHM transport provides better performance than other network transports like UDP / TCP, even when these transports use loopback interface. This is mainly due to the following reasons:

- Large message support: Network protocols need to fragment data in order to comply with the specific protocol and network stacks requirements, increasing communication overhead. SHM transport allows the copy of full messages where the only size limit is the machine's memory capacity.
- Reduce the number of memory copies: When sending the same message to different endpoints, SHM transport can directly share the same memory buffer with all the destination endpoints. Other protocols require to perform one copy of the message per endpoint.
- Less operating system overhead: Once initial setup is completed, shared memory transfers require much less system calls than the other protocols. Therefore, there is a performance/time consume gain by using SHM.

Definition of Concepts

This section describes basic concepts to help explain how the Shared Memory Transport works in order to deliver the data messages to the appropriate *DomainParticipant*. The purpose is not to be an exhaustive reference of the implementation, but to be a comprehensive explanation of each concept, so that users can configure the transport to their needs.

Many of the descriptions in this section will be made following the example use case depicted in the following figure, where *Participant 1* sends a data message to *Participant 2*. Please, refer to the figure when following the definitions.

Fig. 11: Sequence diagram for Shared Memory Transport

Segment

A *Segment* is a block of shared memory that can be accessed from different processes. Every DomainParticipant that has been configured with Shared Memory Transport creates a segment of shared memory. The DomainParticipant writes to this segment any data it needs to deliver to other DomainParticipants, and the remote DomainParticipants are able to read it directly using the shared memory mechanisms.

Note: Launching any of the processes with a higher privileged user (for instance, *root*) can lead to communication problems, as processes run by non-privileged users may not be able to write into the memory segment.

Every segment has a *segmentId*, a 16-character UUID that uniquely identifies each shared memory segment. These *segmentIds* are used to identify and access the segment of each DomainParticipant.

Segment Buffer

A buffer allocated in the shared memory Segment. It works as a container for a DDS message that is placed in the Segment. In other words, each message that the DomainParticipant writes on the Segment will be placed in a different buffer.

Buffer Descriptor

It acts as a pointer to a specific Segment Buffer in a specific Segment. It contains the *segmentId* and the offset of the Segment Buffer from the base of the Segment. When communicating a message to other DomainParticipants, Shared Memory Transport only distributes the Buffer Descriptor, avoiding the copy of the message from a DomainParticipant to another. With this descriptor, the receiving DomainParticipant can access the message written in the buffer, as is uniquely identifies the Segment (through the *segmentId*) and the Segment Buffer (through its offset).

Port

Represents a channel to communicate Buffer Descriptors. It is implemented as a ring-buffer in shared memory, so that any DomainParticipant can potentially read or write information on it. Each port has a unique identifier, a 32 bit number that can be used to refer to the port. Every DomainParticipant that has been configured with Shared Memory Transport creates a port to receive Buffer Descriptors. The identifier of this port is shared during the *Discovery*, so that remote peers know which port to use when they want to communicate with each DomainParticipant.

DomainParticipants create a listener to their receiving port, so that they can be notified when a new Buffer Descriptor is pushed to the port.

Port Health Check

Every time a DomainParticipant opens a Port (for reading or writing), a health check is performed to assess its correctness. The reason is that if one of the processes involved crashes while using a Port, that port can be left inoperative. If the attached listeners do not respond in a given timeout, the Port is considered damaged, and it is destroyed and created again.

SharedMemTransportDescriptor

In addition to the data members defined in the *TransportDescriptorInterface*, the TransportDescriptor for Shared Memory defines the following ones:

Member	Data type	Default	Accessor / Mutator	Description
`segment_size_`	`uint32_t`	`512*1024`	*`segment_size()`*	Size of the shared memory segment (in octets).
`port_queue_capacity_`	`uint32_t`	`512`	*`port_queue_capacity()`*	The size of the listening port (in messages).
`healthy_check_timeout_ms_`	`uint32_t`	`1000`	*`healthy_check_timeout_ms()`*	Timeout for the health check of ports (in milliseconds).
`rtps_dump_file_`	`string`	`""`	*`rtps_dump_file()`*	Full path of the protocol dump file.

If `rtps_dump_file_` is not empty, all the shared memory traffic on the DomainParticipant (sent and received) is traced to a file. The output file format is *tcpdump* hexadecimal text, and can be processed with protocol analyzer applications such as Wireshark. Specifically, to open the file using Wireshark, use the "Import from Hex Dump" option using the "Raw IPv4" encapsulation type.

Note: The *`kind`* value for a *`SharedMemTransportDescriptor`* is given by the value *`LOCATOR_KIND_SHM`*.

Warning: Setting a *`segment_size()`* close to or smaller than the data size poses a high risk of data loss, since the write operation will overwrite the buffer during a single send operation.

Enabling Shared Memory Transport

Fast DDS enables a SHM transport by default. Nevertheless, the application can enable other SHM transports if needed. To enable a new SHM transport in a *DomainParticipant*, first create an instance of *SharedMemTransportDescriptor*, and add it to the user transport list of the *DomainParticipant*.

The examples below show this procedure in both C++ code and XML file.

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
std::shared_ptr<SharedMemTransportDescriptor> shm_transport = std::make_shared
↪<SharedMemTransportDescriptor>();

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(shm_transport);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <!-- Create a descriptor for the new transport -->
        <transport_descriptor>
            <transport_id>shm_transport</transport_id>
            <type>SHM</type>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="SHMParticipant">
        <rtps>
            <!-- Link the Transport Layer to the Participant -->
            <userTransports>
                <transport_id>shm_transport</transport_id>
            </userTransports>
        </rtps>
    </participant>
</profiles>
```

Note: In case that several transports are enabled, the discovery traffic is always performed using the UDP/TCP transport, even if the SHM transport is enabled in both participants running in the same machine. This may cause discovery issues if one or several of the participants only has SHM enabled and other participants use some other transport at the same time. Also, when two participants on the same machine have SHM transport enabled, the user data communication between them is automatically performed by SHM transport only. The rest of the enabled transports are not used between those two participants.

Hint: To configure discovery traffic through Shared Memory, the default builtin transports must be disabled. In that way, communication is performed completely using Shared Memory. The snippet examples below show this procedure in both C++ code and XML file. See *HelloWorldExampleSharedMem* for a complete example.

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
std::shared_ptr<SharedMemTransportDescriptor> shm_transport = std::make_shared
↪<SharedMemTransportDescriptor>();

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(shm_transport);

// Explicit configuration of SharedMem transport
qos.transport().use_builtin_transports = false;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <!-- Create a descriptor for the new transport -->
        <transport_descriptor>
            <transport_id>shm_transport_only</transport_id>
            <type>SHM</type>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="DisableBuiltinTransportsParticipant">
        <rtps>
            <!-- Link the Transport Layer to the Participant -->
            <userTransports>
                <transport_id>shm_transport_only</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
        </rtps>
    </participant>
</profiles>
```

HelloWorldExampleSharedMem

A Shared Memory version of helloworld example can be found in the HelloWorldExampleSharedMem folder. It shows a publisher and a subscriber that communicate through Shared Memory.

6.20.5 Data-sharing delivery

Fast DDS allows to speed up communications between entities within the same machine by sharing the history of the *DataWriter* with the *DataReader* through shared memory. This prevents any of the overhead involved in the transport layer, effectively avoiding any data copy between DataWriter and DataReader.

Note: Fast DDS utilizes the *DomainParticipant*'s `GuidPrefix_t` to identify peers running in the same host. Two participants with identical 4 first bytes on the `GuidPrefix_t` are considered to be running in the same host. `is_on_same_host_as()` API is provided to check this condition. Please, take also into account the caveats included

in *GUID Prefix considerations for intra-process delivery*.

Use of Data-sharing delivery does not prevent data copies between the application and the DataReader and DataWriter. These can be avoided in some cases using *Zero-Copy communication*.

Note: Although Data-sharing delivery uses shared memory, it differs from *Shared Memory Transport* in that Shared Memory is a full-compliant transport. That means that with Shared Memory Transport the data being transmitted must be copied from the DataWriter history to the transport and from the transport to the DataReader. With Data-sharing these copies can be avoided.

- *Overview*
- *Constraints*
- *Data-sharing delivery configuration*
- *DataReader and DataWriter history coupling*

Overview

When the DataWriter is created, *Fast DDS* will pre-allocate a pool of `max_samples` + `extra_samples` samples that reside in a shared memory mapped file. When publishing new data, the DataWriter will take a sample from this pool and add it to its history, and notify the DataReader which sample from the pool has the new data.

The DataReader will have access to the same shared memory mapped file, and will be able to access the data published by the DataWriter.

Constraints

This feature is available only if the following requirements are met:

- The *DataWriter* and *DataReader* have access to the same shared memory.
- The *Topic* has a bounded *TopicDataType*, i.e., its `is_bounded()` member function returns true.
- The Topic *is not keyed*.
- The DataWriter is configured with `PREALLOCATED_MEMORY_MODE` or `PREALLOCATED_WITH_REALLOC_MEMORY_MODE`.

There is also a limitation with the DataReader's HistoryQos. Using Data-sharing mechanism, the DataWriter's history is shared with the DataReaders. This means that the effective HistoryQos depth on the DataReader is, at most, the Datawriter's HistoryQos depth. To avoid confusions, set the DataReaders' history depth to a value equal or less than the DataWriter's.

Data-sharing delivery configuration

Data-sharing delivery can be configured in the *DataWriter* and the *DataReader* using *DataSharingQosPolicy*. Four attributes can be configured:

- The data-sharing delivery kind
- The shared memory directory
- The data-sharing domain identifiers.
- The maximum number of data-sharing domain identifiers.

Data-Sharing delivery kind

Can be set to one of three modes:

- **AUTO**: If both a DataWriter and DataReader meet the requirements, data-sharing delivery will be used between them. This is the default value.
- **ON**: Like **AUTO**, but the creation of the entity will fail if the requirements are not met.
- **OFF**: No data-sharing delivery will be used on this entity.

The following matrix shows when two entities are data-sharing compatible according to their configuration (given that the entity creation does not fail and that both entities have access to a shared memory):

		Reader		
		ON	**OFF**	**AUTO**
Writer	**ON**	Only if they have common domain IDs	No	Only if they have common domain IDs
	OFF	No	No	No
	AUTO	Only if they have common domain IDs	No	Only if the TopicDataType is bounded and they have common domain IDs

Data-sharing domain identifiers

Each entity defines a set of identifiers that represent a *domain* to which the entity belongs. Two entities will be able to use data-sharing delivery between them only if both have at least a common domain.

Users can define the domains of a *DataWriter* or *DataReader* with the *DataSharingQosPolicy*. If no domain identifier is provided by the user, the system will create one automatically. This automatic data-sharing domain will be unique for the machine where the entity is running. That is, all entities running on the same machine, and for which the user has configured no user-specific domains, will be able to use data-sharing delivery (given that the rest of requirements are met).

During the discovery phase, entities will exchange their domain identifiers and check if they can use Data-sharing to communicate.

Note: Even though a data-sharing domain identifier is a 64 bit integer, user-defined identifiers are restricted to 16 bit integers.

Maximum number of Data-sharing domain identifiers

The maximum number of domain identifiers that are expected to be received from a remote entity during discovery. If the remote entity defines (and sends) more than this number of domain identifiers, the discovery will fail.

By default there is no limit to the number of identifiers. The default value can be changed with the *max_domains()* function. Defining a finite number allows to preallocate the required memory to receive the list of identifiers during the entity creation, avoiding dynamic memory allocations afterwards. Note that a value of `0` means no limit.

Shared memory directory

If a user-defined directory is given for the shared memory files, this directory will be used for the memory-mapped files used for data-sharing delivery. If none is given, the default directory configured for the current system is used.

Configuring a user-defined directory may be useful in some scenarios:

- To select a file system with Huge TLB enabled for the memory-mapped files.
- To allow data-sharing delivery between containers that mount the same container.

DataReader and DataWriter history coupling

With traditional *Transport Layer* delivery, the DataReader and DataWriter keep separate and independent histories, each one with their own copy of the sample. Once the sample is sent through the transport and received by the DataReader, the DataWriter is free to remove the sample from its history without affecting the DataReader.

With data-sharing delivery, the DataReader directly accesses the data instance created by the DataWriter. This means that the samples in both the history of the DataReader and the DataWriter refer to the same object in the shared memory. Therefore, there is a strong coupling in the behavior of the DataReader and DataWriter histories.

Important: If the DataWriter reuses the same sample to publish new data, the DataReader loses access to the old data sample.

Note: The DataWriter can remove the sample from its history, and it will still be available on the DataReader, unless the same sample from the pool is reused to publish a new one.

Data acknowledgement

With data-sharing delivery, sample acknowledgment from the DataReader occurs the first time a sample is retrieved by the application (using *DataReader::read_next_sample()*, *DataReader::take_next_sample()*, or any of their variations). Once the data has been accessed by the application, the DataWriter is free to reuse that sample to publish new data. The DataReader detects when a sample has been reused and automatically removes it from its history.

This means that subsequent attempts to access the same sample from the DataReader may return no sample at all.

Blocking reuse of samples until acknowledged

With *KEEP_LAST_HISTORY_QOS* or *BEST_EFFORT_RELIABILITY_QOS* configurations, the DataWriter can remove samples from its history to add new ones, even if they were not acknowledged by the DataReader. In situations where the publishing rate is consistently faster than the rate at which the DataReader can process the samples, this can lead to every sample being reused before the application has a chance to process it, thus blocking the communication at application level.

In order to avoid this situation, the samples in the preallocated pool are never reused unless they have been acknowledged, i.e., they have been processed by the application at least once. If there is no reusable sample in the pool, the writing operation in the DataWriter will be blocked until one is available or until *max_blocking_time* is reached.

Note that the DataWriter history is not affected by this behavior, samples will be removed from the history by standard rules. Only the reuse of pool samples is affected. This means that the DataWriter history can be empty and the write operation be still blocked because all samples in the pool are unacknowledged.

The chance of the DataWriter blocking on a write operation can be reduced using *extra_samples*. This will make the pool to allocate more samples than the history size, so that the DataWriter has more chances to get a free sample, while the DataReader can still access samples that have been removed from the DataWriter history.

6.20.6 Intra-process delivery

eProsima Fast DDS allows to speed up communications between entities within the same process by avoiding any of the overhead involved in the transport layer. Instead, the *Publisher* directly calls the reception functions of the *Subscriber*. This not only avoids the copy or send operations of the transport, but also ensures the message is received by the Subscriber, avoiding the acknowledgement mechanism.

This feature is enabled by default, and can be configured using *XML profiles* (see *Intra-process delivery profiles*). Currently the following options are available:

- **INTRAPROCESS_OFF**: The feature is disabled.
- **INTRAPROCESS_USER_DATA_ONLY**: Discovery metadata keeps using ordinary transport.
- **INTRAPROCESS_FULL**: Default value. Both user data and discovery metadata using Intra-process delivery.

XML

```
<library_settings>
    <intraprocess_delivery>FULL</intraprocess_delivery> <!-- OFF | USER_DATA_ONLY |␣
↪FULL -->
</library_settings>
```

GUID Prefix considerations for intra-process delivery

Fast DDS utilizes the *DomainParticipant*'s *GuidPrefix_t* to identify peers running in the same process. Two participants with identical 8 first bytes on the *GuidPrefix_t* are considered to be running in the same process, and therefore intra-process delivery is used. *is_on_same_process_as()* API is provided to check this condition. This mechanism works out-of-the-box when letting Fast DDS set the GUID prefixes for the created DomainParticipants. However, special consideration is required when setting the *GuidPrefix_t* manually, either programmatically or when using XML.

Important: Fast DDS assigns GUID prefixes considering several host parameters. Among them, the network interfaces enabled. Thus, if at runtime, the network interfaces change, any new DomainParticipant will have a different

GUID prefix and will be considered to be running on another host.

C++ - ASCII format.

```
eprosima::fastrtps::rtps::GuidPrefix_t guid_prefix;
guid_prefix.value[0] = eprosima::fastrtps::rtps::octet(0x77);
guid_prefix.value[1] = eprosima::fastrtps::rtps::octet(0x73);
guid_prefix.value[2] = eprosima::fastrtps::rtps::octet(0x71);
guid_prefix.value[3] = eprosima::fastrtps::rtps::octet(0x85);
guid_prefix.value[4] = eprosima::fastrtps::rtps::octet(0x69);
guid_prefix.value[5] = eprosima::fastrtps::rtps::octet(0x76);
guid_prefix.value[6] = eprosima::fastrtps::rtps::octet(0x95);
guid_prefix.value[7] = eprosima::fastrtps::rtps::octet(0x66);
guid_prefix.value[8] = eprosima::fastrtps::rtps::octet(0x65);
guid_prefix.value[9] = eprosima::fastrtps::rtps::octet(0x82);
guid_prefix.value[10] = eprosima::fastrtps::rtps::octet(0x82);
guid_prefix.value[11] = eprosima::fastrtps::rtps::octet(0x79);

DomainParticipantQos participant_qos;
participant_qos.wire_protocol().prefix = guid_prefix;
```

C++ - Extraction operator.

```
DomainParticipantQos participant_qos;
std::istringstream("77.73.71.85.69.76.95.66.65.82.82.79") >> participant_qos.wire_
↪protocol().prefix;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_guidprefix" >
        <rtps>
            <prefix>77.73.71.85.69.76.95.66.65.82.82.79</prefix>
        </rtps>
    </participant>
</profiles>
```

6.20.7 TLS over TCP

Warning: This documentation assumes the reader has basic knowledge of TLS concepts since terms like Certificate Authority (CA), Private Key, *Rivest–Shamir–Adleman* (RSA) cryptosystem, and Diffie-Hellman encryption protocol are not explained in detail.

Fast DDS allows configuring TCP Transports to use TLS (Transport Layer Security). In order to set up TLS, the *TCPTransportDescriptor* must have its `apply_security` data member set to `true`, and its `tls_config` data member filled with the desired configuration on the `TCPTransportDescriptor`. The following is an example of configuration of TLS on the *TCP server*.

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tls_transport = std::make_shared<TCPv4TransportDescriptor>();
tls_transport->sendBufferSize = 9216;
tls_transport->receiveBufferSize = 9216;
tls_transport->add_listener_port(5100);

// Create the TLS configuration
using TLSOptions =␣
↪eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSOptions;
tls_transport->apply_security = true;
tls_transport->tls_config.password = "test";
tls_transport->tls_config.cert_chain_file = "server.pem";
tls_transport->tls_config.private_key_file = "serverkey.pem";
tls_transport->tls_config.tmp_dh_file = "dh2048.pem";
tls_transport->tls_config.add_option(TLSOptions::DEFAULT_WORKAROUNDS);
tls_transport->tls_config.add_option(TLSOptions::SINGLE_DH_USE);
tls_transport->tls_config.add_option(TLSOptions::NO_SSLV2);

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tls_transport);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tls_transport_server</transport_id>
            <type>TCPv4</type>
            <tls>
                <password>test</password>
                <private_key_file>serverkey.pem</private_key_file>
                <cert_chain_file>server.pem</cert_chain_file>
                <tmp_dh_file>dh2048.pem</tmp_dh_file>
                <options>
                    <option>DEFAULT_WORKAROUNDS</option>
                    <option>SINGLE_DH_USE</option>
                    <option>NO_SSLV2</option>
                </options>
            </tls>
            <sendBufferSize>9216</sendBufferSize>
            <receiveBufferSize>9216</receiveBufferSize>
            <listening_ports>
                <port>5100</port>
            </listening_ports>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TLSServerParticipant">
        <rtps>
            <userTransports>
```

(continues on next page)

(continued from previous page)

```
                <transport_id>tls_transport_server</transport_id>
            </userTransports>
        </rtps>
    </participant>
</profiles>
```

The corresponding configuration on the *TCP client* is shown in the following example.

C++

```
DomainParticipantQos qos;

// Set initial peers.
Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
IPLocator::setIPv4(initial_peer_locator, "192.168.1.10");
initial_peer_locator.port = 5100;
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer_locator);

// Create a descriptor for the new transport.
auto tls_transport = std::make_shared<TCPv4TransportDescriptor>();

// Create the TLS configuration
using TLSOptions =␣
↪eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSOptions;
using TLSVerifyMode =␣
↪eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSVerifyMode;
tls_transport->apply_security = true;
tls_transport->tls_config.verify_file = "ca.pem";
tls_transport->tls_config.add_verify_mode(TLSVerifyMode::VERIFY_PEER);
tls_transport->tls_config.add_verify_mode(TLSVerifyMode::VERIFY_FAIL_IF_NO_PEER_CERT);
tls_transport->tls_config.add_option(TLSOptions::DEFAULT_WORKAROUNDS);
tls_transport->tls_config.add_option(TLSOptions::SINGLE_DH_USE);
tls_transport->tls_config.add_option(TLSOptions::NO_SSLV2);
tls_transport->tls_config.server_name = "my_server.com";

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tls_transport);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tls_transport_client</transport_id>
            <type>TCPv4</type>
            <tls>
                <verify_file>ca.pem</verify_file>
                <verify_mode>
                    <verify>VERIFY_PEER</verify>
                    <verify>VERIFY_FAIL_IF_NO_PEER_CERT</verify>
                </verify_mode>
```

(continues on next page)

(continued from previous page)

```
                <options>
                    <option>DEFAULT_WORKAROUNDS</option>
                    <option>SINGLE_DH_USE</option>
                    <option>NO_SSLV2</option>
                </options>
                <server_name>my_server.com</server_name>
            </tls>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TLSClientParticipant">
        <rtps>
            <userTransports>
                <transport_id>tls_transport_client</transport_id>
            </userTransports>
            <builtin>
                <initialPeersList>
                    <locator>
                        <tcpv4>
                            <address>192.168.1.10</address>
                            <physical_port>5100</physical_port>
                        </tcpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

The following table describes the data members that are configurable on *TLSConfig*.

Member	Data type	Default	Description
password	string	""	Password of the *private_key_file* or *rsa_private_key_file*.
private_key_file	string	""	Path to the private key certificate file.
rsa_private_key_file	string	""	Path to the private key RSA certificate file.
cert_chain_file	string	""	Path to the public certificate chain file.
tmp_dh_file	string	""	Path to the Diffie-Hellman parameters file.
verify_file	string	""	Path to the CA (Certification- Authority) file.
verify_mode	*TLSVerifyMode*	*TLSVerifyMode::UNUSED*	Establishes the verification mode mask. See *TLS Verification Mode*.
options	*TLSOptions*	*TLSOptions::NONE*	Establishes the SSL Context options mask. See *TLS Options*.
verify_paths	vector<string>	Empty vector	Paths where the system will look for verification files.
verify_depth	int32_t	-1	Maximum allowed depth for verifying intermediate certificates.
default_verify_path	bool	false	Look for verification files on the default paths.
handshake_role	*TLSHandShakeRole*	*TLSHandShakeRole::DEFAULT*	Role that the transport will take on handshaking. See *TLS Handshake Role*.
server_name	string	""	Server name or host name required in case Server Name Indication (SNI) is used.

Note: *Fast DDS* uses the Boost.Asio library to handle TLS secure connections. These data members are used to build the asio library context, and most of them are mapped directly into this context without further manipulation. You can find more information about the implications of each member on the Boost.Asio context documentation.

TLS Verification Mode

The verification mode defines how the peer node will be verified. The following table describes the available verification options. Several verification options can be combined in the same *TCPTransportDescriptor* using the *add_verify_mode()* member function.

Value	Description
TLSVerifyMode::VERIFY_NONE	Perform no verification.
TLSVerifyMode::VERIFY_PEER	Perform verification of the peer.
TLSVerifyMode::VERIFY_FAIL_IF_NO_PEER_CERT	Fail verification if the peer has no certificate. Ignored unless *TLSVerifyMode::VERIFY_PEER* is also set.
TLSVerifyMode::VERIFY_CLIENT_ONCE	Do not request client certificate on renegotiation. Ignored unless *TLSVerifyMode::VERIFY_PEER* is also set.

Note: For a complete description of the different verification modes, please refer to the OpenSSL documentation.

TLS Options

These options define which TLS features are to be supported. The following table describes the available options. Several options can be combined in the same TransportDescriptor using the *add_option()* member function.

Value	Description
TLSOptions::DEFAULT_WORKAROUNDS	Implement various bug workarounds. See Boost.Asio context.
TLSOptions::NO_COMPRESSION	Disable compression.
TLSOptions::NO_SSLV2	Disable SSL v2.
TLSOptions::NO_SSLV3	Disable SSL v3.
TLSOptions::NO_TLSV1	Disable TLS v1.
TLSOptions::NO_TLSV1_1	Disable TLS v1.1.
TLSOptions::NO_TLSV1_2	Disable TLS v1.2.
TLSOptions::NO_TLSV1_3	Disable TLS v1.3.
TLSOptions::SINGLE_DH_USE	Always create a new key when using *Diffie-Hellman* parameters.

TLS Handshake Role

The role can take the following values:

Value	Description
TLSHandShakeRole::DEFAULT	Configured as client if connector, and as server if acceptor
TLSHandShakeRole::CLIENT	Configured as client.
TLSHandShakeRole::SERVER	Configured as server.

6.20.8 Listening Locators

Listening *Locators* are used to receive incoming traffic on the *DomainParticipant*. These Locators can be classified according to the communication type and to the nature of the data.

According to the communication type we have:

- **Multicast locators**: Listen to multicast communications.
- **Unicast locators**: Listen to unicast communications.

According to the nature of the data we have:

- **Metatraffic locators**: Used to receive metatraffic information, usually used by built-in endpoints to perform discovery.
- **User locators**: Used by the endpoints created by the user to receive user *Topic* data changes.

Applications can *provide their own Listening Locators*, or use the *Default Listening Locators* provided by *eProsima Fast DDS*.

Adding Listening Locators

Users can add custom Listening Locators to the DomainParticipant using the *DomainParticipantQos*. Depending on the field where the Locator is added, it will be treated as a *multicast*, *unicast*, *user* or *metatraffic* Locator.

Note: Both UDP and TCP unicast Locators support to have a null address. In that case, *Fast DDS* automatically gets and uses local network addresses.

Note: Both UDP and TCP Locators support to have a zero port. In that case, *Fast DDS* automatically calculates and uses well-known ports for that type of traffic. See *Well Known Ports* for details about the well-known ports.

Warning: TCP does not support multicast scenarios, so the network architecture must be carefully planned.

Metatraffic Multicast Locators

Users can set their own metatraffic multicast locators within the *WireProtocolConfigQos*: `builtin.metatrafficMulticastLocatorList`.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22222 over multicast address 239.255.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 239, 255, 0, 1);
locator.port = 22222;

// Add the locator to the DomainParticipantQos
qos.wire_protocol().builtin.metatrafficMulticastLocatorList.push_back(locator);
```

XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="CustomMetatrafficMulticastParticipant">
        <rtps>
            <builtin>
                <metatrafficMulticastLocatorList>
                    <!-- LOCATOR_LIST -->
                    <locator>
                        <udpv4>
                            <address>239.255.0.1</address>
                            <port>22222</port>
                        </udpv4>
                    </locator>
                </metatrafficMulticastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

Metatraffic Unicast Locators

Users can set their own metatraffic unicast locators within the *WireProtocolConfigQos*: *builtin.metatrafficUnicastLocatorList*.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22223 over address 192.168.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 0, 1);
locator.port = 22223;

// Add the locator to the DomainParticipantQos
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
```

XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="CustomMetatrafficUnicastParticipant">
        <rtps>
            <builtin>
                <metatrafficUnicastLocatorList>
                    <!-- LOCATOR_LIST -->
                    <locator>
                        <udpv4>
                            <address>192.168.0.1</address>
                            <port>22223</port>
                        </udpv4>
```

(continues on next page)

(continued from previous page)

```
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

User-traffic Multicast Locators

Users can set their own user-traffic multicast locators within the *WireProtocolConfigQos*: *default_multicast_locator_list*.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22224 over multicast address 239.255.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 239, 255, 0, 1);
locator.port = 22224;

// Add the locator to the DomainParticipantQos
qos.wire_protocol().default_multicast_locator_list.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="CustomUsertrafficMulticastParticipant">
        <rtps>
            <defaultMulticastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <address>239.255.0.1</address>
                        <port>22224</port>
                    </udpv4>
                </locator>
            </defaultMulticastLocatorList>
        </rtps>
    </participant>
</profiles>
```

User-traffic Unicast Locators

Users can set their own user-traffic unicast locators within the *WireProtocolConfigQos*: *default_unicast_locator_list*.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22225 over address 192.168.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 0, 1);
locator.port = 22225;

// Add the locator to the DomainParticipantQos
qos.wire_protocol().default_unicast_locator_list.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="CustomUsertrafficUnicastParticipant">
        <rtps>
            <defaultUnicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <address>192.168.0.1</address>
                        <port>22225</port>
                    </udpv4>
                </locator>
            </defaultUnicastLocatorList>
        </rtps>
    </participant>
</profiles>
```

Default Listening Locators

If the application does not define any Listening Locators, *eProsima Fast DDS* automatically enables a set of listening UDPv4 locators by default. This allows out-of-the-box communication in most cases, without the need of further configuring the *Transport Layer*.

- If the application does not define any *metatraffic* Locator (neither *unicast* nor *multicast*), *Fast DDS* enables one *multicast* Locator that will be used during *Discovery*, and one *unicast* Locator that will be used for peer-to-peer communication with already discovered DomainParticipants.
- If the application does not define any *user-traffic* Locator (neither *unicast* nor *multicast*), *Fast DDS* enables one *unicast* Locator that will be used for peer-to-peer communication of *Topic* data.

For example, it is possible to prevent *multicast* traffic adding a single *metatraffic unicast* Locator as described in *Disabling all Multicast Traffic*.

Default Listening Locators always use *Well Known Ports*.

Well Known Ports

The DDSI-RTPS V2.2 standard (Section 9.6.1.1) defines a set of rules to calculate well-known ports for default Locators, so that DomainParticipants can communicate with these default Locators. Well-known ports are also selected automatically by *Fast DDS* when a Locator is configured with port number *0*.

Well-known ports are calculated using the following predefined rules:

Table 10: Rules to calculate ports on default listening locators

Traffic type	Well-known port expression
Metatraffic multicast	`PB` + `DG` * *domainId* + `offsetd0`
Metatraffic unicast	`PB` + `DG` * *domainId* + `offsetd1` + `PG` * *participantId*
User multicast	`PB` + `DG` * *domainId* + `offsetd2`
User unicast	`PB` + `DG` * *domainId* + `offsetd3` + `PG` * *participantId*

The values used in these rules are explained on the following table. The default values can be modified using the *port* member of the *WireProtocolConfigQos* on the *DomainParticipantQos*.

Table 11: Values used in the rules to calculate well-known ports

Symbol	Meaning	Default value	QoS field
`DG`	DomainID gain	250	*wire_protocol().port.domainIDGain*
`PG`	ParticipantId gain	2	*wire_protocol().port.participantIDGain*
`PB`	Port Base number	7400	*wire_protocol().port.portBase*
`offsetd0`	Additional offset	0	*wire_protocol().port.offsetd0*
`offsetd1`	Additional offset	10	*wire_protocol().port.offsetd1*
`offsetd2`	Additional offset	1	*wire_protocol().port.offsetd2*
`offsetd3`	Additional offset	11	*wire_protocol().port.offsetd3*

6.20.9 Announced Locators

In order for communication to take place, DDS entities need to exchange the list of addresses and ports where they can be reached. Apart from the *default announced locators*, which correspond to addresses of the interfaces in the host where the application is running, the user can configure *additional locators* with addresses and ports on other networks, when routing rules have been correspondingly set up.

Default Announced Locators

The default list of announced locators will be constructed from the *listening locators*, as follows:

- If the address field of the locator is a null address (i.e. 0.0.0.0 for UDPv4), a locator of the same kind and port will be announced for each of the addresses of the network interfaces of the host.
- If the address field of the locator is not a null address, a single locator with that address will be announced.

External Locators

The user can configure a set of external locators for each of the lists of unicast locators:

- `builtin.metatraffic_external_unicast_locators` on *WireProtocolConfigQos*
- `default_external_unicast_locators` on *WireProtocolConfigQos*
- `external_unicast_locators` on *RTPSEndpointQos*

An external locator is made up of the standard locator fields (kind, address, and port), plus the following attributes:

- An *externality* that indicates the number of hops from the host where the application is running to the LAN represented by the external locator.
- A *cost* indicating the communication cost relative to other locators on the same externality level.
- A *mask* with the number of significant bits on the LAN represented by the external locator.

Externality levels

The main purpose of the external locators is to enable communication across different levels of interconnected LANs. Communication will be performed using the locators of the innermost LAN available.

As an example, consider a network topology where the application is running on a host connected to a LAN of an office, which in turn connects to a LAN for all the offices in the same floor, which in turn connects to a LAN for the building.

With the default configuration, communication will only occur between hosts on the LAN for the office. This is considered the externality level 0, which is reserved for the LANs directly connected to the network interfaces of the host where the application is running. This is the externality level that will be used on the *matching algorithm* for the *default announced locators*. The floor LAN will be configured as externality level 1, whereas the building LAN will be configured as externality level 2.

Note that in order for the communication to be successful, routing rules should most probably need to be added to the different network routers.

Important: Externality level 0 is automatically populated by Fast DDS and cannot be configured by the application.

Matching algorithm

When a remote entity is discovered, its list of announced locators is processed to select the ones on the innermost externality level where the communication can be established. The highest externality level is checked first.

If the discovered addresses for one level are equal to the ones announced by the local entity, it means they are on the same host at that level, and the algorithm proceeds to an inner level. If the discovered addresses are not equal to the ones announced by the local entity, processing stops at the current level.

When the externality level on which the communication will be established has been decided, the algorithm will:

- Remove locators that match with addresses on any other externality level.
- Keep locators that match with the selected externality level.
- For the locators with an address that does not match with any of the locators announced by the local entity:
  - Keep them when `ignore_non_matching_locators` is `false` (default behavior)
  - Remove them when `ignore_non_matching_locators` is `true`

Additional considerations

Since using external locators increases the number of locators announced, the *allocation limits for locators discovery* would need to be adjusted for your application.

Participants running on the same host, but using different addresses on their *builtin.metatraffic_external_unicast_locators* will discard shared memory transport locators. Data sharing communication is not affected by this limitation.

6.20.10 Interface Whitelist

Using *Fast DDS*, it is possible to limit the network interfaces used by *TCP Transport* and *UDP Transport*. This is achieved by adding the interfaces' IP addresses to the *interfaceWhiteList* field in the *TCPTransportDescriptor* or *UDPTransportDescriptor*. Thus, the communication interfaces used by the *DomainParticipants* whose *TransportDescriptorInterface* defines an *interfaceWhiteList* is limited to the interfaces' IP addresses defined in that list, therefore avoiding the use of the rest of the network interfaces available in the system. The values on this list should match the IPs of your machine in that networks. For example:

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<TCPv4TransportDescriptor>();

// Add loopback to the whitelist
tcp_transport->interfaceWhiteList.emplace_back("127.0.0.1");

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tcp_transport);

// Avoid using the builtin transports
qos.transport().use_builtin_transports = false;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>CustomTcpTransport</transport_id>
            <type>TCPv4</type>
            <interfaceWhiteList>
                <address>127.0.0.1</address>
            </interfaceWhiteList>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="CustomTcpTransportParticipant">
        <rtps>
            <useBuiltinTransports>false</useBuiltinTransports>
            <userTransports>
                <transport_id>CustomTcpTransport</transport_id>
```

(continues on next page)

(continued from previous page)

```
            </userTransports>
        </rtps>
    </participant>
</profiles>
```

Warning: The interface whitelist feature applies to network interfaces. Therefore, it is only available on *TCP Transport* and *UDP Transport*.

6.20.11 Disabling all Multicast Traffic

If all the peers are known beforehand and have been configured on the *Initial Peers List*, all multicast traffic can be completely disabled.

By defining a custom *Metatraffic Unicast Locators*, the local *DomainParticipant* creates a unicast meta traffic receiving resource for each address-port pair specified in the list, avoiding the creation of the default metatraffic multicast and unicast locators. This prevents the DomainParticipant from listening to any discovery data from multicast sources.

Consideration should be given to the assignment of the ports in the `metatrafficUnicastLocatorList`, avoiding the assignment of ports that are not available or do not match the address-port listed in the publisher participant Initial Peers List.

The following is an example of how to disable all multicast traffic configuring one *metatraffic unicast* locator.

C++

```
DomainParticipantQos qos;

// Metatraffic Multicast Locator List will be empty.
// Metatraffic Unicast Locator List will contain one locator, with null address and null␣
↪port.
// Then Fast DDS will use all network interfaces to receive network messages using a␣
↪well-known port.
Locator_t default_unicast_locator;
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(default_unicast_
↪locator);

// Initial peer will be UDPv4 address 192.168.0.1. The port will be a well-known port.
// Initial discovery network messages will be sent to this UDPv4 address.
Locator_t initial_peer;
IPLocator::setIPv4(initial_peer, 192, 168, 0, 1);
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="disable_multicast" is_default_profile="true">
        <rtps>
            <builtin>
                <metatrafficUnicastLocatorList>
                    <locator>
```

(continues on next page)

(continued from previous page)

```
                        <udpv4>
                            <address>192.168.0.1</address>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
                <initialPeersList>
                    <locator>
                        <udpv4>
                            <address>192.168.0.1</address>
                        </udpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

6.21 Persistence Service

Using default QoS, the *DataWriter* history is only available for *DataReader* throughout the DataWriter's life. This means that the history does not persist between DataWriter initializations and therefore it is on an empty state on DataWriter creation. Similarly, the DataReader history does not persist the DataReader's life, thus also being empty on DataReader creation. However, *eProsima Fast DDS* offers the possibility to configure the DataWriter's history to be stored in a persistent database, so that the DataWriter can load its history from it on creation. Furthermore, DataReaders can be configured to store the last notified change in the database, so that they can recover their state on creation.

This mechanism allows recovering a previous state on starting the Data Distribution Service, thus adding robustness to applications in the case of, for example, unexpected shutdowns. Configuring the persistence service, DataWriters and DataReaders can resume their operation from the state in which they were when the shutdown occurred.

Note: Mind that DataReaders do not store their history into the database, but rather the last notified change from the DataWriter. This means that they will resume operation where they left, but they will not have the previous information, since that was already notified to the application.

6.21.1 Configuration

The configuration of the persistence service is accomplished by setting of the appropriate DataWriter and DataReader *DurabilityQosPolicy*, and by specifying the suitable properties for each entity's (*DomainParticipant*, DataWriter, or DataReader) *PropertyPolicyQos*.

- For the *Persistence Service* to have any effect, the *DurabilityQosPolicyKind* needs to be set to *TRANSIENT_DURABILITY_QOS*.
- A persistence identifier (*Guid_t*) must be set for the entity using the property `dds.persistence.guid`. This identifier is used to load the appropriate data from the database, and also to synchronize DataWriter and DataReader between restarts. The GUID consists of 16 bytes separated into two groups:
  - The first 12 bytes correspond to the *GuidPrefix_t*.
  - The last 4 bytes correspond to the *EntityId_t*.

The persistence identifier is specified using a string of 12 dot-separated bytes, expressed in hexadecimal base, followed by a vertical bar separator (|) and another 4 dot-separated bytes, also expressed in hexadecimal base (see *Example*). For selecting an appropriate GUID for the DataReader and DataWriter, please refer to RTPS standard (section *9.3.1 The Globally Unique Identifier (GUID)*).

- A persistence plugin must be configured for managing the database using property `dds.persistence.plugin` (see *PERSISTENCE:SQLITE3 built-in plugin*):

6.21.2 PERSISTENCE:SQLITE3 built-in plugin

This plugin provides persistence through a local database file using *SQLite3* API. To activate the plugin, `dds.persistence.plugin` property must be added to the PropertyPolicyQos of the DomainParticipant, DataWriter, or DataReader with value `builtin.SQLITE3`. Furthermore, `dds.persistence.sqlite3.filename` property must be added to the entities PropertyPolicyQos, specifying the database file name. These properties are summarized in the following table:

Table 12: **Persistence::SQLITE3 configuration properties**

Property name	Property value
`dds.persistence.plugin`	`builtin.SQLITE3`
`dds.persistence.sqlite3.filename`	Name of the file used for persistent storage. Default value: `persistence.db`

Note: To avoid undesired delays caused by concurrent access to the SQLite3 database, it is advisable to specify a different database file for each DataWriter and DataReader.

Important: The plugin set in the PropertyPolicyQos of DomainParticipant only applies if that of the DataWriter/DataReader does no exist or is invalid.

6.21.3 Example

This example shows how to configure the persistence service using *PERSISTENCE:SQLITE3 built-in plugin* plugin both from C++ and using *eProsima Fast DDS* XML profile files (see *XML profiles*).

C++

```
/*
 * In order for this example to be self-contained, all the entities are created␣
↪programatically, including the data
 * type and type support. This has been done using Fast DDS Dynamic Types API, but it␣
↪could be substituted with a
 * Fast DDS-Gen generated type support if an IDL file is available. The Dynamic Type␣
↪created here is the equivalent
 * of the following IDL:
 *
 *     struct persistence_topic_type
 *     {
 *         unsigned long index;
 *         string message;
```

(continues on next page)

(continued from previous page)

```
 *     };
 */

// Configure persistence service plugin for DomainParticipant
DomainParticipantQos pqos;
pqos.properties().properties().emplace_back("dds.persistence.plugin", "builtin.SQLITE3");
pqos.properties().properties().emplace_back("dds.persistence.sqlite3.filename",
↪"persistence.db");
DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);

/
↪*****************************************************************************************************
* CREATE TYPE AND TYPE SUPPORT
*******************************************************************************************************
* This part could be replaced if IDL file and Fast DDS-Gen are available.
* The type is created with name "persistence_topic_type"
* Additionally, create a data object and populate it, just to show how to do it
*******************************************************************************************************
↪
// Create a struct builder for a type with name "persistence_topic_type"
const std::string topic_type_name = "persistence_topic_type";
eprosima::fastrtps::types::DynamicTypeBuilder_ptr struct_type_builder(
    eprosima::fastrtps::types::DynamicTypeBuilderFactory::get_instance()->create_struct_
↪builder());
struct_type_builder->set_name(topic_type_name);

// The type consists of two members, and index and a message. Add members to the struct.
struct_type_builder->add_member(0, "index",
        eprosima::fastrtps::types::DynamicTypeBuilderFactory::get_instance()->create_
↪uint32_type());
struct_type_builder->add_member(1, "message",
        eprosima::fastrtps::types::DynamicTypeBuilderFactory::get_instance()->create_
↪string_type());

// Build the type
eprosima::fastrtps::types::DynamicType_ptr dyn_type_ptr = struct_type_builder->build();

// Create type support and register the type
TypeSupport type_support(new eprosima::fastrtps::types::DynamicPubSubType(dyn_type_ptr));
type_support.register_type(participant);

// Create data sample a populate data. This is to be used when calling `writer->write()`
eprosima::fastrtps::types::DynamicData* dyn_helloworld;
dyn_helloworld = eprosima::fastrtps::types::DynamicDataFactory::get_instance()->create_
↪data(dyn_type_ptr);
dyn_helloworld->set_uint32_value(0, 0);
dyn_helloworld->set_string_value("HelloWorld", 1);
/
↪*****************************************************************************************************
* END CREATE TYPE AND TYPE SUPPORT
*******************************************************************************************************
↪
```

(continues on next page)

(continued from previous page)

```
// Create a topic
Topic* topic = participant->create_topic("persistence_topic_name", topic_type_name,␣
↪TOPIC_QOS_DEFAULT);

// Create a publisher and a subscriber with default QoS
Publisher* publisher = participant->create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);
Subscriber* subscriber = participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT, nullptr);

// Configure DataWriter's durability and persistence GUID so it can use the persistence␣
↪service
DataWriterQos dwqos = DATAWRITER_QOS_DEFAULT;
dwqos.durability().kind = TRANSIENT_DURABILITY_QOS;
dwqos.properties().properties().emplace_back("dds.persistence.guid",
        "77.72.69.74.65.72.5f.70.65.72.73.5f|67.75.69.64");
DataWriter* writer = publisher->create_datawriter(topic, dwqos);

// Configure DataReaders's durability and persistence GUID so it can use the persistence␣
↪service
DataReaderQos drqos = DATAREADER_QOS_DEFAULT;
drqos.durability().kind = TRANSIENT_DURABILITY_QOS;
drqos.properties().properties().emplace_back("dds.persistence.guid",
        "72.65.61.64.65.72.5f.70.65.72.73.5f|67.75.69.64");
DataReader* reader = subscriber->create_datareader(topic, drqos);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <!-- DomainParticipant configuration -->
        <participant profile_name="persistence_service_participant">
            <rtps>
                <propertiesPolicy>
                    <properties>
                        <!-- Select persistence plugin -->
                        <property>
                            <name>dds.persistence.plugin</name>
                            <value>builtin.SQLITE3</value>
                        </property>
                        <!-- Database file name -->
                        <property>
                            <name>dds.persistence.sqlite3.filename</name>
                            <value>persistence_service.db</value>
                        </property>
                    </properties>
                </propertiesPolicy>
            </rtps>
        </participant>

        <!-- DataWriter configuration -->
        <data_writer profile_name="persistence_service_data_writer">
```

(continues on next page)

(continued from previous page)

```
            <qos>
                <!-- Set durability to TRANSIENT_DURABILITY_QOS -->
                <durability>
                    <kind>TRANSIENT</kind>
                </durability>
            </qos>
            <propertiesPolicy>
                <properties>
                    <!-- Persistence GUID -->
                    <property>
                        <name>dds.persistence.guid</name>
                        <value>77.72.69.74.65.72.5f.70.65.72.73.5f|67.75.69.64</value>
                    </property>
                </properties>
            </propertiesPolicy>
        </data_writer>

        <data_reader profile_name="persistence_service_data_reader">
            <qos>
                <!-- Set durability to TRANSIENT_DURABILITY_QOS -->
                <durability>
                    <kind>TRANSIENT</kind>
                </durability>
            </qos>
            <propertiesPolicy>
                <properties>
                    <!-- Persistence GUID -->
                    <property>
                        <name>dds.persistence.guid</name>
                        <value>72.65.61.64.65.72.5f.70.65.72.73.5f|67.75.69.64</value>
                    </property>
                </properties>
            </propertiesPolicy>
        </data_reader>
    </profiles>
</dds>
```

Note: For instructions on how to create DomainParticipants, DataReaders, and DataWriters, please refer to *Profile based creation of a DomainParticipant*, *Profile based creation of a DataWriter*, and *Profile based creation of a DataReader* respectively.

6.22 Security

The DDS Security specification includes five security builtin plugins.

1. Authentication plugin: *DDS:Auth:PKI-DH*. This plugin provides authentication for each *DomainParticipant* joining a DDS Domain using a trusted *Certificate Authority* (CA). Support mutual authentication between DomainParticipants and establish a shared secret.
2. Access Control plugin: *DDS:Access:Permissions*. This plugin provides access control to DomainParticipants which perform protected operations.
3. Cryptographic plugin: *DDS:Crypto:AES-GCM-GMAC*. This plugin provides authenticated encryption using Advanced Encryption Standard (AES) in Galois Counter Mode (AES-GCM).
4. Logging plugin: *DDS:Logging:DDS_LogTopic*. This plugin logs security events.
5. Data Tagging: DDS:Tagging:DDS_Discovery. This plugin enables the addition of security labels to the data. Thus it is possible to specify classification levels of the data. In the DDS context it can be used as a complement to access control, creating an access control based on data tagging; for message prioritization; and to prevent its use by the middleware to be used instead by the application or service.

Note: Currently the DDS:Tagging:DDS_Discovery plugin is not implemented in Fast DDS. Its implementation is expected for future release of Fast DDS.

In compliance with the DDS Security specification, Fast DDS provides secure communication by implementing pluggable security at three levels: a) DomainParticipants authentication (DDS:Auth:PKI-DH), b) access control of Entities (DDS:Access:Permissions), and c) data encryption (DDS:Crypto:AES-GCM-GMAC). Furthermore, for the monitoring of the security plugins and logging relevant events, Fast DDS implements the logging plugin (DDS:Logging:DDS_LogTopic).

By default, Fast DDS does not compile any security support, but it can be activated adding `-DSECURITY=ON` at CMake configuration step. For more information about Fast DDS compilation, see *Linux installation from sources* and *Windows installation from sources*.

Security plugins can be activated through the *DomainParticipantQos* properties. A `Property` is defined by its name (`std::string`) and its value (`std::string`).

Warning: For the full understanding of this documentation it is required the user to have basic knowledge of network security since terms like Certificate Authority (CA), Public Key Infrastructure (PKI), and Diffie-Hellman encryption protocol are not explained in detail. However, it is possible to configure basic system security settings, i.e. authentication, access control and encryption, to Fast DDS without this knowledge.

The following sections describe how to configure each of these properties to set up the Fast DDS security plugins.

6.22.1 Authentication plugin: DDS:Auth:PKI-DH

This is the starting point for all the security mechanisms. The authentication plugin provides the mechanisms and operations required for *DomainParticipants* authentication at discovery. If the security module was activated at Fast DDS compilation, when a DomainParticipant is either locally created or discovered, it needs to be authenticated in order to be able to communicate in a DDS Domain. Therefore, when a DomainParticipant detects a remote DomainParticipant, both try to authenticate themselves using the activated authentication plugin. If the authentication process finishes successfully both DomainParticipant match and the discovery mechanism continues. On failure, the remote DomainParticipant is rejected.

The authentication plugin implemented in Fast DDS is referred to as "DDS:Auth:PKI-DH", in compliance with the DDS Security specification. The DDS:Auth:PKI-DH plugin uses a trusted *Certificate Authority* (CA) and the ECDSA Digital Signature Algorithms to perform the mutual authentication. It also establishes a shared secret using Elliptic Curve Diffie-Hellman (ECDH) Key Agreement Methods. This shared secret can be used by other security plugins as *Cryptographic plugin: DDS:Crypto:AES-GCM-GMAC*.

The DDS:Auth:PKI-DH authentication plugin, can be activated setting the *DomainParticipantQos* `properties()` `dds.sec.auth.plugin` with the value `builtin.PKI-DH`. The following table outlines the properties used for the DDS:Auth:PKI-DH plugin configuration.

Property name	Property value
identity_ca	URI to the X.509 v3 certificate of the Identity CA in PEM format. Supported URI schemes: file.
identity_certificate	URI to an X.509 v3 certificate signed by the Identity CA in PEM format containing the signed public key for the Participant. Supported URI schemes: file.
identity_crl *(optional)*	URI to a X.509 Certificate Revocation List (CRL). Supported URI schemes: file.
private_key	URI to access the private Private Key for the Participant. Supported URI schemes: file, *PKCS#11*.
password *(optional)*	A password used to decrypt the *private_key*. If the *password* property is not present, then the value supplied in the *private_key* property must contain the decrypted private key. The *password* property is ignored if the *private_key* is given in PKCS#11 scheme.

Note: All listed properties have "dds.sec.auth.builtin.PKI-DH." prefix. For example: `dds.sec.auth.builtin.PKI-DH.identity_ca`.

The following is an example of how to set the properties of DomainParticipantQoS for the DDS:Auth:PKI-DH plugin configuration.

C++

```
DomainParticipantQos pqos;

// Activate DDS:Auth:PKI-DH plugin
pqos.properties().properties().emplace_back("dds.sec.auth.plugin",
        "builtin.PKI-DH");

// Configure DDS:Auth:PKI-DH plugin
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_ca",
    "file://maincacert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_certificate",
    "file://partcert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_crl",
    "file://crl.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.private_key",
    "file://partkey.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.password",
    "domainParticipantPassword");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_auth_plugin_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.plugin</name>
                    <value>builtin.PKI-DH</value>
                </property>
                <!-- Configure DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_ca</name>
                    <value>file://maincacert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_certificate</name>
                    <value>file://partcert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_crl</name>
                    <value>file://crl.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.private_key</name>
                    <value>file://partkey.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.password</name>
                    <value>domainParticipantPassword</value>
                </property>
            </properties>
        </propertiesPolicy>
```

Generation of X.509 certificates

An X.509 digital certificate is a document that has been encrypted and/or digitally signed according to RFC 5280. The X.509 certificate refers to the Public Key Infrastructure (PKI) certificate of the IETF , and specifies the standard formats for public-key certificates and a certification route validation algorithm. A simple way to generate these certificates for a proprietary PKI structure is through the OpenSSL toolkit. This section explains how to build a certificate infrastructure from the trusted CA certificate to the end-entity certificate, i.e. the DomainParticipant.

Generating the CA certificate for self-signing

First, since multiple certificates will need to be issued, one for each of the DomainParticipants, a dedicated CA is set up, and the CA's certificate is installed as the root key of all DomainParticipants. Thus, the DomainParticipants will accept all certificates issued by our own CA. To create a proprietary CA certificate, a configuration file must first be written with the CA information. An example of the CA configuration file is shown below. The OpenSSL commands shown in this example are compatible with both Linux and Windows Operating Systems (OS). However, all other commands are only compatible with Linux OS.

```
# File: maincaconf.cnf
# OpenSSL example Certificate Authority configuration file

####################################################################
[ ca ]
default_ca = CA_default # The default ca section

####################################################################
[ CA_default ]

dir = . # Where everything is kept
certs = $dir/certs # Where the issued certs are kept
crl_dir = $dir/crl # Where the issued crl are kept
database = $dir/index.txt # database index file.
unique_subject = no # Set to 'no' to allow creation of
                    # several ctificates with same subject.
new_certs_dir = $dir

certificate = $dir/maincacert.pem # The CA certificate
serial = $dir/serial # The current serial number
crlnumber = $dir/crlnumber # the current crl number
                           # must be commented out to leave a V1 CRL
crl = $dir/crl.pem # The current CRL
private_key = $dir/maincakey.pem # The private key
RANDFILE = $dir/private/.rand # private random number file

name_opt = ca_default # Subject Name options
cert_opt = ca_default # Certificate field options

default_days= 1825 # how long to certify for
default_crl_days = 30 # how long before next CRL
default_md = sha256 # which md to use.
preserve = no # keep passed DN ordering

policy = policy_match
```

(continues on next page)

(continued from previous page)

```
# For the CA policy
[ policy_match ]
countryName = match
stateOrProvinceName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

[ req ]
prompt = no
#default_bits = 1024
#default_keyfile = privkey.pem
distinguished_name= req_distinguished_name
#attributes = req_attributes
#x509_extensions = v3_ca # The extentions to add to the self signed cert
string_mask = utf8only

[ req_distinguished_name ]
countryName = ES
stateOrProvinceName = MA
localityName = Tres Cantos
0.organizationName = eProsima
commonName = eProsima Main Test CA
emailAddress = mainca@eprosima.com
```

After writing the configuration file, next commands generate the certificate using the Elliptic Curve Digital Signature Algorithm (ECDSA).

```
openssl ecparam -name prime256v1 > ecdsaparam

openssl req -nodes -x509 \
  -days 3650 \
  -newkey ec:ecdsaparam \
  -keyout maincakey.pem \
  -out maincacert.pem \
  -config maincaconf.cnf
```

Generating the DomainParticipant certificate

As was done for the CA, a DomainParticipant certificate configuration file needs to be created first.

```
# File: partconf.cnf

prompt = no
string_mask = utf8only
distinguished_name = req_distinguished_name

[ req_distinguished_name ]
countryName = ES
stateOrProvinceName = MA
localityName = Tres Cantos
organizationName = eProsima
emailAddress = example@eprosima.com
commonName = DomainParticipantName
```

After writing the DomainParticipant certificate configuration file, next commands generate the X.509 certificate, using ECDSA, for a DomainParticipant.

```
openssl ecparam -name prime256v1 > ecdsaparam

openssl req -nodes -new \
  -newkey ec:ecdsaparam \
  -config partconf.cnf \
  -keyout partkey.pem \
  -out partreq.pem

openssl ca -batch -create_serial \
  -config maincaconf.cnf \
  -days 3650 \
  -in partreq.pem \
  -out partcert.pem
```

Generating the Certificate Revocation List (CRL)

Finally, the CRL is created. This is a list of the X.509 certificates revoked by the certificate issuing CA before they reach their expiration date. Any certificate that is on this list will no longer be trusted. To create a CRL using OpenSSL just run the following commands.

```
echo -ne '00' > crlnumber

openssl ca -gencrl \
  -config maincaconf.cnf \
  -cert maincacert.pem \
  -keyfile maincakey.pem \
  -out crl.pem
```

As an example, below is shown how to add the X.509 certificate of a DomainParticipant to the CRL.

```
openssl ca \
  -config maincaconf.cnf \
  -cert maincacert.pem \
  -keyfile maincakey.pem \
  -revoke partcert.pem

openssl ca -gencrl \
  -config maincaconf.cnf \
  -cert maincacert.pem \
  -keyfile maincakey.pem \
  -out crl.pem
```

6.22.2 Access control plugin: DDS:Access:Permissions

The access control plugin provides the mechanisms and operations required for validating the *DomainParticipant* permissions. If the security module was activated at Fast DDS compilation, after a remote DomainParticipant is authenticated, its permissions need to be validated and enforced.

Access rights that each DomainParticipant has over a resource are defined using the access control plugin. For the proper functioning of a DomainParticipant in a DDS Domain, the DomainParticipant must be authorized to operate in that specific domain. The DomainParticipant is responsible for creating the *DataWriters* and *DataReaders* that communicate over a certain *Topic*. Hence, a DomainParticipant must have the permissions needed to create a Topic, to publish through its DataWriters under defined Topics, and to subscribe via its DataReaders to other Topics. Access control plugin can configure the Cryptographic plugin as its usage is based on the DomainParticipant's permissions.

The authentication plugin implemented in Fast DDS is referred to as "DDS:Access:Permissions", in compliance with the DDS Security specification. This plugin is explained in detail below.

This builtin plugin provides access control using a permissions document signed by a trusted CA. The DDS:Access:Permissions plugin requires three documents for its configuration which contents are explained in detail below.

1. The Permissions CA certificate.
2. The Domain governance signed by the Permissions CA.
3. The DomainParticipant permissions signed by the Permissions CA.

The DDS:Access:Permissions authentication plugin, can be activated setting the *DomainParticipantQos properties()* `dds.sec.access.plugin` with the value `builtin.Access-Permissions`. The following table outlines the properties used for the DDS:Access:Permissions plugin configuration.

Property name	Property value
permissions_ca	URI to the X509 certificate of the Permissions CA. Supported URI schemes: file. The file schema shall refer to an X.509 v3 certificate in PEM format.
governance	URI to shared Governance Document signed by the Permissions CA in S/MIME format. Supported URI schemes: file.
permissions	URI to the Participant permissions document signed by the Permissions CA in S/MIME format. Supported URI schemes: file.

Note: All listed properties have "dds.sec.access.builtin.Access-Permissions." prefix. For example: `dds.sec.access.builtin.Access-Permissions.permissions_ca`.

The following is an example of how to set the properties of *DomainParticipantQos* for the DDS:Access:Permissions configuration.

C++

```
DomainParticipantQos pqos;

// Activate DDS:Access:Permissions plugin
pqos.properties().properties().emplace_back("dds.sec.access.plugin",
        "builtin.Access-Permissions");

// Configure DDS:Access:Permissions plugin
pqos.properties().properties().emplace_back(
    "dds.sec.access.builtin.Access-Permissions.permissions_ca",
    "file://certs/maincacert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.access.builtin.Access-Permissions.governance",
    "file://certs/governance.smime");
pqos.properties().properties().emplace_back(
    "dds.sec.access.builtin.Access-Permissions.permissions",
    "file://certs/permissions.smime");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_access_control_plugin_xml_
↪profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.plugin</name>
                    <value>builtin.Access-Permissions</value>
                </property>
                <!-- Configure DDS:Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.permissions_ca</
↪name>
                    <value>file://maincacet.pem</value>
                </property>
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.governance</name>
                    <value>file://governance.smime</value>
                </property>
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.permissions</name>
                    <value>file://permissions.smime</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Permissions CA Certificate

This is an X.509 certificate that contains the Public Key of the CA that will be used to sign the *Domain Governance Document* and the *DomainParticipant Permissions Document*.

Domain Governance Document

Domain Governance document is an XML document that specifies the mechanisms to secure the DDS Domain. It shall be signed by the Permissions CA in S/MIME format. The XML scheme of this document is defined in *Domain Governance XSD*. The following is an example of the Domain Governance XML file contents.

```
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="omg_shared_ca_domain_governance.xsd">
    <domain_access_rules>
        <domain_rule>
            <domains>
                <id_range>
                    <min>0</min>
                    <max>230</max>
                </id_range>
            </domains>
            <allow_unauthenticated_participants>false</allow_unauthenticated_
↪participants>
            <enable_join_access_control>true</enable_join_access_control>
            <discovery_protection_kind>ENCRYPT</discovery_protection_kind>
            <liveliness_protection_kind>ENCRYPT</liveliness_protection_kind>
            <rtps_protection_kind>ENCRYPT</rtps_protection_kind>
            <topic_access_rules>
                <topic_rule>
                    <topic_expression>HelloWorldTopic</topic_expression>
                    <enable_discovery_protection>true</enable_discovery_protection>
                    <enable_liveliness_protection>false</enable_liveliness_protection>
                    <enable_read_access_control>true</enable_read_access_control>
                    <enable_write_access_control>true</enable_write_access_control>
                    <metadata_protection_kind>ENCRYPT</metadata_protection_kind>
                    <data_protection_kind>ENCRYPT</data_protection_kind>
                </topic_rule>
            </topic_access_rules>
        </domain_rule>
    </domain_access_rules>
</dds>
```

The Governance XSD file and the Governance XML example can also be downloaded from the eProsima Fast DDS Github repository.

Domain Rules

It allows the application of rules to the DDS Domain. The domain rules define aspects of the DDS Domain such as:

- Whether the discovery data should be protected and the type of protection: MAC only or encryption followed by MAC.
- Whether the whole RTPS message should be encrypted.
- Whether the liveliness of the messages should be protected.
- Whether a non-authenticated DomainParticipant can access or not to the unprotected discovery metatraffic and unprotected Topics.
- Whether an authenticated DomainParticipant can access the domain without evaluating the access control policies.
- Whether discovery information on a certain Topic should be sent with secure DataWriters.
- Whether or not the access to Topics should be restricted to DomainParticipants with the appropriate permission to read them.
- Whether the metadata sent on a certain Topic should be protected and the type of protection.
- Whether payload data on a certain Topic should be protected and the type of protection.

The domain rules are evaluated in the same order as they appear in the document. A rule only applies to a particular DomainParticipant if the domain section matches the DDS `Domain_Id` to which the DomainParticipant belongs. If multiple rules match, the first rule that matches is the only one that applies. Each domain rule is delimited by the `<domain_rule>` XML element tag.

Some domain rules may have an additional configuration if enabled. This configuration defines the level of protection that the rule applies to the domain:

- **NONE**: no cryptographic transformation is applied.
- **SIGN**: cryptographic transformation based on Message Authentication Code (MAC) is applied, without additional encryption.
- **ENCRYPT**: the data is encrypted and followed by a MAC computed on the ciphertext, also known as Encrypt-then-MAC.

The following table summarizes the elements and sections that each domain rule may contain.

Type	Name	XML element tag	Values
Element	*Allow Unauthenticated Participants*	`<allow_unauthenticated_participants>`	`false`
			`true`
	Enable Join Access Control	`<enable_join_access_control>`	`false`
			`true`
	Discovery Protection Kind	`<discovery_protection_kind>`	`SIGN`
			`ENCRYPT`
			`NONE`
	Liveliness Protection Kind	`<liveliness_protection_kind>`	`SIGN`
			`ENCRYPT`
			`NONE`
	RTPS Protection Kind	`<rtps_protection_kind>`	`SIGN`
			`ENCRYPT`
			`NONE`
Section	*Domains*	`<domains>`	`<domains>`
	Topic Access Rules	`<topic_access_rules>`	`<topic_rule>`

The following describes the possible configurations of each of the elements and sections listed above that are contained in the domain rules.

Domains

This element is delimited by the <domains> XML element tag. The value in this element identifies the collection of DDS Domains to which the rule applies. The <domains> element can contain:

- A single domain identifier:

```
<domains>
    <id>1</id>
</domains>
```

- A range of domain identifiers:

```
<domains>
    <id_range>
        <min>1</min>
        <max>10</max>
    </id_range>
</domains>
```

Or a combination of both, a list of domain identifiers and ranges of domain identifiers.

Allow Unauthenticated Participants

This element is delimited by the <allow_unauthenticated_participants> XML element tag. It indicates whether the matching of a DomainParticipant with a remote DomainParticipant requires authentication. The possible values for this element are:

- `false`: the DomainParticipant shall enforce the authentication of remote *DomainParticipants* and disallow matching those that cannot be successfully authenticated.
- `true`: the DomainParticipant shall allow matching other DomainParticipants (event if the remote DomainParticipant cannot authenticate) as long as there is not an already valid authentication with the same DomainParticipant's GUID.

In accordance with the DDS Security specification, the following premises should be considered:

- Topics protected with `enable_read_access_control` or `enable_write_access_control` will not communicate regardless of the allow_unauthenticated_participants flag value.
- If *RTPS Protection Kind* is not `NONE` and *Allow Unauthenticated Participants* is enabled, the entity creation will fail with an error.
- Authentication is always attempted first regardless of <allow_unauthenticated_participants> configuration.

Enable Join Access Control

This element is delimited by the <enable_join_access_control> XML element tag. Indicates whether the matching of the participant with a remote DomqainParticipant requires authorization by the DDS:Access:Permissions plugin. Its possible values are:

- false: the DomainParticipant shall not check the permissions of the authenticated remote DomainParticipant.
- true: the DomainParticipant shall check the permissions of the authenticated remote DomainParticipant.

Discovery Protection Kind

This element is delimited by the <discovery_protection_kind> XML element tag. Indicates whether the secure channel of the endpoint discovery phase needs to be encrypted. The possible values are:

- NONE: the secure channel shall not be protected.
- SIGN: the secure channel shall be protected by MAC.
- ENCRYPT: the secure channel shall be encrypted.

Liveliness Protection Kind

This element is delimited by the <liveliness_protection_kind> XML element tag. Indicates whether the secure channel of the liveliness mechanism needs to be encrypted. The possible values are:

- NONE: the secure channel shall not be protected.
- SIGN: the secure channel shall be protected by MAC.
- ENCRYPT: the secure channel shall be encrypted.

RTPS Protection Kind

This element is delimited by the <rtps_protection_kind> XML element tag. Indicates whether the whole RTPS Message needs to be encrypted. The possible values are:

- NONE: whole RTPS Messages shall not be protected.
- SIGN: whole RTPS Messages shall be protected by MAC.
- ENCRYPT: whole RTPS Messages shall be encrypted.

Topic Rule

This element is delimited by the <topic_rule> XML element tag and appears within the Topic Access Rules Section whose XML element tag is <topic_access_rules>. The following table summarizes the elements and sections that each domain rule may contain.

Elements	XML element tag	Values
Topic expression	<topic_expression>	Topic name
Enable Discovery Protection	<enable_discovery_protection>	false
		true
Enable Liveliness Protection	<enable_liveliness_protection>	false
		true
Enable Read Access Control	<enable_read_access_control>	false
		true
Enable Write Access Control	<enable_write_access_control>	false
		true
Metadata protection Kind	<metadata_protection_kind>	SIGN
		ENCRYPT
		NONE
Data protection Kind	<data_protection_kind>	SIGN
		ENCRYPT
		NONE

The topic expression within the rules selects a set of Topic names. The rule applies to any *DataReader* or *DataWriter* associated with a *Topic* whose name matches the Topic expression name. The topic access rules are evaluated in the same order as they appear within the <topic_access_rules> section. If multiple rules match, the first rule that matches is the only one that applies. If no matching <topic_rule> is found, the entity creation will fail.

Topic expression

This element is delimited by the <topic_expression> XML element tag. The value in this element identifies the set of Topic names to which the rule applies. The rule applies to any DataReader or DataWriter associated with a Topic whose name matches the value.

The Topic name expression syntax and matching shall use the syntax and rules of the POSIX `fnmatch()` function as specified in IEEE 1003.1-2017.

Enable Discovery Protection

This element is delimited by the <enable_discovery_protection> XML element tag. Indicates whether the entity related discovery information shall go through the secure channel of endpoint discovery phase.

- `false`: the entity discovery information shall be sent by an unsecured channel of discovery.
- `true`: the information shall be sent by the secure channel.

Enable Liveliness Protection

This element is delimited by the <enable_liveliness_protection> XML element tag. Indicates whether the entity related liveliness information shall go through the secure channel of liveliness mechanism.

- `false`: the entity liveliness information shall be sent by an unsecured channel of liveliness.
- `true`: the information shall be sent by the secure channel.

Enable Read Access Control

This element is delimited by the <enable_read_access_control> XML element tag. Indicates whether read access to the Topic is protected.

- false: then local Subscriber creation and remote Subscriber matching can proceed without further access-control mechanisms imposed.
- true: they shall be checked using Access control plugin.

Enable Write Access Control

This element is delimited by the <enable_write_access_control> XML element tag. Indicates whether write access to the Topic is protected.

- false: then local Publisher creation and remote Publisher matching can proceed without further access-control mechanisms imposed.
- true: they shall be checked using Access control plugin.

Metadata Protection Kind

This element is delimited by the <metadata_protection_kind> XML element tag. Indicates whether the entity's RTPS submessages shall be encrypted by the Cryptographic plugin.

- NONE: shall not be protected.
- SIGN: shall be protected by MAC.
- ENCRYPT: shall be encrypted.

Data Protection Kind

This element is delimited by the <data_protection_kind> XML element tag. Indicates whether the data payload shall be encrypted by the Cryptographic plugin.

- NONE: shall not be protected.
- SIGN: shall be protected by MAC.
- ENCRYPT: shall be encrypted.

Domain Governance XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="dds" type="DomainAccessRulesNode" />
    <xs:complexType name="DomainAccessRulesNode">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="domain_access_rules"
                type="DomainAccessRules" />
        </xs:sequence>
    </xs:complexType>
```

(continues on next page)

(continued from previous page)

```
    <xs:complexType name="DomainAccessRules">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="domain_rule" type="DomainRule" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DomainRule">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="domains" type="DomainIdSet" />
            <xs:element name="allow_unauthenticated_participants"
                type="xs:boolean" />
            <xs:element name="enable_join_access_control"
                type="xs:boolean" />
            <xs:element name="discovery_protection_kind"
                type="ProtectionKind" />
            <xs:element name="liveliness_protection_kind"
                type="ProtectionKind" />
            <xs:element name="rtps_protection_kind"
                type="ProtectionKind" />
            <xs:element name="topic_access_rules"
                type="TopicAccessRules" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DomainIdSet">
        <xs:choice minOccurs="1" maxOccurs="unbounded">
            <xs:element name="id" type="DomainId" />
            <xs:element name="id_range" type="DomainIdRange" />
        </xs:choice>
    </xs:complexType>
    <xs:simpleType name="DomainId">
        <xs:restriction base="xs:nonNegativeInteger" />
    </xs:simpleType>
    <xs:complexType name="DomainIdRange">
        <xs:choice>
            <xs:sequence>
                <xs:element name="min" type="DomainId" />
                <xs:element name="max" type="DomainId" minOccurs="0" />
            </xs:sequence>
            <xs:element name="max" type="DomainId" />
        </xs:choice>
    </xs:complexType>
    <xs:simpleType name="ProtectionKind">
        <xs:restriction base="xs:string">
            <xs:enumeration value="ENCRYPT_WITH_ORIGIN_AUTHENTICATION" />
            <xs:enumeration value="SIGN_WITH_ORIGIN_AUTHENTICATION" />
            <xs:enumeration value="ENCRYPT" />
            <xs:enumeration value="SIGN" />
            <xs:enumeration value="NONE" />
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="BasicProtectionKind">
        <xs:restriction base="ProtectionKind">
            <xs:enumeration value="ENCRYPT" />
```

(continues on next page)

(continued from previous page)

```
            <xs:enumeration value="SIGN" />
            <xs:enumeration value="NONE" />
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="TopicAccessRules">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="topic_rule" type="TopicRule" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TopicRule">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="topic_expression" type="TopicExpression" />
            <xs:element name="enable_discovery_protection"
                type="xs:boolean" />
            <xs:element name="enable_liveliness_protection"
                type="xs:boolean" />
            <xs:element name="enable_read_access_control"
                type="xs:boolean" />
            <xs:element name="enable_write_access_control"
                type="xs:boolean" />
            <xs:element name="metadata_protection_kind"
                type="ProtectionKind" />
            <xs:element name="data_protection_kind"
                type="BasicProtectionKind" />
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="TopicExpression">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
</xs:schema>
```

Back to the *Domain Governance Document*.

DomainParticipant Permissions Document

The permissions document is an XML file which contains the permissions of a DomainParticipant and binds them to the DomainParticipant distinguished name defined in the DDS:Auth:PKI-DH plugin. The permissions document shall be signed by the Permissions CA in S/MIME format. The XML scheme of this document is defined in *DomainParticipant Permissions XSD*. The following is an example of the DomainParticipant Permissions XML file contents.

```
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.omg.org/spec/DDS-Security/20170801/omg_
↪shared_ca_permissions.xsd">
    <permissions>
        <grant name="ParticipantPermissions">
            <subject_name>emailAddress=example@eprosima.com, CN=DomainParticipantName,␣
↪O=eProsima, ST=MA, C=ES</subject_name>
            <validity>
                <not_before>2013-06-01T13:00:00</not_before>
                <not_after>2038-06-01T13:00:00</not_after>
            </validity>
```

(continues on next page)

(continued from previous page)

```
            <allow_rule>
                <domains>
                    <id_range>
                        <min>0</min>
                        <max>230</max>
                    </id_range>
                </domains>
                <publish>
                    <topics>
                        <topic>HelloWorldTopic</topic>
                    </topics>
                </publish>
            </allow_rule>
            <default>DENY</default>
        </grant>

        <!-- Fill the subject name with the information specified in the Participant␣
↪certificate -->
        <grant name="OtherParticipantPermissions">
            <subject_name> emailAddress=xxx@eprosima.com, CN=xxx, OU=x, O=x, ST=XX, C=XX
↪</subject_name>
            <validity>
                <not_before>2013-06-01T13:00:00</not_before>
                <not_after>2038-06-01T13:00:00</not_after>
            </validity>
            <allow_rule>
                <domains>
                    <id_range>
                        <min>0</min>
                        <max>230</max>
                    </id_range>
                </domains>
                <subscribe>
                    <topics>
                        <topic>HelloWorldTopic</topic>
                    </topics>
                </subscribe>
            </allow_rule>
            <default>DENY</default>
        </grant>
    </permissions>
</dds>
```

The Permissions XSD file and the Permissions XML example can also be downloaded from the eProsima Fast DDS Github repository.

Grant Section

This section is delimited by the <grant> XML element tag. Each grant section contains three sections:

- Subject name
- Validity
- Rules

Subject name

This section is delimited by XML element <subject_name>. The subject name identifies the DomainParticipant to which the permissions apply. Each subject name can only appear in a single <permissions> section within the XML Permissions document. The contents of the subject name element shall be the X.509 subject name of the DomainParticipant that was given in the authorization X.509 Certificate.

Validity

This section is delimited by the XML element <validity>. It reflects the valid dates for the permissions.

Rules

This section contains the permissions assigned to the DomainParticipant. The rules are applied in the same order that appears in the document. If the criteria for the rule matched the Domain join, publish or subscribe operation that is being attempted, then the *allow* or *deny* decision is applied. If the criteria for a rule does not match the operation being attempted, the evaluation shall proceed to the next rule. If all rules have been examined without a match, then the decision specified by the <default> rule is applied. The default rule, if present, must appear after all *allow* and *deny* rules. If the default rule is not present, the implied default decision is DENY.

For the grant to match there shall be a match of the topics and partitions criteria.

Allow rules are delimited by the XML element <allow_rule>. Deny rules are delimited by the XML element``<deny_rule>``. Both contain the same element children.

Domains Section

This section is delimited by the XML element <domains>. The value in this element identifies the collection of DDS Domains to which the rule applies. The syntax is the same as for the *Domains* of the *Domain Governance Document*.

Format of the Allowed/Denied Actions sections

The sections for each of the three actions have a similar format. The only difference is the name of the XML element used to delimit the action:

Action	XML element tag
Allow/Deny Publish	<publish>
Allow/Deny Subscribe	<subscribe>
Allow/Deny Relay	<relay>

Each action contains two conditions.

- Allowed/Denied *Topics Condition*
- Allowed/Denied *Partitions Condition*

Topics Condition

This section is delimited by the <topics> XML element. It defines the Topic names that must be matched for the allow/deny rule to apply. Topic names may be given explicitly or by means of Topic name expressions. Each explicit topic name or Topic name expressions appears separately in a <topic> sub-element within the <topics> element.

The Topic name expression syntax and matching shall use the syntax and rules of the POSIX fnmatch() function as specified in

```
<topics>
    <topic>Plane</topic>
    <topic>Hel*</topic>
</topics>
```

Partitions Condition

This section is delimited by the <partitions> XML element. It limits the set Partitions names that may be associated with the (publish, subscribe, relay) action for the rule to apply. Partition names expression syntax and matching shall use the syntax and rules of the POSIX fnmatch() function as specified in IEEE 1003.1-2017. If there is no <partitions> section within a rule, then the default "empty string" partition is assumed.

```
<partitions>
    <partition>A</partition>
    <partition>B*</partition>
</partitions>
```

DomainParticipant Permissions XSD

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="dds" type="PermissionsNode" />
    <xs:complexType name="PermissionsNode">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="permissions" type="Permissions" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Permissions">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="grant" type="Grant" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Grant">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="subject_name" type="xs:string" />
            <xs:element name="validity" type="Validity" />
```

(continues on next page)

(continued from previous page)

```
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:choice minOccurs="1" maxOccurs="1">
                    <xs:element name="allow_rule" minOccurs="0" type="Rule" />
                    <xs:element name="deny_rule" minOccurs="0" type="Rule" />
                </xs:choice>
            </xs:sequence>
            <xs:element name="default" type="DefaultAction" />
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required" />
    </xs:complexType>
    <xs:complexType name="Validity">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="not_before" type="xs:dateTime" />
            <xs:element name="not_after" type="xs:dateTime" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Rule">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="domains" type="DomainIdSet" />
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="publish" type="Criteria" />
            </xs:sequence>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="subscribe" type="Criteria" />
            </xs:sequence>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="relay" type="Criteria" />
            </xs:sequence>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DomainIdSet">
        <xs:choice minOccurs="1" maxOccurs="unbounded">
            <xs:element name="id" type="DomainId" />
            <xs:element name="id_range" type="DomainIdRange" />
        </xs:choice>
    </xs:complexType>
    <xs:simpleType name="DomainId">
        <xs:restriction base="xs:nonNegativeInteger" />
    </xs:simpleType>
    <xs:complexType name="DomainIdRange">
        <xs:choice>
            <xs:sequence>
                <xs:element name="min" type="DomainId" />
                <xs:element name="max" type="DomainId" minOccurs="0" />
            </xs:sequence>
            <xs:element name="max" type="DomainId" />
        </xs:choice>
    </xs:complexType>
    <xs:complexType name="Criteria">
        <xs:all minOccurs="1">
            <xs:element name="topics" minOccurs="1"
                type="TopicExpressionList" />
```

(continues on next page)

(continued from previous page)

```
            <xs:element name="partitions" minOccurs="0"
                type="PartitionExpressionList" />
            <xs:element name="data_tags" minOccurs="0" type="DataTags" />
        </xs:all>
    </xs:complexType>
    <xs:complexType name="TopicExpressionList">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="topic" type="TopicExpression" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="PartitionExpressionList">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="partition" type="PartitionExpression" />
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="TopicExpression">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
    <xs:simpleType name="PartitionExpression">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
    <xs:complexType name="DataTags">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="tag" type="TagNameValuePair" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TagNameValuePair">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="name" type="xs:string" />
            <xs:element name="value" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="DefaultAction">
        <xs:restriction base="xs:string">
            <xs:enumeration value="ALLOW" />
            <xs:enumeration value="DENY" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Back to the *DomainParticipant Permissions Document*.

Signing documents using x509 certificate

Domain Governance Document and *DomainParticipant Permissions Document* have to be signed using an X.509 certificate. Generation of an X.509 certificate is explained in *Generation of X.509 certificates*. Next commands sign the necessary documents for its use by the DDS:Access:Permissions plugin.

```
# Governance document: governance.xml
openssl smime -sign -in governance.xml -text -out governance.smime -signer maincacert.
↪pem -inkey maincakey.pem
```

(continues on next page)

(continued from previous page)

```
# Permissions document: permissions.xml
openssl smime -sign -in permissions.xml -text -out permissions.smime -signer maincacert.
↪pem -inkey maincakey.pem
```

6.22.3 Cryptographic plugin: DDS:Crypto:AES-GCM-GMAC

The cryptographic plugin provides the tools and operations required to support encryption and decryption, digests computation, message authentication codes computation and verification, key generation, and key exchange for DomainParticipants, *DataWriters* and *DataReaders*. Encryption can be applied over three different levels of DDS protocol:

- The whole RTPS messages.
- The RTPS submessages of a specific DDS Entity (DataWriter or DataReader).
- The payload (user data) of a particular DataWriter.

The authentication plugin implemented in Fast DDS is referred to as "DDS:Crypto:AES-GCM-GMAC", in compliance with the DDS Security specification. This plugin is explained in detail below.

The DDS:Crypto:AES-GCM-GMAC plugin provides authentication encryption using Advanced Encryption Standard (AES) in Galois Counter Mode (AES-GCM). It supports 128 bits and 256 bits AES key sizes. It may also provide additional DataReader-specific Message Authentication Codes (MACs) using Galois MAC (AES-GMAC).

The DDS:Crypto:AES-GCM-GMAC authentication plugin, can be activated setting the *DomainParticipantQos properties()* `dds.sec.crypto.plugin` with the value `builtin.AES-GCM-GMAC`. Moreover, this plugin needs the activation of the *Authentication plugin: DDS:Auth:PKI-DH*. The DDS:Crypto:AES-GCM-GMAC plugin is configured using the *Access control plugin: DDS:Access:Permissions*, i.e the cryptography plugin is configured through the properties and configuration files of the access control plugin.

The following is an example of how to set the properties of DomainParticipantQoS for the DDS:Crypto:AES-GCM-GMAC configuration.

C++

```
DomainParticipantQos pqos;

// Activate DDS:Crypto:AES-GCM-GMAC plugin
pqos.properties().properties().emplace_back("dds.sec.crypto.plugin",
        "builtin.AES-GCM-GMAC");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_crypto_plugin_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Crypto:AES-GCM-GMAC plugin -->
                <property>
                    <name>dds.sec.crypto.plugin</name>
                    <value>builtin.AES-GCM-GMAC</value>
                </property>
            </properties>
        </propertiesPolicy>
```

(continues on next page)

(continued from previous page)

```
    </rtps>
</participant>
```

6.22.4 Logging plugin: DDS:Logging:DDS_LogTopic

The logging plugin provides the necessary operations to log the security events triggered by the other security plugins supported by Fast DDS (*Authentication plugin: DDS:Auth:PKI-DH*, *Access control plugin: DDS:Access:Permissions*, and *Cryptographic plugin: DDS:Crypto:AES-GCM-GMAC*). Therefore, the aforementioned security plugins will use the logging plugin to log their events. These events can be reporting of expected behavior, as well as security breaches and errors.

The logging plugin implemented in Fast DDS collects all security event data of a *DomainParticipant* and saves them in a local file. The log messages generated by the logging plugin include an ID that uniquely identifies the DomainParticipant that triggered the event, the DDS Domain identifier to which the DomainParticipant belongs, and a time-stamp.

The logging plugin implemented in Fast DDS is referred to as "DDS:Logging:DDS_LogTopic", in compliance with the DDS Security specification. This plugin is explained in detail below. This plugin can be configured to filter according to up to eight levels of severity of the messages.

The DDS:Logging:DDS_LogTopic authentication plugin, can be activated setting the *DomainParticipantQos properties()* `dds.sec.log.plugin` with the value `builtin.DDS_LogTopic`. The following table outlines the properties used for the DDS:Logging:DDS_LogTopic plugin configuration.

Property name	Property value	
	Value	Definition
logging_level	`EMERGENCY_LEVEL`	System is unusable. Should not continue use.
	`ALERT_LEVEL`	Should be corrected immediately.
	`CRITICAL_LEVEL`	A failure in primary application.
	`ERROR_LEVEL`	General error conditions. Default value.
	`WARNING_LEVEL`	May indicate future error if action not taken.
	`NOTICE_LEVEL`	Unusual, but nor erroneous event or condition.
	`INFORMATIONAL_LEVEL`	Normal operational. Requires no action.
	`DEBUG_LEVEL`	Normal operational.
log_file	Path of the file in which the log messages are to be saved.	

Note: All listed properties have "dds.sec.log.builtin.DDS_LogTopic." prefix. For example: `dds.sec.log.builtin.DDS_LogTopic.logging_level`.

The following is an example of how to set the properties of DomainParticipantQoS for the DDS:Logging:DDS_LogTopic plugin configuration.

C++

```
DomainParticipantQos pqos;

// Activate DDS:Logging:DDS_LogTopic plugin
pqos.properties().properties().emplace_back("dds.sec.log.plugin",
        "builtin.DDS_LogTopic");

// Configure DDS:Logging:DDS_LogTopic plugin
pqos.properties().properties().emplace_back(
    "dds.sec.log.builtin.DDS_LogTopic.logging_level",
    "EMERGENCY_LEVEL");
pqos.properties().properties().emplace_back(
    "dds.sec.log.builtin.DDS_LogTopic.log_file",
    "myLogFile.log");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_logging_plugin_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.log.plugin</name>
                    <value>builtin.DDS_LogTopic</value>
                </property>
                <!-- Configure DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.log.builtin.DDS_LogTopic.logging_level</name>
                    <value>EMERGENCY_LEVEL</value>
                </property>
                <property>
                    <name>dds.sec.log.builtin.DDS_LogTopic.log_file</name>
                    <value>myLogFile.log</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

6.22.5 PKCS#11 support

The *private key* property used for the DDS:Auth:PKI-DH plugin configuration can be specified using a PKCS#11 compliant URI that represents a key stored in a HSM (Hardware Security Module). When a PKCS#11 URI is given, the *private key* is never taken out of the HSM, providing a more secure setup.

Support for PKCS#11 URIs is provided by the libp11 library. This library provides a PKCS#11 engine for OpenSSL that acts as a proxy between OpenSSL and the HSM driver provided by the manufacturer. To make OpenSSL aware of the new engine, the OpenSSL configuration file might need to be updated. For details on how to set up the PKCS#11 engine in different platforms follow the dedicated documentation:

- *Libp11 and SoftHSM libraries* on Windows.
- *Libp11 and SoftHSM libraries* on Linux distributions

Also, the following section contains the list of *PropertyPolicyQos* that can be set within *Fast DDS Security*: *Security Property QoS Settings*.

6.23 Logging

eProsima Fast DDS provides an extensible built-in logging module that exposes the following main functionalities:

- Three different logging levels: `Log::Kind::Info`, `Log::Kind::Warning`, and `Log::Kind::Error` (see *Logging Messages*).
- Message filtering according to different criteria: category, content, or source file (see *Filters*).
- Output to STDOUT, STDERR and/or log files (see *Consumers*).

This section is devoted to explain the use, configuration, and extensibility of Fast DDS' logging module.

6.23.1 Module Structure

The logging module provides the following classes:

- `Log` is the core class of the logging module. This singleton is not only in charge of the logging operations (see *Logging Messages*), but it also provides configuration APIs to set different logging configuration aspects (see *Module Configuration*), as well as logging filtering at various levels (see *Filters*). It contains zero or more `LogConsumer` objects. The singleton's consuming thread feeds the log entries added to the logging queue using the macros defined in *Logging Messages* to the log consumers sequentially (see *Logging Thread*).

  Warning: `Log` API exposes member function `Log::QueueLog()`. However, this function is not intended to be used directly. To add messages to the log queue, use the methods described in *Logging Messages*.

- `LogConsumer` is the base class for all the log consumers (see *Consumers*). It includes the member functions that derived classes should overload to consume log entries.
  - `OStreamConsumer` derives from `LogConsumer`. It defines how to consume log entries for outputting to an `std::ostream` object. It includes a member function that derived classes must overload to define the desired `std::ostream` object.

    1. `StdoutConsumer` derives from `OStreamConsumer`. It defines STDOUT as the output `std::ostream` object (see *StdoutConsumer*).

    2. `StdoutErrConsumer` derives from `OStreamConsumer`. It defines a `Log::Kind` threshold so that if the `Log::Kind` is equal to or more severe than the selected threshold, the output defined will be STDERR. Otherwise, it defines STDOUT as the output (see *StdoutErrConsumer*).

    3. `FileConsumer` derives from `OStreamConsumer`. It defines an user specified file as the output `std::ostream` object (see *FileConsumer*).

Fig. 12: Logging module class diagram

The module can be further extended by creating new consumer classes deriving from `LogConsumer` and/or `OStreamConsumer`. To enable a custom consumer just follow the instructions on *Register Consumers*.

6.23.2 Log Entry Specification

Log entries created by *StdoutConsumer*, *StdoutErrConsumer* and *FileConsumer* (*eProsima Fast DDS* built-in *Consumers*) adhere to the following structure:

```
<Timestamp> [<Category> <Verbosity Level>] <Message> (<File Name>:<Line Number>) ->␣
↪Function <Function Name>
```

An example of such log entry is given by:

```
2020-05-27 11:45:47.447 [DOCUMENTATION_CATEGORY Error] This is an error message (example.
↪cpp:50) -> Function main
```

Note: *File Name* and *Line Number*, as well as *Function Name* are only present when enabled. See *Module Configuration* for details.

6.23.3 Logging Thread

Calls to the macros presented in *Logging Messages* merely add the log entry to a ready-to-consume queue. Upon creation, the logging module spawns a thread that awakes every time an entry is added to the queue. When awaken, this thread feeds all the entries in the queue to all the registered *Consumers*. Once the work is done, the thread falls back into idle state. This strategy prevents the module from blocking the application thread when a logging operation is performed. However, sometimes applications may want to wait until the logging routine is done to continue their operation. The logging module provides this capability via the member function *Log::Flush()*. Furthermore, it is possible to completely eliminate the thread and its resources using member function *Log::KillThread()*.

```
// Block current thread until the log queue is empty.
Log::Flush();

// Stop the loggin thread and free its resources.
Log::KillThread();
```

Warning: A call to any of the macros present in *Logging Messages* will spawn the logging thread even if it has been previously killed with *Log::KillThread()*.

6.23.4 Logging Messages

The logging of messages is handled by three dedicated macros, one for each available verbosity level (see *Verbosity Level*):

- *EPROSIMA_LOG_INFO*: Logs messages with *Log::Kind::Info* verbosity.
- *EPROSIMA_LOG_WARNING*: Logs messages with *Log::Kind::Warning* verbosity.
- *EPROSIMA_LOG_ERROR*: Logs messages with *Log::Kind::Error* verbosity.

Said macros take exactly two arguments, a category and a message, and produce a log entry showing the message itself plus some meta information depending on the module's configuration (see *Log Entry Specification* and *Log Entry*).

```
EPROSIMA_LOG_INFO(DOCUMENTATION_CATEGORY, "This is an info message");
EPROSIMA_LOG_WARNING(DOCUMENTATION_CATEGORY, "This is an warning message");
EPROSIMA_LOG_ERROR(DOCUMENTATION_CATEGORY, "This is an error message");
```

There exist some old log macros used in previous versions: `logInfo`, `logWarning` and `logError`. These macros are still available as long as user does not manually disable them by `ENABLE_OLD_LOG_MACROS` CMake option or in-site macro `ENABLE_OLD_LOG_MACROS_` before including *Log* module. See section *Old Log macros disable* for more information.

Warning: Note that each message level is deactivated when CMake options `LOG_NO_INFO`, `LOG_NO_WARNING` or `LOG_NO_ERROR` are set to `ON` respectively. For more information about how to enable and disable each individual logging macro, please refer to *Disable Logging Module*.

6.23.5 Module Configuration

The logging module offers a variety of configuration options. The different components of a log entry (see *Log Entry Specification*) can be configured as explained in *Log Entry*. Furthermore, the logging module allows for registering several log consumer, allowing applications to direct the logging output to different destinations (see *Register Consumers*). In addition, some of the logging features can be configured using *eProsima Fast DDS* XML configuration files (see *XML Configuration*).

- *Log Entry*
- *Register Consumers*
- *Reset Configuration*
- *XML Configuration*

Log Entry

All the different components of a log entry are summarized in the following table (please refer to each component's section for further explanation):

Component	Optional	Default
Timestamp	NO	ENABLED
Category	NO	ENABLED
Verbosity Level	NO	ENABLED
Message	NO	ENABLED
File Context	YES	DISABLED
Function Name	YES	ENABLED

Timestamp

The log timestamp follows the ISO 8601 standard for local timestamps, i.e. *YYYY-MM-DD hh:mm:ss.sss*. This component cannot be further configured or disabled.

Category

Log entries have a category assigned when producing the log via the macros presented in *Logging Messages*. The category component can be used to filter log entries so that only those categories specified in the filter are consumed (see *Filters*). This component cannot be further configured or disabled.

Verbosity Level

eProsima Fast DDS logging module provides three verbosity levels defined by the *Log::Kind* enumeration, those are:

- *Log::Kind::Error*: Used to log error messages.
- *Log::Kind::Warning*: Used to log error and warning messages.
- *Log::Kind::Info*: Used to log error, warning, and info messages.

The logging module's verbosity level defaults to *Log::Kind::Error*, which means that only messages logged with *EPROSIMA_LOG_ERROR* would be consumed. The verbosity level can be set and retrieved using member functions *Log::SetVerbosity()* and *Log::GetVerbosity()* respectively.

```
// Set log verbosity level to Log::Kind::Info
Log::SetVerbosity(Log::Kind::Info);

// Get log verbosity level
Log::Kind verbosity_level = Log::GetVerbosity();
```

Warning: Setting any of the CMake options `LOG_NO_INFO`, `LOG_NO_WARNING` or `LOG_NO_ERROR` to `ON` will completely disable the corresponding verbosity level. `LOG_NO_INFO` is set to `ON` for Single-Config generators as default value if not in `Debug` mode.

Message

This component constitutes the body of the log entry. It is specified when producing the log via the macros presented in *Logging Messages*. The message component can be used to filter log entries so that only those entries whose message pattern-matches the filter are consumed (see *Filters*). This component cannot be further configured or disabled.

File Context

This component specifies the origin of the log entry in terms of file name and line number (see *Logging Messages* for a log entry example featuring this component). This is useful when tracing code flow for debugging purposes. The file context component can be enabled/disabled using the member function *Log::ReportFilenames()*.

```
// Enable file name and line number reporting
Log::ReportFilenames(true);

// Disable file name and line number reporting
Log::ReportFilenames(false);
```

Function Name

This component specifies the origin of the log entry in terms of the function name (see *Logging Messages* for a log entry example featuring this component). This is useful when tracing code flow for debugging purposes. The function name component can be enabled/disabled using the member function *Log::ReportFunctions()*.

```
// Enable function name reporting
Log::ReportFunctions(true);

// Disable function name reporting
Log::ReportFunctions(false);
```

Register Consumers

eProsima Fast DDS logging module supports zero or more *consumers* logging the entries registered in the logging queue with the methods described in *Logging Messages*. To register a consumer, the *Log* class exposes member function *Log::RegisterConsumer()*

```
// Create a FileConsumer consumer that logs entries in "archive.log"
std::unique_ptr<FileConsumer> file_consumer(new FileConsumer("archive.log"));
// Register the consumer. Log entries will be logged to STDOUT and "archive.log"
Log::RegisterConsumer(std::move(file_consumer));
```

The consumers list can be emptied with member function *Log::ClearConsumers()*.

```
// Clear all the consumers. Log entries are discarded upon consumption.
Log::ClearConsumers();
```

Note: Registering and configuring consumers can also be done using *Fast DDS* XML configuration files. Please refer to *XML Configuration* for details.

Warning: *Log::ClearConsumers()* empties the consumers lists. All log entries are discarded until a new consumer is register via *Log::RegisterConsumer()*, or until *Log::Reset()* is called.

Reset Configuration

The logging module's configuration can be reset to default settings with member function *Log::Reset()*.

Warning: Resetting the module's configuration entails:

- Setting *Verbosity Level* to *Log::Kind::Error*.
- Disabling *File Context* component.
- Enabling *Function Name* component.
- Clear all *Filters*.
- Clear all consumers and reset the default consumer according to CMake option LOG_CONSUMER_DEFAULT.

XML Configuration

eProsima Fast DDS allows for registering and configuring log consumers using XML configuration files. Please refer to *Log profiles* for details.

6.23.6 Filters

eProsima Fast DDS logging module allows for log entry filtering when consuming the logs, so that an application execution output can be limited to specific areas of interest. Beside the *Verbosity Level*, *Fast DDS* provides three different filtering possibilities.

- *Category Filtering*
- *File Name Filtering*
- *Content Filtering*
- *Reset Logging Filters*

It is worth mentioning that filters are applied in the specific order presented above, meaning that file name filtering is only applied to the entries that pattern-match the category filter, and content filtering is only applied to the entries that pattern-match both category and file name filters.

Category Filtering

Log entries can be filtered upon consumption according to their *Category* component using regular expressions. Each time an entry is ready to be consumed, the category filter is applied using `std::regex_search()`. To set a category filter, member function *Log::SetCategoryFilter()* is used:

```
// Set filter using regular expression
Log::SetCategoryFilter(std::regex("(CATEGORY_1)|(CATEGORY_2)"));

// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY_1, "First log entry");
// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY_2, "Second log entry");
```

(continues on next page)

(continued from previous page)

```
// Would NOT be consumed
EPROSIMA_LOG_ERROR(CATEGORY_3, "Third log entry");
```

The previous example would produce the following output:

```
2020-05-27 15:07:05.771 [CATEGORY_FILTER_1 Error] First log entry -> Function main
2020-05-27 15:07:05.771 [CATEGORY_FILTER_2 Error] Second log entry -> Function main
```

File Name Filtering

Log entries can be filtered upon consumption according to their *File Context* component using regular expressions. Each time an entry is ready to be consumed, the file name filter is applied using `std::regex_search()`. To set a file name filter, member function *Log::SetFilenameFilter()* is used:

```
// Filename: example.cpp

// Enable file name and line number reporting
Log::ReportFilenames(true);

// Set filter using regular expression so filename must match "example"
Log::SetFilenameFilter(std::regex("example"));
// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY, "First log entry");

// Set filter using regular expression so filename must match "other"
Log::SetFilenameFilter(std::regex("other"));
// Would NOT be consumed
EPROSIMA_LOG_ERROR(CATEGORY, "Second log entry");
```

The previous example would produce the following output:

```
2020-05-27 15:07:05.771 [CATEGORY Error] First log entry (example.cpp:50) -> Function␣
↪main
```

Note: File name filters are applied even when the *File Context* entry component is disabled.

Content Filtering

Log entries can be filtered upon consumption according to their *Message* component using regular expressions. Each time an entry is ready to be consumed, the content filter is applied using `std::regex_search()`. To set a content filter, member function *Log::SetErrorStringFilter()* is used:

```
// Set filter using regular expression so message component must match "First"
Log::SetErrorStringFilter(std::regex("First"));
// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY, "First log entry");
// Would NOT be consumed
EPROSIMA_LOG_ERROR(CATEGORY, "Second log entry");
```

The previous example would produce the following output:

```
2020-05-27 15:07:05.771 [CATEGORY Error] First log entry -> Function main
```

Reset Logging Filters

The logging module's filters can be reset with member function *Log::Reset()*.

Warning: Resetting the module's filters entails:

- Setting *Verbosity Level* to *Log::Kind::Error*.
- Disabling *File Context* component.
- Enabling *Function Name* component.
- Clear all *Filters*.
- Clear all consumers and reset the default consumer according to CMake option `LOG_CONSUMER_DEFAULT`.

6.23.7 Consumers

Consumers are classes that take a *Log::Entry* and produce a log output accordingly. *eProsima Fast DDS* provides three different log consumers that output log entries to different streams:

- *StdoutConsumer*: Outputs log entries to STDOUT
- *StdoutErrConsumer*: Outputs log entries to STDOUT or STDERR depending on the given threshold.
- *FileConsumer*: Outputs log entries to a user specified file.

StdoutConsumer

StdoutConsumer outputs log entries to STDOUT stream following the convection specified in *Log Entry Specification*. It is the default and only log consumer of the logging module if the CMake option `LOG_CONSUMER_DEFAULT` is set to `AUTO`, `STDOUT`, or not set at all. It can be registered and unregistered using the methods explained in *Register Consumers* and *Reset Configuration*.

```
// Create a StdoutConsumer consumer that logs entries to stdout stream.
std::unique_ptr<StdoutConsumer> stdout_consumer(new StdoutConsumer());

// Register the consumer.
Log::RegisterConsumer(std::move(stdout_consumer));
```

StdoutErrConsumer

StdoutErrConsumer uses a *Log::Kind* threshold to filter the output of the log entries. Those log entries whose *Log::Kind* is equal to or more severe than the given threshold output to STDERR. Other log entries output to STDOUT. By default, the threshold is set to *Log::Kind::Warning*. *StdoutErrConsumer::stderr_threshold()* allows the user to modify the default threshold.

Additionally, if CMake option `LOG_CONSUMER_DEFAULT` is set to `STDOUTERR`, the logging module will use this consumer as the default log consumer.

```
// Create a StdoutErrConsumer consumer that logs entries to stderr only when the␣
↪Log::Kind is equal to ERROR
std::unique_ptr<StdoutErrConsumer> stdouterr_consumer(new StdoutErrConsumer());
stdouterr_consumer->stderr_threshold(Log::Kind::Error);

// Register the consumer
Log::RegisterConsumer(std::move(stdouterr_consumer));
```

FileConsumer

FileConsumer provides the logging module with log-to-file logging capabilities. Applications willing to hold a persistent execution log record can specify a logging file using this consumer. Furthermore, the application can choose whether the file stream should be in "write" or "append" mode, according to the behaviour defined by `std::fstream::open()`.

```
// Create a FileConsumer consumer that logs entries in "archive_1.log", opening the file␣
↪in "write" mode.
std::unique_ptr<FileConsumer> write_file_consumer(new FileConsumer("archive_1.log",␣
↪false));

// Create a FileConsumer consumer that logs entries in "archive_2.log", opening the file␣
↪in "append" mode.
std::unique_ptr<FileConsumer> append_file_consumer(new FileConsumer("archive_2.log",␣
↪true));

// Register the consumers.
Log::RegisterConsumer(std::move(write_file_consumer));
Log::RegisterConsumer(std::move(append_file_consumer));
```

6.23.8 Disable Logging Module

Setting the *Verbosity Level*, translates into entries not being added to the log queue if the entry's level has lower importance than the set one. This check is performed when calling the macros defined in *Logging Messages*. However, it is possible to fully disable each macro (and therefore each verbosity level individually) at build time.

- *EPROSIMA_LOG_INFO* is fully disabled by either:
  - Setting CMake option `LOG_NO_INFO` to `ON` (default for Single-Config generators if `CMAKE_BUILD_TYPE` is other than `Debug`).
  - Defining macro `HAVE_LOG_NO_INFO` to `1`.
- *EPROSIMA_LOG_WARNING* is fully disabled by either:
  - Setting CMake option `LOG_NO_WARNING` to `ON`.
  - Defining macro `HAVE_LOG_NO_WARNING` to `1`.
- *EPROSIMA_LOG_ERROR* is fully disabled by either:
  - Setting CMake option `LOG_NO_ERROR` to `ON`.
  - Defining macro `HAVE_LOG_NO_ERROR` to `1`.

Applying either of the previously described methods will set the macro to be empty at configuration time, thus allowing the compiler to optimize the call out. This is done so that all the debugging messages present on the library are optimized out at build time if not building for debugging purposes, thus preventing them to impact performance.

`INTERNAL_DEBUG` CMake option activates log macros compilation, so the arguments of the macros are compiled. However:

- It does not activate the log Warning and Error messages, i.e. the messages are not written in the log queue.
- *EPROSIMA_LOG_INFO* has a special behaviour to simplify working with Multi-Config capability IDEs. If CMake option `LOG_NO_INFO` is `OFF`, or the C++ definition `HAVE_LOG_NO_INFO` is `0`, then logging is enabled only for `Debug` configuration. In this scenario, setting `FASTDDS_ENFORCE_LOG_INFO` to `ON` will enable *EPROSIMA_LOG_INFO* even on non `Debug` configurations. This is specially useful when using the *Fast DDS*' logging module in an external application which links with *Fast DDS* compiled in `Release`. In that case, applications wanting to use all three levels of logging can simply add the following code prior to including any Fast DDS header:

```
#define HAVE_LOG_NO_INFO 0
#define FASTDDS_ENFORCE_LOG_INFO 1
```

Warning: `INTERNAL_DEBUG` can be automatically set to `ON` if CMake option `EPROSIMA_BUILD` is set to `ON`.

6.23.9 Old Log macros disable

Before version 2.8.2, Fast DDS project used log macros: `logInfo`, `logWarning` and `logError`, which may collide with other libraries. These log macros have been replaced by new ones with a more specific format: (e.g. `EPROSIMA_LOG_INFO`). In order to disable old macros compilation, use CMake option `ENABLE_OLD_LOG_MACROS = ON` or define `ENABLE_OLD_LOG_MACROS_ 0` before including the log module `#include <fastdds/dds/log/Log.hpp>`.

Warning: These macros will be deprecated in future versions of Fast DDS. The use of the new format ones is encouraged.

6.24 Statistics Module

The *Fast DDS Statistics module* is an extension of Fast DDS that enables the recollection of data concerning the DDS communication. The collected data is published using DDS over dedicated topics using builtin DataWriters within the *Statistics module*. Consequently, by default, Fast DDS does not compile this module because it may entail affecting the application's performance. Nonetheless, the Statistics module can be activated using the `-DFASTDDS_STATISTICS=ON` at CMake configuration step. For more information about *Fast DDS* compilation, see *Linux installation from sources* and *Windows installation from sources*.

Besides enabling the *Statistics Module* compilation, the user must enable those DataWriters that are publishing data on the topics of interest for the user's application. Therefore, the standard *DDS Layer* has been extended. The following section explains this DDS extended API.

Note: Please refer to *Statistics QoS Troubleshooting* for any problems related to the statistics module.

6.24.1 Statistics Module DDS Layer

This section explains the extended DDS API provided for the *Statistics Module*. First, the Statistics Topic List is presented together with the corresponding collected data. Next, the methods to enable/disable the corresponding DataWriters are explained. Then, the recommended QoS for enabling the DataWriters and creating the user's DataReaders that subscribe to the Statistics topics are described. Finally, a guide on how to overcome common problems when using the module are presented.

Statistics Topic names

Data collected by the *Fast DDS Statistics module* is published in one of the topics listed below. In order to simplify its use, the API provides aliases for the different statistics topics (see *Topic names*). The following table shows the correlation between the topic name and the corresponding alias.

Topic name	Alias
_fastdds_statistics_history2history_latency	*HISTORY_LATENCY_TOPIC*
_fastdds_statistics_network_latency	*NETWORK_LATENCY_TOPIC*
_fastdds_statistics_publication_throughput	*PUBLICATION_THROUGHPUT_TOPIC*
_fastdds_statistics_subscription_throughput	*SUBSCRIPTION_THROUGHPUT_TOPIC*
_fastdds_statistics_rtps_sent	*RTPS_SENT_TOPIC*
_fastdds_statistics_rtps_lost	*RTPS_LOST_TOPIC*
_fastdds_statistics_heartbeat_count	*HEARTBEAT_COUNT_TOPIC*
_fastdds_statistics_acknack_count	*ACKNACK_COUNT_TOPIC*
_fastdds_statistics_nackfrag_count	*NACKFRAG_COUNT_TOPIC*
_fastdds_statistics_gap_count	*GAP_COUNT_TOPIC*
_fastdds_statistics_data_count	*DATA_COUNT_TOPIC*
_fastdds_statistics_resent_datas	*RESENT_DATAS_TOPIC*
_fastdds_statistics_sample_datas	*SAMPLE_DATAS_TOPIC*
_fastdds_statistics_pdp_packets	*PDP_PACKETS_TOPIC*
_fastdds_statistics_edp_packets	*EDP_PACKETS_TOPIC*
_fastdds_statistics_discovered_entity	*DISCOVERY_TOPIC*
_fastdds_statistics_physical_data	*PHYSICAL_DATA_TOPIC*

HISTORY_LATENCY_TOPIC

The `_fastdds_statistics_history2history_latency` statistics topic collects data related with the latency between any two matched endpoints. This measurement provides information about the DDS overall latency independent of the user's application overhead. Specifically, the measured latency corresponds to the time spent between the instant when the sample is written to the DataWriter's history and the time when the sample is added to the DataReader's history and the notification is issued to the corresponding user's callback.

NETWORK_LATENCY_TOPIC

The `_fastdds_statistics_network_latency` statistics topic collects data related with the network latency (expressed in *ns*) between any two communicating locators. This measurement provides information about the transport layer latency. The measured latency corresponds to the time spent between the message being written in the `RTPSMessageGroup` until the message being received in the `MessageReceiver`.

Important: In the case of *TCP Transport*, the reported latency also includes the time spent on the datagram's CRC related operations. Mind that is possible to disable CRC operations when defining the *TCPTransportDescriptor*.

PUBLICATION_THROUGHPUT_TOPIC

The `_fastdds_statistics_publication_throughput` statistics topic collects the amount of data (expressed in *B/s*) that is being sent by each DataWriter. This measurement provides information about the publication's throughput.

SUBSCRIPTION_THROUGHPUT_TOPIC

The `_fastdds_statistics_subscription_throughput` statistics topic collects the amount of data (expressed in *B/s*) that is being received by each DataReader. This measurement provides information about the subscription's throughput.

RTPS_SENT_TOPIC

The `_fastdds_statistics_rtps_sent` statistics topic collects the number of RTPS packets and bytes that are being sent from each DDS entity to each locator.

RTPS_LOST_TOPIC

The `_fastdds_statistics_rtps_lost` statistics topic collects the number of RTPS packets and bytes that are being lost in the transport layer (dropped somewhere in between) in the communication between each DDS entity and locator.

HEARTBEAT_COUNT_TOPIC

The `_fastdds_statistics_heartbeat_count` statistics topic collects the number of heartbeat messages sent by each user's DataWriter. This topic does not apply to builtin (related to *Discovery*) and statistics DataWriters. Heartbeat messages are only sent if the *ReliabilityQosPolicy* is set to *RELIABLE_RELIABILITY_QOS*. These messages report the DataWriter's status.

ACKNACK_COUNT_TOPIC

The `_fastdds_statistics_acknack_count` statistics topic collects the number of acknack messages sent by each user's DataReader. This topic does not apply to builtin DataReaders (related to *Discovery*). Acknack messages are only sent if the *ReliabilityQosPolicy* is set to *RELIABLE_RELIABILITY_QOS*. These messages report the DataReader's status.

NACKFRAG_COUNT_TOPIC

The `_fastdds_statistics_nackfrag_count` statistics topic collects the number of nackfrag messages sent by each user's DataReader. This topic does not apply to builtin DataReaders (related to *Discovery*). Nackfrag messages are only sent if the *ReliabilityQosPolicy* is set to *RELIABLE_RELIABILITY_QOS*. These messages report the data fragments that have not been received yet by the DataReader.

GAP_COUNT_TOPIC

The `_fastdds_statistics_gap_count` statistics topic collects the number of gap messages sent by each user's DataWriter. This topic does not apply to builtin (related to *Discovery*) and statistics DataWriters. Gap messages are only sent if the *ReliabilityQosPolicy* is set to *RELIABLE_RELIABILITY_QOS*. These messages report that some specific samples are not relevant to a specific DataReader.

DATA_COUNT_TOPIC

The `_fastdds_statistics_data_count` statistics topic collects the total number of user's data messages and data fragments (in case that the message size is large enough to require RTPS fragmentation) that have been sent by each user's DataWriter. This topic does not apply to builtin (related to *Discovery*) and statistics DataWriters.

RESENT_DATAS_TOPIC

The `_fastdds_statistics_resent_data` statistics topic collects the total number of user's data messages and data fragments (in case that the message size is large enough to require RTPS fragmentation) that have been necessary to resend by each user's DataWriter. This topic does not apply to builtin (related to *Discovery*) and statistics DataWriters.

SAMPLE_DATAS_TOPIC

The `_fastdds_statistics_sample_datas` statistics topic collects the number of user's data messages (or data fragments in case that the message size is large enough to require RTPS fragmentation) that have been sent by the user's DataWriter to completely deliver a single sample. This topic does not apply to builtin (related to *Discovery*) and statistics DataWriters.

PDP_PACKETS_TOPIC

The `_fastdds_statistics_pdp_packets` statistics topic collects the number of PDP discovery traffic RTPS packets transmitted by each DDS `DomainParticipant`. PDP packets are the data messages exchanged during the PDP discovery phase (see *Discovery phases* for more information).

EDP_PACKETS_TOPIC

The `_fastdds_statistics_edp_packets` statistics topic collects the number of EDP discovery traffic RTPS packets transmitted by each DDS `DomainParticipant`. EDP packets are the data messages exchanged during the EDP discovery phase (see *Discovery phases* for more information).

DISCOVERY_TOPIC

The `_fastdds_statistics_discovered_entity` statistics topic reports the time when each local `DomainParticipant` discovers any remote DDS entity (with the exception of those DDS entities related with the *Fast DDS Statistics module*). This topic also carries the `PHYSICAL_DATA_TOPIC` information for the case of discovered `DomainParticipant`; if the discovered entity is either a `DataReader` or `DataWriter`, then the physical information is empty (see *Physical Data in Discovery Information* for more information about how to configure the physical data conveyed on the discovery messages).

PHYSICAL_DATA_TOPIC

The `_fastdds_statistics_physical_data` statistics topic reports the host, user and process where the *Fast DDS Statistics module* is running.

Statistics Domain Participant

In order to start collecting data in one of the statistics topics (*Statistics Topic names*), the corresponding statistics DataWriter should be enabled. In fact, *Fast DDS Statistics module* can be enabled and disabled at runtime. For this purpose, *Fast DDS Statistics module* exposes an extended DDS `DomainParticipant` API:

- *Enable statistics DataWriters*
- *Disable statistics DataWriters*
- *Obtain pointer to the extended* `DomainParticipant` *class*
- *Example*
- *Automatically enabling statistics DataWriters*

Enable statistics DataWriters

Statistics DataWriters can be enabled in different ways. It can be done automatically (see *Automatically enabling statistics DataWriters*). Alternatively, Statistics DataWriters can be enabled at run time using one of two methods: *enable_statistics_datawriter()* or *enable_statistics_datawriter_with_profile()*.

enable_statistics_datawriter() method requires as parameters:

- Name of the statistics topic to be enabled (see *Statistics Topic names* for the statistics topic list).
- DataWriter QoS profile (see *Statistics DataWriter recommended QoS* for the recommended profile).

It is possible to define specific desired QoS through DataWriter profile on the *FASTRTPS_DEFAULT_PROFILES_FILE* (see *XML profiles*). *enable_statistics_datawriter_with_profile()* method enables a DataWriter by searching a specific DataWriter XML profile. On those profiles, specific QoS can be set.

enable_statistics_datawriter_with_profile() method requires as parameters:

- Name of the XML profile to use to fill the QoS structure of the DataWriter.
- Name of the statistics topic name to be enabled. (see *Statistics Topic names* for the statistics topic list).

Disable statistics DataWriters

Statistics DataWriters are disabled using the method *disable_statistics_datawriter()*. This method requires as parameter:

- Name of the statistics topic to be disabled (see *Statistics Topic names* for the statistics topic list).

Obtain pointer to the extended `DomainParticipant` class

The *DomainParticipant* is created using the *create_participant()* provided by the *DomainParticipantFactory*. This method returns a pointer to the DDS standard DomainParticipant created. In order to obtain the pointer to the child *DomainParticipant* which extends the DDS API, the `static` method *narrow()* is provided.

Example

The following example shows how to use the Statistics module extended DDS API:

```
// Create a DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Obtain pointer to child class
eprosima::fastdds::statistics::dds::DomainParticipant* statistics_participant =
        eprosima::fastdds::statistics::dds::DomainParticipant::narrow(participant);
```

(continues on next page)

(continued from previous page)

```
// Enable statistics DataWriter
if (statistics_participant->enable_statistics_
↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC,
        eprosima::fastdds::statistics::dds::STATISTICS_DATAWRITER_QOS) != ReturnCode_
↪t::RETCODE_OK)
{
    // Error
    return;
}

// Use the DomainParticipant to communicate
// (...)

// Disable statistics DataWriter
if (statistics_participant->disable_statistics_
↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC) !=
        ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Delete DomainParticipant
if (DomainParticipantFactory::get_instance()->delete_participant(participant) !=␣
↪ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Automatically enabling statistics DataWriters

The statistics DataWriters can be directly enabled using the *DomainParticipantQos* `properties()` `fastdds.statistics`. The value of this property is a semicolon separated list containing the *statistics topic name aliases* of those DataWriters that the user wants to enable. The property can be set either programmatically or loading an XML file. If the property is set in both ways, the priority would depend on the API and the QoS profile provided:

- XML settings have priority if `create_participant_with_profile()` is called with a valid participant profile.
- XML settings also have priority if `create_participant()` is called using `PARTICIPANT_QOS_DEFAULT` and a participant profile exists in the XML file with the `is_default_profile` option set to `true` (*DomainParticipant XML attributes*).
- The property set programmatically is used only when `create_participant()` is called with the specific QoS.

Another way of enabling statistics DataWriters, compatible with the previous one, is setting the *FASTDDS_STATISTICS* environment variable. The statistics DataWriters that will be enabled when the `DomainParticipant` is enabled would be the union between those specified in the `properties()` `fastdds.statistics` and those included with the environment variable.

The following examples show how to use all the previous methods:

C++

```
DomainParticipantQos pqos;

// Activate Fast DDS Statistics module
pqos.properties().properties().emplace_back("fastdds.statistics",
        "HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC
↪");
```

XML

```
<participant profile_name="statistics_domainparticipant_conf_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate Fast DDS Statistics Module -->
                <property>
                    <name>fastdds.statistics</name>
                    <value>HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
↪PHYSICAL_DATA_TOPIC</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Environment Variable Linux

```
export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
↪PHYSICAL_DATA_TOPIC"
```

Environment Variable Windows

```
set FASTDDS_STATISTICS=HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
↪PHYSICAL_DATA_TOPIC
```

Note: These are all the statistics topics:

```
HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;SUBSCRIPTION_
↪THROUGHPUT_TOPIC;RTPS_SENT_TOPIC;RTPS_LOST_TOPIC;HEARTBEAT_COUNT_TOPIC;ACKNACK_COUNT_
↪TOPIC;NACKFRAG_COUNT_TOPIC;GAP_COUNT_TOPIC;DATA_COUNT_TOPIC;RESENT_DATAS_TOPIC;SAMPLE_
↪DATAS_TOPIC;PDP_PACKETS_TOPIC;EDP_PACKETS_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC
```

Note: Be aware that automatically enabling the statistics DataWriters using all these methods implies using the recommended QoS profile *STATISTICS_DATAWRITER_QOS*. For more information, please refer to *Statistics DataWriter recommended QoS*. However, if an XML profile is defined, the QoS applied are those defined in the profile, and for those QoS that are not specified in that profile, the default library QoS are applied (see *DataWriterQos* for the standard eProsima's DataWriter QoS), and not the recommended QoS for the Statistics DataWriters.

For the creation of an automatically enabled Datawriter, the priority for setting its QoS is the following:

- First, if a specific profile exists for the statistics topic, that one is applied.
- If that is not the case but a generic profile for statistics DataWriters exists, that one is applied.
- If no profile is defined in XML file, the recommended statistics QoS are applied.

Note: The generic DataWriter profile defined in the *FASTRTPS_DEFAULT_PROFILES_FILE* XML needs to be named as *GENERIC_STATISTICS_PROFILE*.

The specific DataWriter profile defined in the *FASTRTPS_DEFAULT_PROFILES_FILE* XML needs to be named using the same statistic topic alias or name (see *Statistics Topic names* for the alias corresponding to each statistic topic) that has been used in the *DomainParticipantQos* `properties()` `fastdds.statistics` (see *Statistics Module Settings*) or the *FASTDDS_STATISTICS* environment variable, where the enabling of the corresponding statistics topic has been set.

Statistics recommended QoS

Although the statistics DataWriters can be enabled using any valid QoS profile, the recommended profile is presented below. Also, the DataReaders created by the user to receive the data being published by the statistics DataWriters can use any compatible QoS profile. However, a recommended DataReader QoS profile is also provided.

Statistics DataWriter recommended QoS

The following table shows the recommended *DataWriterQos* profile for enabling the statistics DataWriters. This profile enables the `pull mode` *operating mode* on the statistics DataWriters. This entails that the DataWriters will only send information upon the reception of acknack submessages sent by the monitoring DataReader. This QoS profile is always used when the statistics DataWriters are *auto-enabled*. The recommended profile can be accessed through the constant *STATISTICS_DATAWRITER_QOS*.

Qos Policy	Value
ReliabilityQosPolicyKind	*RELIABLE_RELIABILITY_QOS*
DurabilityQosPolicyKind	*TRANSIENT_LOCAL_DURABILITY_QOS*
PublishModeQosPolicyKind	*ASYNCHRONOUS_PUBLISH_MODE*
flow_controller_name	*FASTDDS_STATISTICS_FLOW_CONTROLLER_DEFAULT*
HistoryQosPolicyKind	*KEEP_LAST_HISTORY_QOS*
history_depth	10
PropertyPolicyQos name = value	`"fastdds.push_mode" = "false"`

Statistics DataReader recommended QoS

The following table shows the recommended *DataReaderQos* profile for creating the monitoring DataReaders. The recommended profile can be accessed through constant *STATISTICS_DATAREADER_QOS*.

Qos Policy	Value
ReliabilityQosPolicyKind	*RELIABLE_RELIABILITY_QOS*
DurabilityQosPolicyKind	*TRANSIENT_LOCAL_DURABILITY_QOS*
HistoryQosPolicyKind	*KEEP_LAST_HISTORY_QOS*
history_depth	100
MemoryManagementPolicy	*PREALLOCATED_WITH_REALLOC_MEMORY_MODE*

Troubleshooting

This section aims to give quick solutions to overcome the most common problems arising from the use of the statistics module.

Monitoring application is not receiving any statistic data

Sometimes, especially in the case of monitoring large applications with many DataWriters and DataReaders, it may happen that the application monitoring Fast DDS statistics does not receive any data. This is generally caused by the default configuration of the statistics DataWriters, which includes the `push_mode` set to `false` (i.e. `pull_mode`), the History Kind set to `KEEP_LAST`, and the History Depth set to `10`. With this configuration, the following may happen:

1. Fast DDS adds new samples to one of the statistics DataWriters.
2. The DataWriter notifies the DataReader of the availability of said samples.
3. The DataReader sends a request to the DataWriter to "pull" those samples.
4. Before the request arrives to the DataWriter, some new statistics samples are added to that same DataWriter, which causes the previous samples to be overwritten.
5. Once the DataReader request arrives to the DataWriter, since the requested samples have been overwritten, they are not available any more, so the DataWriter send a notification to the DataReader informing of the presence of the newer samples instead.
6. The loop starts again.

The easiest fix to overcome this situation is to simply increase the History Depth of the DataWriter to create Some buffer to answer to requests:

Generic profile

```
<?xml version="1.0" encoding="utf-8"?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="statistics_domainparticipant_conf_xml_general_profile
↪">
            <rtps>
                <propertiesPolicy>
                    <properties>
                        <!-- Activate various Fast DDS Statistics Module DataWriters -->
                        <property>
                            <name>fastdds.statistics</name>
                            <value>HISTORY_LATENCY_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_
↪TOPIC</value>
                        </property>
                    </properties>
                </propertiesPolicy>
            </rtps>
        </participant>

        <!-- Generic profile for all the statistics DataWriter -->
        <data_writer profile_name="GENERIC_STATISTICS_PROFILE">
            <!-- Configure History QoS as KEEP_LAST 20 -->
            <!-- History depth depends on the user application constraints (publication␣
↪rate for instance) -->
```

(continues on next page)

(continued from previous page)

```
            <topic>
                <historyQos>
                    <kind>KEEP_LAST</kind>
                    <depth>20</depth>
                </historyQos>
            </topic>
            <!-- Enable pull mode -->
            <propertiesPolicy>
                <properties>
                    <property>
                        <name>fastdds.push_mode</name>
                        <value>false</value>
                    </property>
                </properties>
            </propertiesPolicy>
            <!-- Set durability, reliability, and publication mode -->
            <qos>
                <durability>
                    <kind>TRANSIENT_LOCAL</kind>
                </durability>

                <reliability>
                    <kind>RELIABLE</kind>
                </reliability>

                <publishMode>
                    <kind>ASYNCHRONOUS</kind>
                </publishMode>
            </qos>
        </data_writer>
    </profiles>
</dds>
```

Specific profile

```
<?xml version="1.0" encoding="utf-8"?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="statistics_domainparticipant_conf_xml_specific_profile
↪">
            <rtps>
                <propertiesPolicy>
                    <properties>
                        <!-- Activate various Fast DDS Statistics Module DataWriters -->
                        <property>
                            <name>fastdds.statistics</name>
                            <value>HISTORY_LATENCY_TOPIC</value>
                        </property>
                    </properties>
                </propertiesPolicy>
            </rtps>
        </participant>
```

(continues on next page)

(continued from previous page)

```
        <!-- Generic profile for a specific statistics DataWriters -->
        <data_writer profile_name="HISTORY_LATENCY_WRITER">
            <!-- Configure History QoS as KEEP_LAST 20 -->
            <!-- History depth depends on the user application constraints (publication␣
↪rate for instance) -->
            <topic>
                <historyQos>
                    <kind>KEEP_LAST</kind>
                    <depth>20</depth>
                </historyQos>
            </topic>
            <!-- Enable pull mode -->
            <propertiesPolicy>
                <properties>
                    <property>
                        <name>fastdds.push_mode</name>
                        <value>false</value>
                    </property>
                </properties>
            </propertiesPolicy>
            <!-- Set durability, reliability, and publication mode -->
            <qos>
                <durability>
                    <kind>TRANSIENT_LOCAL</kind>
                </durability>

                <reliability>
                    <kind>RELIABLE</kind>
                </reliability>

                <publishMode>
                    <kind>ASYNCHRONOUS</kind>
                </publishMode>
            </qos>
        </data_writer>
    </profiles>
</dds>
```

Note: Increasing the History Depth of the statistics DataWriters has an impact on memory usage, as sufficient space is pre-allocated for each of the DataWriter's histories to hold that number of samples per topic instance.

6.25 XML profiles

eProsima Fast DDS allows for loading XML configuration files, each one containing one or more XML profiles. In addition to the API functions for loading user XML files, *Fast DDS* tries to locate and load several XML files upon initialization. *Fast DDS* offers the following options:

- Load an XML file named *DEFAULT_FASTRTPS_PROFILES.xml* located in the current execution path.
- Load an XML file which location is defined using the environment variable `FASTRTPS_DEFAULT_PROFILES_FILE` (see *FASTRTPS_DEFAULT_PROFILES_FILE*).
- Load the configuration parameters directly from the classes' definitions without looking for the *DEFAULT_FASTRTPS_PROFILES.xml* in the working directory (see *SKIP_DEFAULT_XML*).
- Load directly the XML as a string data buffer.

An XML profile is defined by a unique name that is used to reference the XML profile during the creation of an *Entity*, the *Transport* configuration, or the *DynamicTypes* definition.

Both options can be complemented, i.e. it is possible to load multiple XML files but these must not have XML profiles with the same name. This section explains how to configure DDS entities using XML profiles. This includes the description of all the configuration values available for each of the XML profiles, as well as how to create complete XML files.

6.25.1 Creating an XML profiles file

An XML file can contain several XML profiles. These XML profiles are defined within the `<dds>` element, and in turn, within the `<profiles>` XML elements. The possible topologies for the definition of XML profiles are specified in *Rooted vs Standalone profiles definition*.

It is worth mentioning that the first element of the xml profile must have the `xmlns` attribute with the link `xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles"`, in both rooted or standalone definitions. That link defines the reference of the `xsd` schema that the `xml` document complies with.

The available profile types are:

- *DomainParticipant profiles*,
- *DataWriter profiles*,
- *DataReader profiles*,
- *Topic profiles*,
- *Transport descriptors*,
- *Intra-process delivery profiles*,
- *Log profiles*, and
- *Dynamic Types profiles*.

The following sections will show implementation examples for each of these profiles.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="participant_profile">
            <!-- ... -->
        </participant>
```

(continues on next page)

(continued from previous page)

```
        <data_writer profile_name="datawriter_profile">
            <!-- ... -->
        </data_writer>

        <data_reader profile_name="datareader_profile">
            <!-- ... -->
        </data_reader>

        <topic profile_name="topic_profile">
            <!-- ... -->
        </topic>

        <transport_descriptors>
            <!-- ... -->
        </transport_descriptors>
    </profiles>

    <library_settings>
        <!-- ... -->
    </library_settings>

    <log>
        <!-- ... -->
    </log>

    <types>
        <!-- ... -->
    </types>
</dds>
```

Note: The *Example* section shows an XML file with all the possible configurations and profile types. This example is useful as a quick reference to look for a particular property and how to use it. The Fast DDS XSD scheme can be used as a quick reference too.

Loading and applying profiles

In case the user defines the *Entity* profiles via XML files, it is required to load these XML files using the *load_XML_profiles_file()* public member function before creating any entity. It is also possible to load directly the XML information as a string data buffer using the *load_XML_profiles_string()* public member function. Moreover, *create_participant_with_profile()*, *create_publisher_with_profile()*, *create_subscriber_with_profile()*, *create_datawriter_with_profile()*, and *create_datareader_with_profile()* member functions expect a profile name as an argument. *Fast DDS* searches the given profile name over all the loaded XML profiles, applying the profile to the entity if founded.

```
if (ReturnCode_t::RETCODE_OK ==
        DomainParticipantFactory::get_instance()->load_XML_profiles_file("my_profiles.xml
↪"))
{
```

(continues on next page)

(continued from previous page)

```
    DomainParticipant* participant =
            DomainParticipantFactory::get_instance()->create_participant_with_profile(
        0, "participant_xml_profile");

    Topic* topic =
            participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);

    Publisher* publisher = participant->create_publisher_with_profile("publisher_xml_
↪profile");
    DataWriter* datawriter = publisher->create_datawriter_with_profile(topic,
↪"datawriter_xml_profile");
    Subscriber* subscriber = participant->create_subscriber_with_profile("subscriber_xml_
↪profile");
    DataReader* datareader = subscriber->create_datareader_with_profile(topic,
↪"datareader_xml_profile");
}

// Load XML as string data buffer
std::string xml_profile =
        "\
        <?xml version=\"1.0\" encoding=\"UTF-8\" ?>\
        <dds>\
            <profiles xmlns=\"http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles\" >\
                <data_writer profile_name=\"test_publisher_profile\" is_default_profile=\
↪"true\">\
                    <qos>\
                        <durability>\
                            <kind>TRANSIENT_LOCAL</kind>\
                        </durability>\
                    </qos>\
                </data_writer>\
            </profiles>\
        </dds>\
        ";
if (ReturnCode_t::RETCODE_OK ==
        DomainParticipantFactory::get_instance()->load_XML_profiles_string(xml_profile.c_
↪str(),
        xml_profile.length()))
{
    // Create DDS entities with profiles
}
```

Warning: It is worth mentioning that if the same XML profile file is loaded multiple times, the second loading of the file will result in an error together with the consequent error log.

Note: To load dynamic types from XML files see the *Loading dynamic types in a Fast DDS application* subsection of *Dynamic Types profiles*.

Rooted vs Standalone profiles definition

Fast DDS offers various options for the definition of XML profiles. These options are:

- Stand-alone: The element defining the XML profile is the root element of the XML file. Elements <dds>, <profiles>, <library_settings>, <types>, and <log> can be defined in a stand-alone manner.
- Rooted: The element defining the XML profile is the child element of another element. For example, the <participant>, <data_reader>, <data_writer>, <topic>, and <transport_descriptors> elements must be defined as child elements of the <profiles> element.

The following is an example of the definition of the <types> XML profile using the two previously discussed approaches.

Stand-alone

```
<?xml version="1.0" encoding="UTF-8" ?>
<types xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <type>
        <!-- Type definition -->
    </type>

    <type>
        <!-- Type definition -->
        <!-- Type definition -->
    </type>
</types>
```

Rooted

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <types>
        <type>
            <!-- Type definition -->
        </type>

        <type>
            <!-- Type definition -->
            <!-- Type definition -->
        </type>
    </types>
</dds>
```

Note: Make sure that the first element of the xml profile must have the `xmlns` tag with the link `xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles"`, in both rooted or standalone definitions.

Modifying predefined XML profiles

Some scenarios may require to modify some of the QoS after loading the XML profiles. For such cases the *Types of Entities* which act as factories provide methods to get the QoS from the XML profile. This allows the user to read and modify predefined XML profiles before applying them to a new entity.

```
if (ReturnCode_t::RETCODE_OK ==
        DomainParticipantFactory::get_instance()->load_XML_profiles_file("my_profiles.xml
↪"))
{
    DomainParticipantQos participant_qos;
    DomainParticipantFactory::get_instance()->get_participant_qos_from_profile(
        "participant_xml_profile",
        participant_qos);

    // Name obtained in another section of the code
    participant_qos.name() = custom_name;

    // Modify number of preallocations (this overrides the one set in the XML profile)
    participant_qos.allocation().send_buffers.preallocated_number = 10;

    // Create participant using the modified XML Qos
    DomainParticipant* participant =
            DomainParticipantFactory::get_instance()->create_participant(
        0, participant_qos);
}
```

Dynamic content by leveraging environment variables

For deployment scenarios that require part of the XML content to be dynamically generated, Fast DDS supports using environment variables on the text content of any XML tag. The format for environment variables expansion is `${ENV_VAR_NAME}`, where `ENV_VAR_NAME` follows the restrictions from IEEE 1003.1:

Note: Environment variable names … consist solely of uppercase letters, digits, and the '_' (underscore) from the characters defined in Portable Character Set and do not begin with a digit.

More than one environment variable can be used, and they can be mixed with literal text.

The expansion will take place when the XML file is loaded, so changing the value of an environment variable afterwards will have no effect.

The following is an example of an XML allowing a participant to exclusively communicate with the participants on a fixed IP address, taken from `REMOTE_IP_ADDRESS` environment variable. It also gives the participant a name that mixes literal text with the content from two environment variables.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="env_var_parsing_example">
            <rtps>
                <name>This is app '${MY_APP_NAME}' running on host '${MY_HOST_NAME}'</
↪name>
```

(continues on next page)

(continued from previous page)

```
                <builtin>
                    <initialPeersList>
                        <locator>
                            <udpv4>
                                <address>${REMOTE_IP_ADDRESS}</address>
                            </udpv4>
                        </locator>
                    </initialPeersList>
                </builtin>
            </rtps>
        </participant>
    </profiles>
</dds>
```

Warning: The Fast DDS XSD schema does not support the environment variables expansion feature, so validation of an XML file with environment variables expansion expressions will fail.

6.25.2 DomainParticipant profiles

The *DomainParticipant* profiles allow the definition of the configuration of `DomainParticipants` through XML files. These profiles are defined within the <participant> XML tags.

DomainParticipant XML attributes

The <participant> element has two attributes defined: `profile_name` and `is_default_profile`.

Name	Description	Use
`profile_name`	Sets the name under which the <participant> profile is registered in the DDS Domain, so that it can be loaded later by the `DomainParticipantFactory`, as shown in *Loading and applying profiles*.	Mandatory
`is_default_profile`	Sets the <participant> profile as the default profile. Thus, if a default profile exists, it will be used when no other DomainParticipant profile is specified at the DomainParticipant's creation.	Optional

DomainParticipant configuration

The <participant> element has two child elements: <domainId> and <rtps>. All the DomainParticipant configuration options belong to the <rtps> element, except for the DDS `DomainId` which is defined by the <domainId> element. Below a list with the configuration XML elements is presented:

Name	Description	Values	Default
<domainId>	DomainId to be used by the DomainParticipant. See *Profile based creation of a DomainParticipant*.	`uint32_t`	0
<rtps>	*Fast DDS* DomainParticipant configurations. See *RTPS element type*.	*RTPS element type*	

RTPS element type

The following is a list with all the possible child XML elements of the <rtps> element. These elements allow the user to define the DomainParticipant configuration.

Name	Description	Values	Default
<name>	The DomainParticipant's name.	string_255	
<defaultUnicastLocatorList>	List of default reception unicast locators for user data traffic (see <metatrafficUnicastLocatorList> defined in *Builtin parameters*). It expects a *LocatorListType*.	<locator>	
<defaultMulticastLocatorList>	List of default reception multicast locators for user data traffic (see <metatrafficMulticastLocatorList> defined in *Builtin parameters*). It expects a *LocatorListType*.	<locator>	
<default_external_unicast_locators>	List of *ExternalLocators* to announce for the default user traffic of this participant.	*ExternalLocatorListType*	
<ignore_non_matching_locators>	When true, ignore locators received on announcements from other participants when they don't match with any of the locators announced by this participant.	bool	false
<sendSocketBufferSize>	Size in bytes of the send socket buffer. If the value is zero then *Fast DDS* will use the system default socket size.	uint32_t	0
<listenSocketBufferSize>	Size in bytes of the reception socket buffer. If the value is zero then *Fast DDS* will use the system default socket size.	uint32_t	0
<builtin>	*builtin* public data member of the *WireProtocolConfigQos* class. See the *Builtin parameters* section.	*Builtin parameters*	
<port>	Allows defining the port and gains related to the RTPS protocol. See the *Port* section.	*Port*	
<participantID>	DomainParticipant's identifier. Typically it will be automatically generated by the *DomainParticipantFactory*.	int32_t	0
<userTransports>	Transport descriptors to be used by the DomainParticipant. See *Transport descriptors*.	List <string>	
<useBuiltinTransports>	Boolean field to indicate the system whether the DomainParticipant will use the default *builtin* transports in addition to its <userTransports>.	bool	true
<builtinTransportsConfig>	Configuration option to determine which transports will be instantiated if the useBuiltinTransports is set to true. See *Managing the Builtin Transports*.	string_255	DEFAULT
<propertiesPolicy>	Additional configuration properties. See *PropertyPolicyQos*.	*PropertiesPolicyType*	
<allocation>	Configuration regarding allocation behavior. It expects a *DomainParticipantAllocationType*.	*DomainParticipantAllocationType*	
<userData>	Additional information attached to the DomainParticipant and transmitted with the discovery information. See *UserDataQosPolicy*.	List <string>	Empty
<prefix>	*DomainParticipant*'s *GuidPrefix_t* identifies peers running in the same process. Two participants with identical 8 first bytes on the *GuidPrefix_t* are considered to be running in the same process, and therefore intra-process delivery is used. See *Intra-process delivery*.	string	Empty

Example

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="domainparticipant_profile_name">
        <domainId>4</domainId>
        <rtps>
            <name>DomainParticipant Name</name>

            <defaultUnicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <port>7400</port>
                        <address>192.168.1.41</address>
                    </udpv4>
                </locator>
            </defaultUnicastLocatorList>

            <defaultMulticastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <port>7400</port>
                        <address>192.168.2.41</address>
                    </udpv4>
                </locator>
            </defaultMulticastLocatorList>

            <default_external_unicast_locators>
                <!-- EXTERNAL_LOCATOR_LIST -->
                <udpv4 externality="1" cost="0" mask="24">
                    <address>100.100.100.10</address>
                    <port>23456</port>
                </udpv4>
            </default_external_unicast_locators>

            <ignore_non_matching_locators>true</ignore_non_matching_locators>

            <sendSocketBufferSize>8192</sendSocketBufferSize>

            <listenSocketBufferSize>8192</listenSocketBufferSize>

            <builtin>
                <!-- BUILTIN -->
            </builtin>

            <port>
                <portBase>7400</portBase>
                <domainIDGain>200</domainIDGain>
                <participantIDGain>10</participantIDGain>
                <offsetd0>0</offsetd0>
                <offsetd1>1</offsetd1>
                <offsetd2>2</offsetd2>
                <offsetd3>3</offsetd3>
```

(continues on next page)

(continued from previous page)

```
            </port>

            <participantID>99</participantID>

            <userTransports>
                <transport_id>TransportId1</transport_id>
                <transport_id>TransportId2</transport_id>
            </userTransports>

            <useBuiltinTransports>false</useBuiltinTransports>

            <builtinTransports>DEFAULT</builtinTransports>

            <propertiesPolicy>
                <!-- PROPERTIES_POLICY -->
                <properties>
                    <property>
                        <name>Property1Name</name>
                        <value>Property1Value</value>
                        <propagate>false</propagate>
                    </property>
                </properties>
            </propertiesPolicy>

            <allocation>
                <!-- ALLOCATION -->
            </allocation>

            <userData>
                <value>3.4.7.0.C</value>
            </userData>

            <prefix>72.61.73.70.66.61.72.6d.74.65.73.74</prefix>
        </rtps>
    </participant>
</profiles>
```

Note:

- `LOCATOR_LIST` means a *LocatorListType* is expected.
- `EXTERNAL_LOCATOR_LIST` means a *ExternalLocatorListType* is expected.
- For `BUILTIN` details, please refer to *Builtin parameters*.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- For `ALLOCATION` details, please refer to *ParticipantAllocationType*.

Builtin parameters

By calling the *wire_protocol()* member function of the *DomainParticipantQos*, it is possible to access the *builtin* public data member of the *WireProtocolConfigQos* class. This section specifies the available XML members for the configuration of this *builtin* parameters.

Name	Description	Values	Default
<discovery_config>	This is the main element within which discovery-related settings can be configured. See *Discovery*.	*discovery_config*	
<avoid_builtin_multicast>	Restrict multicast metatraffic to PDP only.	bool	true
<use_WriterLiveliness Protocol>	Indicates whether to use the DataWriterLiveliness protocol.	bool	true
<metatrafficUnicastLocatorList>	Metatraffic Unicast Locator List.	A set of <locator> members. See *LocatorListType*	
<metatrafficMulticastLocatorList>	Metatraffic Multicast Locator List.	A set of <locator> members. See *LocatorListType*	
<initialPeersList>	The list of IP-port address pairs of all other *DomainParticipants* with which a *DomainParticipant* will communicate. See *Initial peers*	A set of <locator> members. See *LocatorListType*	
<metatraffic_external_unicast_locators>	List of *ExternalLocators* to announce for the metatraffic of this participant.	*ExternalLocatorListType*	
<readerHistoryMemoryPolicy>	Memory policy for DataReaders. See *HistoryQosPolicyKind*.	*HistoryMemoryPolicy*	*PREALLOCATED*
<writerHistoryMemoryPolicy>	Memory policy for DataWriters. See *HistoryQosPolicyKind*.	*HistoryMemoryPolicy*	*PREALLOCATED*
<readerPayloadSize>	Maximum DataReader's History payload size. Allows to reserve all the required memory at DataReader initialization. See *MemoryManagementPolicy*.	uint32_t	512
<writerPayloadSize>	Maximum DataWriter's History payload size. Allows to reserve all the required memory at DataWriter initialization. See *MemoryManagementPolicy*.	uint32_t	512
<mutation_tries>	Number of different ports to try if DataReader's physical port is already in use.	uint32_t	100
typelookup_config	TypeLookup Service settings. See *TypeLookup Service*.	*TypeLookup Service Configuration*	

Example

```
<builtin>
    <discovery_config>
        <discoveryProtocol>CLIENT</discoveryProtocol>

        <discoveryServersList>
            <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.74">
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.57</address>
                            <port>56542</port>
                        </udpv4>
```

(continues on next page)

(continued from previous page)

```
                </locator>
            </metatrafficUnicastLocatorList>
            <metatrafficMulticastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.10.58</address>
                        <port>24565</port>
                    </udpv4>
                </locator>
            </metatrafficMulticastLocatorList>
        </RemoteServer>
    </discoveryServersList>

    <ignoreParticipantFlags>FILTER_DIFFERENT_HOST</ignoreParticipantFlags>

    <EDP>SIMPLE</EDP>

    <simpleEDP>
        <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
        <PUBREADER_SUBWRITER>true</PUBREADER_SUBWRITER>
    </simpleEDP>

    <leaseDuration>
        <!-- DURATION -->
        <sec>20</sec>
    </leaseDuration>

    <leaseAnnouncement>
        <!-- DURATION -->
        <sec>3</sec>
    </leaseAnnouncement>

    <initialAnnouncements>
        <!-- INITIAL_ANNOUNCEMENTS -->
        <count>10</count>
        <period>
            <nanosec>50</nanosec>
        </period>
    </initialAnnouncements>

    <clientAnnouncementPeriod>
        <nanosec>250000000</nanosec>
    </clientAnnouncementPeriod>

    <static_edp_xml_config>file://filename1.xml</static_edp_xml_config>
    <static_edp_xml_config>file://filename2.xml</static_edp_xml_config>
    <static_edp_xml_config>file://filename3.xml</static_edp_xml_config>
</discovery_config>

<avoid_builtin_multicast>true</avoid_builtin_multicast>

<use_WriterLivelinessProtocol>false</use_WriterLivelinessProtocol>
```

(continues on next page)

(continued from previous page)

```
    <metatrafficUnicastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udpv4>
                <address>192.168.0.1</address>
            </udpv4>
        </locator>
    </metatrafficUnicastLocatorList>

    <metatrafficMulticastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udpv4>
                <address>192.168.0.1</address>
            </udpv4>
        </locator>
    </metatrafficMulticastLocatorList>

    <initialPeersList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udpv4>
                <address>192.168.0.1</address>
            </udpv4>
        </locator>
    </initialPeersList>

    <metatraffic_external_unicast_locators>
        <!-- EXTERNAL_LOCATOR_LIST -->
        <udpv4 externality="1" cost="0" mask="24">
            <address>100.100.100.10</address>
            <port>34567</port>
        </udpv4>
    </metatraffic_external_unicast_locators>

    <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</readerHistoryMemoryPolicy>

    <writerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</writerHistoryMemoryPolicy>

    <readerPayloadSize>512</readerPayloadSize>

    <writerPayloadSize>512</writerPayloadSize>

    <mutation_tries>55</mutation_tries>

    <typelookup_config>
        <use_client>true</use_client>
        <use_server>true</use_server>
    </typelookup_config>
</builtin>
```

discovery_config

Through the <discovery_config> element, *Fast DDS* allows the configuration of the discovery mechanism via an XML file. Please refer to the *Discovery* section for more detail on the various types of discovery mechanisms and configurable settings.

Name	Description	Values	Default
<discoveryProtocol>	Indicates which discovery protocol the DomainParticipant will use. See *Discovery mechanisms*. If set to *CLIENT*, <discoveryServersList> element would be used.	*SIMPLE* *CLIENT* *SERVER* *BACKUP* *NONE*	*SIMPLE*
<discoveryServersList>	Describes servers from which it receives only the discovery information they require to establish communication with matching endpoints. See *Discovery Server Settings*	*discoveryServersList*	
<ignoreParticipantFlags>	Restricts traffic using several filtering criteria. See *Ignore Participant flags*.	*ignoreParticipantFlags*	*NO_FILTER*
<EDP>	If set to *SIMPLE*, <simpleEDP> element would be used. If set to STATIC, EDPStatic will be performed, configured with the contents of the XML file set in <static_edp_xml_config>. See *Discovery*.	*SIMPLE* STATIC	*SIMPLE*
<simpleEDP>	Attributes of the Simple Discovery Protocol. See *Simple EDP Attributes*.	*simpleEDP*	
<leaseDuration>	Indicates how long the DomainParticipant should consider remote DomainParticipants alive. See *Lease Duration*.	*DurationType*	20s
<leaseAnnouncement>	The period for the DomainParticipant to send its discovery message to all other discovered DomainParticipants as well as to all Multicast ports. See *Announcement Period*.	*DurationType*	3s
<initialAnnouncements>	Allows the user to configure the number and period of the DomainParticipant's initial discovery messages. See *Initial Announcements*.	*InitialAnnouncements*	
<clientAnnouncementPeriod>	The period for the DomainParticipant to send its Discovery Message to its servers and check for EDP endpoints matching.	*DurationType*	450 ms
<static_edp_xml_config>	The XML file name(s) with the static EDP configuration. Only necessary if the <EDP> member is set to STATIC. See *STATIC Discovery Settings*.	List <string>	

discoveryServersList

Contains a list of <RemoteServer> (*server*) elements, which are defined by the *GuidPrefix_t* and their own locator list which must be populated with RemoteServerAttributes. The DomainParticipant set as *client* would discover the *servers* described in this section.

Server attributes

The <RemoteServer> element has a mandatory attribute defined: prefix.

Name	Description	Use
prefix	RTPS standard participant unique identifier, a 12-byte chain. This identifier allows clients to assess whether they are receiving messages from the right server.	Mandatory

RemoteServer configuration

Each *client* (or a *server* connecting to another *server*) must keep a list of locators associated to the *servers* to which it wants to link. Those locator would be defined as `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList`.

Name	Description	Values
`<metatrafficUnicastLocatorList>`	Metatraffic Unicast Locator List.	A set of `<locator>` members. See *LocatorListType*
`<metatrafficMulticastLocatorList>`	Metatraffic Multicast Locator List.	A set of `<locator>` members. See *LocatorListType*

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="UDP CLIENT" is_default_profile="true">
        <rtps>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>CLIENT</discoveryProtocol>
                    <discoveryServersList>
                        <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.74">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.57</address>
                                        <port>56542</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                            <metatrafficMulticastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.58</address>
                                        <port>24565</port>
                                    </udpv4>
                                </locator>
                            </metatrafficMulticastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

ignoreParticipantFlags

Possible values	Description
NO_FILTER	All Discovery traffic is processed.
FILTER_DIFFERENT_HOST	Discovery traffic from another host is discarded.
FILTER_DIFFERENT_PROCESS	Discovery traffic from another process on the same host is discarded.
FILTER_SAME_PROCESS	Discovery traffic from DomainParticipant's own process is discarded.

This option also supports the OR (|) operator to filter discovery traffic from other configurations. For instance, `FILTER_DIFFERENT_PROCESS|FILTER_SAME_PROCESS` value discards discovery traffic from the DomainParticipant's own host.

simpleEDP

Name	Description	Values	Default
<PUBWRITER_SUBREADER>	Indicates if the participant must use Publication DataWriter and Subscription DataReader.	`bool`	`true`
<PUBREADER_SUBWRITER>	Indicates if the participant must use Publication DataReader and Subscription DataWriter.	`bool`	`true`

Initial Announcements

Name	Description	Values	Default
<count>	Number of initial discovery messages to send at the period specified by <period>. After these announcements, the DomainParticipant will continue sending its discovery messages at the <leaseAnnouncement> rate.	`uint32_t`	5
<period>	The period for the DomainParticipant to send its discovery messages.	*DurationType*	100 ms

TypeLookup Service Configuration

Name	Description	Values	Default
`use_client`	Create TypeLookup Service client builtin endpoints.	`bool`	`false`
`use_server`	Create TypeLookup Service server builtin endpoints.	`bool`	`false`

Port Configuration

According to the RTPS standard (Section 9.6.1.1), the *RTPSParticipants*' discovery traffic unicast listening ports are calculated using the following equation: $7400 + 250 * DomainId + 10 + 2 * ParticipantId$. Therefore the following parameters can be specified:

Name	Description	Values	Default
<portBase>	Base port.	uint16_t	7400
<domainIDGain>	Gain in DomainId.	uint16_t	250
<participantIDGain>	Gain in *participant_id*.	uint16_t	2
<offsetd0>	Multicast metadata offset.	uint16_t	0
<offsetd1>	Unicast metadata offset.	uint16_t	10
<offsetd2>	Multicast user data offset.	uint16_t	1
<offsetd3>	Unicast user data offset.	uint16_t	11

Warning: Changing these default parameters may break compatibility with other RTPS compliant implementations, as well as with other *Fast DDS* applications with default port settings.

ParticipantAllocationType

The ParticipantAllocationType defines the <allocation> element, which allows setting of the parameters related with the allocation behavior on the DomainParticipant. Please refer to *ParticipantResourceLimitsQos* for a detailed documentation on DomainParticipants allocation configuration.

Name	Description	Values	Default
<remote_locators>	Defines the limits for the remote locators' collections. See *RemoteLocatorsAllocationAttributes*.	*Remote Locators Allocations*	
<total_participants>	DomainParticipant *Allocation Configuration* to specify the total number of DomainParticipants in the domain (local and remote). See *ResourceLimitedContainerConfig*.	*Allocation Configuration*	
<total_readers>	DomainParticipant *Allocation Configuration* to specify the total number of DataReader on each DomainParticipant (local and remote). See *ResourceLimitedContainerConfig*.	*Allocation Configuration*	
<total_writers>	DomainParticipant *Allocation Configuration* related to the total number of DataWriters on each DomainParticipant (local and remote). See *ResourceLimitedContainerConfig*.	*Allocation Configuration*	
<max_partitions>	Maximum size of the partitions submessage. Set to zero for no limit.	uint32_t	
<max_user_data>	Maximum size of the user data submessage. Set to zero for no limit.	uint32_t	
<max_properties>	Maximum size of the properties submessage. Set to zero for no limit.	uint32_t	
<send_buffers>	Allocation behaviour for the send buffer manager.	*Send buffers*	

Example

```
<allocation>
    <remote_locators>
        <max_unicast_locators>4</max_unicast_locators>
        <max_multicast_locators>1</max_multicast_locators>
```

(continues on next page)

(continued from previous page)

```
    </remote_locators>

    <total_participants>
        <initial>0</initial>
        <maximum>0</maximum>
        <increment>1</increment>
    </total_participants>

    <total_readers>
        <initial>0</initial>
        <maximum>0</maximum>
        <increment>1</increment>
    </total_readers>

    <total_writers>
        <initial>0</initial>
        <maximum>0</maximum>
        <increment>1</increment>
    </total_writers>

    <max_partitions>256</max_partitions>

    <max_user_data>256</max_user_data>

    <max_properties>512</max_properties>

    <send_buffers>
        <preallocated_number>127</preallocated_number>
        <dynamic>true</dynamic>
    </send_buffers>

    <!-- content_filter cannot be configured using XML (yet) -->
</allocation>
```

Remote Locators Allocations

Name	Description	Values	Default
<max_unicast_locators>	Maximum number of unicast locators expected on a remote entity. It is recommended to use the maximum number of network interfaces available on the machine on which DomainParticipant is running. See *RemoteLocatorsAllocationAttributes*.	uint32_t	4
<max_multicast_locators>	Maximum number of multicast locators expected on a remote entity. May be set to zero to disable multicast traffic. See *RemoteLocatorsAllocationAttributes*.	uint32_t	1

Send buffers

Name	Description	Values	Default
<preallocated_number>	Initial number of send buffers to allocate. See *SendBuffersAllocationAttributes*.	uint32_t	0
<dynamic>	Whether the number of send buffers is allowed to grow. See *SendBuffersAllocationAttributes*.	bool	false

Note: The default value 0 of <preallocated_number> will perform an initial guess of the number of buffers required, based on the number of threads from which a send operation could be started. So it does not mean there are no buffers, instead it would use the maximum amount of buffers available.

6.25.3 DataWriter profiles

The DataWriter profiles allow for configuring *DataWriters* from an XML file. These profiles are defined within the <data_writer> XML tags.

DataWriter XML attributes

The <data_writer> element has two attributes defined: profile_name and is_default_profile.

Name	Description	Use
profile_name	Sets the name under which the <data_writer> profile is registered in the DDS Domain, so that it can be loaded later by the *DomainParticipant*, as shown in *Loading and applying profiles*.	Mandatory
is_default_profile	Sets the <data_writer> profile as the default profile. Thus, if a default profile exists, it will be used when no other DataWriter profile is specified at the DataWriter's creation.	Optional

DataWriter configuration

The DataWriter configuration is performed through the XML elements listed in the following table.

Name	Description	Values	Default
<topic>	*TopicType* configuration of the DataWriter.	*TopicType*	
<qos>	DataWriter *QoS* configuration.	*QoS*	
<times>	It configures some time related parameters of the DataWriter.	*WriterTimes*	
<unicastLocatorList>	List of input unicast locators. It expects a *LocatorListType*.	<locator>	
<multicastLocatorList>	List of input multicast locators. It expects a *LocatorListType*.	<locator>	
<external_unicastLocators>	List of *External Locators* to announce for the communication with this DataWriter.	*ExternalLocatorListType*	
<ignore_non_matching_locators>	Whether to ignore locators received on announcements from other entities when they don't match with any of the locators announced by this DataWriter.	bool	false
<historyMemoryPolicy>	Memory allocation kind for DataWriter's history. See *HistoryQosPolicyKind*.	*HistoryMemoryPolicy*	*PREALLOCATED*
<propertiesPolicy>	Additional configuration properties.	*PropertiesPolicyType*	
<userDefinedID>	Used for EDPStatic.	int16_t	-1
<entityID>	Sets the *entity_id* of the *RTPSEndpointQos* class.	int16_t	-1
<matchedSubscribersAllocation>	Sets the limits of the collection of matched DataReaders. See *ParticipantResourceLimitsQos*.	*AllocationConfiguration*	

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <data_writer profile_name="datawriter_profile_name">
            <topic>
                <historyQos>
                    <kind>KEEP_LAST</kind>
                </historyQos>
            </topic>

            <qos>
                <!-- QOS -->
            </qos>

            <times> <!-- writerTimesType -->
                <initialHeartbeatDelay>
                    <nanosec>12</nanosec>
                </initialHeartbeatDelay>
```

(continues on next page)

(continued from previous page)

```
            <heartbeatPeriod>
                <sec>3</sec>
            </heartbeatPeriod>

            <nackResponseDelay>
                <nanosec>5</nanosec>
            </nackResponseDelay>

            <nackSupressionDuration>
                <sec>0</sec>
            </nackSupressionDuration>
        </times>

        <unicastLocatorList>
            <!-- LOCATOR_LIST -->
            <locator>
                <udpv4>
                    <address>192.168.0.1</address>
                </udpv4>
            </locator>
        </unicastLocatorList>

        <multicastLocatorList>
            <!-- LOCATOR_LIST -->
            <locator>
                <udpv4>
                    <address>192.168.0.1</address>
                </udpv4>
            </locator>
        </multicastLocatorList>

        <external_unicast_locators>
            <!-- EXTERNAL_LOCATOR_LIST -->
            <udpv4 externality="1" cost="0" mask="24">
                <address>100.100.100.10</address>
                <port>12345</port>
            </udpv4>
        </external_unicast_locators>

        <ignore_non_matching_locators>true</ignore_non_matching_locators>

        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

        <!-- PROPERTIES_POLICY -->
        <propertiesPolicy>
            <properties>
                <property>
                    <name>PropertyName</name>
                    <value>PropertyValue</value>
                </property>
            </properties>
        </propertiesPolicy>
```

(continues on next page)

(continued from previous page)

```
            <userDefinedID>55</userDefinedID>

            <entityID>66</entityID>

            <matchedSubscribersAllocation>
                <initial>0</initial>
                <maximum>0</maximum>
                <increment>1</increment>
            </matchedSubscribersAllocation>

            <!-- reader_filters_allocation cannot be configured using XML (yet) -->
        </data_writer>
    </profiles>
</dds>
```

Note:

- `LOCATOR_LIST` means a *LocatorListType* is expected.
- `EXTERNAL_LOCATOR_LIST` means a *ExternalLocatorListType* is expected.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- For `QOS` details, please refer to *QoS*.
- `TOPIC_TYPE` is detailed in section *TopicType*.

WriterTimes

These parameters are included within *RTPSReliableWriterQos* in the *WriterTimes* structure.

Name	Description	Values	Default
`<initialHeartbeatDelay>`	Initial heartbeat delay.	*DurationType*	12 ms
`<heartbeatPeriod>`	Periodic heartbeat period.	*DurationType*	3 s
`<nackResponseDelay>`	Delay to apply to the response of an ACKNACK message.	*DurationType*	5 ms
`<nackSupressionDuration>`	This time allows the DataWriter to ignore NACK messages for a given period of time right after the data has been sent.	*DurationType*	0 ms

6.25.4 DataReader profiles

The DataReader profiles allow declaring *DataReaders* from an XML file. These profiles are defined within the <data_reader> XML tags.

DataReader XML attributes

The <data_reader> element has two attributes defined: profile_name and is_default_profile.

Name	Description	Use
profile_name	Sets the name under which the <data_reader> profile is registered in the DDS Domain, so that it can be loaded later by the *DomainParticipant*, as shown in *Loading and applying profiles*.	Mandatory
is_default_profile	Sets the <data_reader> profile as the default profile. Thus, if a default profile exists, it will be used when no other DataReader profile is specified at the DataReader's creation.	Optional

DataReader configuration

The DataReader configuration is performed through the XML elements listed in the following table.

Name	Description	Values	Default
<topic>	*TopicType* configuration of the DataReader.	*TopicType*	
<qos>	Subscriber *QoS* configuration.	*QoS*	
<times>	It allows configuring some time related parameters of the DataReader.	*ReaderTimes*	
<unicastLocatorList>	List of input unicast locators. It expects a *LocatorListType*.	List of *LocatorListType*	
<multicastLocatorList>	List of input multicast locators. It expects a *LocatorListType*.	List of *LocatorListType*	
<external_unicast_locators>	List of *External Locators* to announce for the communication with this DataReader.	*ExternalLocatorListType*	
<ignore_non_matching_locators>	Whether to ignore locators received on announcements from other entities when they don't match with any of the locators announced by this DataReader.	bool	false
<expectsInlineQos>	It indicates if QoS is expected inline.	bool	false
<historyMemoryPolicy>	Memory allocation kind for DataReaders's history.	*HistoryMemoryPolicy*	*PREALLOCATED*
<propertiesPolicy>	Additional configuration properties.	*PropertiesPolicyType*	
<userDefinedID>	Used for StaticEndpointDiscovery.	int16_t	-1
<entityID>	Set the *entity_id* of the *RTPSEndpointQos* class.	int16_t	-1
<matchedPublishersAllocation>	Allocation of the collection of matched DataWriters. See *ParticipantResourceLimitsQos*.	*AllocationConfiguration*	

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <data_reader profile_name="data_reader_profile_name">
            <topic>
                <historyQos>
                    <kind>KEEP_LAST</kind>
                </historyQos>
            </topic>

            <qos>
                <!-- QOS -->
            </qos>

            <times> <!-- readerTimesType -->
                <initialAcknackDelay>
                    <nanosec>70</nanosec>
                </initialAcknackDelay>

                <heartbeatResponseDelay>
                    <nanosec>5</nanosec>
                </heartbeatResponseDelay>
            </times>

            <unicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <address>192.168.0.1</address>
                    </udpv4>
                </locator>
            </unicastLocatorList>

            <multicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <address>192.168.0.1</address>
                    </udpv4>
                </locator>
            </multicastLocatorList>

            <external_unicast_locators>
                <!-- EXTERNAL_LOCATOR_LIST -->
                <udpv4 externality="1" cost="0" mask="24">
                    <address>100.100.100.10</address>
                    <port>12345</port>
                </udpv4>
            </external_unicast_locators>

            <ignore_non_matching_locators>true</ignore_non_matching_locators>
```

(continues on next page)

(continued from previous page)

```
            <expectsInlineQos>true</expectsInlineQos>

            <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

            <!-- PROPERTIES_POLICY -->
            <propertiesPolicy>
                <properties>
                    <property>
                        <name>PropertyName</name>
                        <value>PropertyValue</value>
                    </property>
                </properties>
            </propertiesPolicy>

            <userDefinedID>55</userDefinedID>

            <entityID>66</entityID>

            <matchedPublishersAllocation>
                <initial>0</initial>
                <maximum>0</maximum>
                <increment>1</increment>
            </matchedPublishersAllocation>
        </data_reader>
    </profiles>
</dds>
```

Note:

- `LOCATOR_LIST` means it expects a *LocatorListType*.
- `EXTERNAL_LOCATOR_LIST` means a *ExternalLocatorListType* is expected.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- For `QOS` details, please refer to *QoS*.
- `TOPIC_TYPE` is detailed in section *TopicType*.

ReaderTimes

These parameters are included within *RTPSReliableReaderQos* in the *ReaderTimes* structure.

Name	Description	Values	Default
`<initialAcknackDelay>`	Initial ACKNACK delay.	*DurationType*	70 ms
`<heartbeatResponseDelay>`	Response time delay when receiving a Heartbeat.	*DurationType*	5 ms

6.25.5 Topic profiles

The topic profiles allow for configuring *Topic* from an XML file. These profiles are defined within the <topic> XML tags.

Topic XML attributes

The <topic> element has two attributes defined: profile_name and is_default_profile.

Name	Description	Use
profile_name	Sets the name under which the <topic> profile is registered in the DDS Domain, so that it can be loaded later by the *DataWriter* or the *DataReader*	Mandatory
is_default_profile	Sets the <topic> profile as the default profile. Thus, if a default profile exists, it will be used when no other Topic profile is specified at the Topic's creation.	Optional

Topic configuration

This XML element allows the configuration of the *TopicQos*.

Name	Description	Values
<historyQos>	It controls the behavior of *Fast DDS* when the value of an instance changes before it is finally communicated to some of its existing DataReaders.	*HistoryQoS*
<resourceLimitsQos>	It controls the resources that *Fast DDS* can use in order to meet the requirements imposed by the application and other QoS settings.	*ResourceLimitsQos*

Example

```
<topic profile_name="topic_example">
    <historyQos>
        <kind>KEEP_LAST</kind>
        <depth>20</depth>
    </historyQos>
    <resourceLimitsQos>
        <max_samples>5</max_samples>
        <max_instances>2</max_instances>
        <max_samples_per_instance>1</max_samples_per_instance>
        <allocated_samples>20</allocated_samples>
        <extra_samples>10</extra_samples>
    </resourceLimitsQos>
</topic>
```

6.25.6 Transport descriptors

This section defines the XML elements available for configuring the transport layer parameters in *Fast DDS*. These elements are defined within the XML tag <transports_descriptors>. The <transport_descriptors> can contain one or more <transport_descriptor> XML elements. Each <transport_descriptor> element defines a configuration for a specific type of transport protocol. Each of these <transport_descriptor> elements are uniquely identified by a transport ID with the <transport_id> XML tag. Once the user defines a valid <transports_descriptor>, i.e. defines the transport layer parameters, these can be loaded into the XML profile of the DomainParticipant using the <transport_id> XML tag. An example of how to load the <transport_descriptor> into the XML profile of the DomainParticipant is found in *DomainParticipant profiles*.

The following table lists all the available XML elements that can be defined within the <transport_descriptor> element for the configuration of the transport layer. A more detailed explanation of each of these elements can be found in *Transport Layer*.

Name	Description	Values	Default
<transport_id>	Unique name to identify each transport descriptor.	string	
<type>	Type of the transport descriptor.	UDPv4	UDPv4
		UDPv6	
		TCPv4	
		TCPv6	
		SHM	
<sendBufferSize>	Size in bytes of the send socket buffer. If the value is zero then *Fast DDS* will use the system default socket size.	uint32_t	0
<receiveBufferSize>	Size in bytes of the reception socket buffer. If the value is zero then *Fast DDS* will use the system default socket size.	uint32_t	0
<maxMessageSize>	The maximum size in bytes of the transport's message buffer.	uint32_t	65500
<maxInitialPeersRange>	Number of channels opened with each initial remote peer.	uint32_t	4
<interfaceWhiteList>	Allows defining an interfaces *Whitelist*.	*Whitelist*	
<TTL>	*Time To Live* (**UDP only**). See *UDP Transport*.	uint8_t	1
<non_blocking_send>	Whether to set the non-blocking send mode on the socket (**UDP only**). See *UDPTransportDescriptor*.	bool	false
<output_port>	Port used for output bound. If this field isn't defined, the output port will be random (**UDP only**).	uint16_t	0
<wan_addr>	Public WAN address when using **TCPv4 transports**. This field is optional if the transport doesn't need to define a WAN address (**TCPv4 only**).	IPv4 formatted string: XXX.XXX.XXX.XXX	
<keep_alive_frequency_ms>	Frequency in milliseconds for sending *RTCP* keep-alive requests (**TCP only**).	uint32_t	50000
<keep_alive_timeout_ms>	Time out in milliseconds since the last keep-alive request was sent to consider a connection as broken (**TCP only**).	uint32_t	10000
<max_logical_port>	The maximum number of logical ports to try during *RTCP* negotiations (**TCP only**).	uint16_t	100
<logical_port_range>	The maximum number of logical ports per request to try during *RTCP* negotiations (**TCP only**).	uint16_t	20
<logical_port_increment>	Increment between logical ports to try during *RTCP* negotiation (**TCP only**).	uint16_t	2
<listening_ports>	Local port to work as TCP acceptor for input connections. If not set, the transport will work as TCP client only. If set to 0, an available port will be automatically assigned (**TCP only**).	List <uint16_t>	
<tls>	Allows to define TLS related parameters and options (**TCP only**).	*TLS Configuration*	
<calculate_crc>	Calculates the Cyclic Redundancy Code (CRC) for error control (**TCP only**).	bool	true
<check_crc>	Check the CRC for error control (**TCP only**).	bool	true
<enable_tcp_nodelay>	Socket option for disabling the Nagle algorithm. (**TCP only**).	bool	false
<segment_size>	Size (in bytes) of the shared-memory segment. (Optional, **SHM only**).	uint32_t	262144
<port_queue_capacity>	Capacity (in number of messages) available to every Listener (Optional, **SHM only**).	uint32_t	512
<healthy_check_timeout_ms>	Maximum time out (in milliseconds) used when checking whether a Listener is alive (Optional, **SHM only**).	uint32_t	1000
<rtps_dump_file>	Complete path (including file) where RTPS messages will be stored for debugging purposes. An empty string indicates no trace will be performed (Optional, **SHM only**).	string	Empty

The following XML code shows an example of transport protocol configuration using all configurable parameters.

More examples of transports descriptors can be found in the *Transport Layer* section.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>my_udpv4_transport</transport_id>
                <type>UDPv4</type>
                <sendBufferSize>8192</sendBufferSize>
                <receiveBufferSize>8192</receiveBufferSize>
                <maxMessageSize>16384</maxMessageSize>
                <maxInitialPeersRange>100</maxInitialPeersRange>
                <interfaceWhiteList>
                    <address>192.168.1.41</address>
                    <address>127.0.0.1</address>
                </interfaceWhiteList>
                <TTL>250</TTL>
                <non_blocking_send>false</non_blocking_send>
                <output_port>5101</output_port>
                <wan_addr>80.80.55.44</wan_addr>
                <keep_alive_frequency_ms>5000</keep_alive_frequency_ms>
                <keep_alive_timeout_ms>25000</keep_alive_timeout_ms>
                <max_logical_port>9000</max_logical_port>
                <logical_port_range>100</logical_port_range>
                <logical_port_increment>2</logical_port_increment>
                <listening_ports>
                    <port>5100</port>
                    <port>5200</port>
                </listening_ports>
                <tls><!-- TLS Section --></tls>
                <calculate_crc>false</calculate_crc>
                <check_crc>false</check_crc>
                <enable_tcp_nodelay>false</enable_tcp_nodelay>
                <segment_size>262144</segment_size>
                <port_queue_capacity>512</port_queue_capacity>
                <healthy_check_timeout_ms>1000</healthy_check_timeout_ms>
                <rtps_dump_file>rtsp_messages.log</rtps_dump_file>
            </transport_descriptor>
        </transport_descriptors>
    </profiles>
</dds>
```

Note: The Real-time Transport Control Protocol (RTCP) is the control protocol for communications with RTPS over TCP/IP connections.

TLS Configuration

Fast DDS provides mechanisms to configure the Transport Layer Security (TLS) protocol parameters through the <tls> XML element of its <transport_descriptor>. Please, refer to *TLS over TCP* for a detailed explanation of the entire TLS configuration in *Fast DDS*. More information on how to set up secure communication in *Fast DDS* can be found in the *Security* section.

Warning: For the full understanding of this section, a basic knowledge of network security in terms of SSL/TLS, Certificate Authority (CA), Public Key Infrastructure (PKI), and Diffie-Hellman is required; encryption protocols are not explained in detail.

The full list of available XML elements that can be defined within the <tls> element to configure the TLS protocol are listed in the following table:

Name	Description	Values	Default
<password>	Password of the <private_key_file> or <rsa_private_key_file> if provided.	string	
<private_key_file>	Path to the private key certificate file.	string	
<rsa_private_key_file>	Path to the private key RSA certificate file.	string	
<cert_chain_file>	Path to the public certificate chain file.	string	
<tmp_dh_file>	Path to the Diffie-Hellman parameters file	string	
<verify_file>	Path to the Certification Authority (CA) file.	string	
<verify_mode>	Establishes the verification mode mask. Several verification options can be combined in the same <transport_descriptor>.	VERIFY_NONE	
		VERIFY_PEER	
		VERIFY_FAIL_IF_NO_PEER_CERT	
		VERIFY_CLIENT_ONCE	
<options>	Establishes the SSL Context options mask. Several options can be combined in the same <transport_descriptor>.	DEFAULT_WORKAROUNDS	
		NO_COMPRESSION	
		NO_SSLV2	
		NO_SSLV3	
		NO_TLSV1	
		NO_TLSV1_1	
		NO_TLSV1_2	
		NO_TLSV1_3	
		SINGLE_DH_USE	
<verify_paths>	Paths where the system will look for verification files.	List <string>	
<verify_depth>	Maximum allowed depth to verify intermediate certificates.	uint32_t	
<default_verify_paths>	Specifies whether the system will look on the default paths for the verification files.	bool	false
<handshake_role>	Role that the transport will take on handshaking. On default, the acceptors act as SERVER and the connectors as CLIENT.	DEFAULT	DEFAULT
		SERVER	
		CLIENT	
<server_name>	server name or host name required in case Server Name Indication (SNI) is used.	string	

An example of TLS protocol parameter configuration is shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
```

(continues on next page)

(continued from previous page)

```
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>Test</transport_id>
                <type>TCPv4</type>
                <tls>
                    <password>Password</password>
                    <private_key_file>Key_file.pem</private_key_file>
                    <rsa_private_key_file>RSA_file.pem</rsa_private_key_file>
                    <cert_chain_file>Chain.pem</cert_chain_file>
                    <tmp_dh_file>DH.pem</tmp_dh_file>
                    <verify_file>verify.pem</verify_file>
                    <verify_mode>
                        <verify>VERIFY_PEER</verify>
                    </verify_mode>
                    <options>
                        <option>NO_TLSV1</option>
                        <option>NO_TLSV1_1</option>
                    </options>
                    <verify_paths>
                        <verify_path>Path1</verify_path>
                        <verify_path>Path2</verify_path>
                        <verify_path>Path3</verify_path>
                    </verify_paths>
                    <verify_depth>55</verify_depth>
                    <default_verify_path>true</default_verify_path>
                    <handshake_role>SERVER</handshake_role>
                    <server_name>my_server.com</server_name>
                </tls>
            </transport_descriptor>
<!-->
    </profiles>
```

6.25.7 Intra-process delivery profiles

This section defines the XML elements available for configuring the *Intra-process delivery* settings parameters in *Fast DDS*. These elements are defined within the XML tag `<library_settings>`.

Intra-process delivery configuration

The Intra-process delivery configuration is performed through the XML elements listed in the following table.

Name	Description	Values	Default

`<intraprocess_delivery>`	Enable communications between entities within the same process by avoiding any of the overhead involved in the transport layer.	`OFF`
		`USER_DATA_ONLY`
		`FULL`
			`OFF`

Example

```
    <library_settings>
        <intraprocess_delivery>FULL</intraprocess_delivery> <!-- OFF | USER_DATA_ONLY |␣
↪FULL -->
    </library_settings>
```

6.25.8 Log profiles

eProsima Fast DDS allows for registering and configuring *Log consumers* using XML configuration files. Please refer to *Logging* for more information on *Fast DDS* extensible Logging built-in module. The logging profiles are defined within the <log> XML tags. The <log> element has two child elements: <use_default> and <consumer>. These are described in the following table.

Name	Description	Values	Default
<use_default>	If set to FALSE, a call to *Log::ClearConsumers()* is performed. See *Register Consumers*.	bool	true
<consumer>	Defines the class and configuration of the consumer to be registered. Multiple consumers can be registered this way. See *Consumers*.	*ConsumerDataType*	

The following constitutes an example of an XML configuration file that sets the *Log* to use one *StdoutConsumer*, one *StdoutErrConsumer*, and one *FileConsumer*:

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <log>
        <!--
        Clear consumers
        -->
        <use_default>false</use_default>

        <!--
        StdoutConsumer does not have any properties
        -->
        <consumer>
            <class>StdoutConsumer</class>
        </consumer>

        <!--
        StdoutErrConsumer with threshold set to Log::Kind::Error
        -->
        <consumer>
            <class>StdoutErrConsumer</class>
            <property>
                <name>stderr_threshold</name>
                <value>Log::Kind::Error</value>
            </property>
        </consumer>

        <!--
        FileConsumer openning "execution.log" in append mode
        -->
```

(continues on next page)

(continued from previous page)

```
        <consumer>
            <class>FileConsumer</class>
            <property>
                <name>filename</name>
                <value>execution.log</value>
            </property>
            <property>
                <name>append</name>
                <value>true</value>
            </property>
        </consumer>
    </log>
</dds>
```

ConsumerDataType

Name	Description	Values
<class>	The class of the consumer.	StdoutConsumer StdoutErrConsumer FileConsumer
<property>	This element is used to configure the log consumer and only applies if <class> is set to StdoutErrConsumer or FileConsumer.	*PropertyType*

PropertyType

Name	Description	Values	Default
<name>	Name of the property to be configured.	`filename`	
		`append`	
		`stderr_threshold`	
<value>	The value of the property.		
	• If <name> is set to `filename`, then this element contains the name of the log file. This property only applies if <class> is set to `FileConsumer`	`string`	*output.log*
	• If <name> is set to `append`, then this element defines whether the consumer should, upon creation, open the file for appending or overriding. This property only applies if <class> is set to `FileConsumer`	`Boolean`	`false`
	• If <name> is set to `stderr_threshold`, then this element defines the threshold used by the *Log consumers*. This property only applies if <class> is set to `StdoutErrConsumer`	`Log::Kind`	`Log::Kind::Warning`

6.25.9 Dynamic Types profiles

Fast DDS supports the implementation of *DynamicType* by defining them through XML files. Thus the *Dynamic Types* can be modified without the need to modify the source code of the DDS application.

XML Structure

The definition of type profiles in the XML file is done with the `<types>` tag. Each `<types>` element can contain one or more *Type definitions*. Defining several types within a `<types>` element or a single type for each `<types>` element has the same result. Below, an example of a stand-alone types definition via XML is shown.

```
<types xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <type>
        <!-- Type definition -->
    </type>
    <type>
        <!-- Type definition -->
        <!-- Type definition -->
    </type>
</types>
```

Note: For more information on the difference between stand-alone and rooted definitions please refer to section *Rooted vs Standalone profiles definition*.

Type definition

Below, the types supported by *Fast DDS* are presented . For further information about the supported *DynamicType*, please, refer to *Supported Types*. For each of the types detailed below, an example of how to build the type's XML profile is provided.

- *Member types*
  - *Primitive types*
  - *Bounded strings*
  - *Sequences*
  - *Arrays*
  - *Maps*
- *Enumerations*
- *Typedef*
- *Struct*
- *Union*
- *Bitset*
- *Bitmask*
- *Complex types*

Member types

Member types are defined as any type that can belong to a *Struct* or a *Union*, or be aliased by a *Typedef*. These can be defined by the <member> XML tag.

Primitive types

The identifiers of the available basic types are listed in the table below. Please, refer to *Primitive Types* for more information on the primitive types.

boolean	char8	char16
byte	octet	uint8
int8	int16	int32
uint16	uint32	int64
uint64	float32	float64
float128	string	wstring

All of them are defined as follows:

```
<struct name="primitive_types_example">
    <!-- Primitive type definitions inside a struct -->
    <member name="my_bool" type="boolean"/>
    <member name="my_char" type="char8"/>
    <member name="my_wchar" type="char16"/>
    <member name="my_byte" type="byte"/>
    <member name="my_octet" type="octet"/>
    <member name="my_uint8" type="uint8"/>
    <member name="my_short" type="int16"/>
    <member name="my_long" type="int32"/>
    <member name="my_unsignedshort" type="uint16"/>
    <member name="my_unsignedlong" type="uint32"/>
    <member name="my_longlong" type="int64"/>
    <member name="my_unsignedlonglong" type="uint64"/>
    <member name="my_float" type="float32"/>
    <member name="my_double" type="float64"/>
    <member name="my_longdouble" type="float128"/>
    <member name="my_string" type="string"/>
    <member name="my_wstring" type="wstring"/>
</struct>
```

Bounded strings

Bounded strings are defined as any other `string` or `wstring` but adding the attribute `stringMaxLength` with the maximum length available for that specific string. Please, refer to *String and WString* for more information on string type.

```
<member name="my_large_string" type="string" stringMaxLength="41925"/>
<member name="my_large_wstring" type="wstring" stringMaxLength="20925"/>
```

Sequences

The sequence type is implemented by setting three attributes: `name`, `type`, and the `sequenceMaxLength`. The type of its content should be defined by the `type` attribute. Please, refer to *Sequence* section for more information on sequence type.

```
<member name="short_sequence" sequenceMaxLength="5" type="int16"/>
```

Arrays

Arrays are defined in the same way as any other member type but they add the attribute `arrayDimensions`. The format of the `arrayDimensions` attribute value is the size of each dimension separated by commas. Please, refer to *Array* explanation for more information on array type.

```
<member name="long_array" type="int32" arrayDimensions="2,3,4"/>
```

Maps

Maps are similar to sequences, but they need to define two content types. The `key_type` defines the type of the map key, while the `type` defines the map value type. See section *Map* for more information on map type.

```
<member name="long_long_map" type="int32" key_type="int32" mapMaxLength="2"/>
```

Complex types

The complex types are a combination of the aforementioned types. Complex types can be defined using the `<member>` element in the same way a basic or an array type would be. The `type` in this case is `nonBasic` (not a *Primitive types*) and the name of the previously defined type is given in the `nonBasicTypeName` attribute. Please, refer to *Complex Types* section for more information on complex types.

The following example shows a new structure with the `primitive_types_example` struct defined in *Primitive types* example.

```
<struct name="ComplexTypeStruct">
    <member name="my_complex_struct" type="nonBasic" nonBasicTypeName="primitive_types_
↪example" arrayDimensions="5"/>
</struct>
```

Complex types attributes

The attributes of a complex type element can be highly varied depending on the type being defined. Since the attributes that can be defined for each of the types have already been listed, these attributes are then defined in the following table.

Name	Description
type	Data type. This can be a *Primitive types* or a nonBasic type. The latter is used to denote that a complex type is defined.
nonBasicTypeName	Name of the complex type. Only applies if the type attribute is set to nonBasic.
stringMaxLength	Maximum length of a string.
sequenceMaxLength	Maximum length of a *Sequences*.
arrayDimensions	Dimensions of an array.
key_type	Data type of a map key.
mapMaxLength	Maximum length of a *Maps*.

Typedef

The <typedef> XML element is defined by a name and a type mandatory attributes, and any of the optional attributes presented in *Complex types attributes* section. This element allows for defining complex types without the need to define them previously as members. Maps, arrays and sequences can be elements within another container using <typedef>. The <typedef> element corresponds to *Alias* in *Supported Types* section.

```
<typedef name="MyAliasEnum" type="nonBasic" nonBasicTypeName="MyEnum"/>
<typedef name="MyAliasArray" type="int32" arrayDimensions="2,2"/>
<!--Complex map-->
<typedef name="my_map_inner" type="int32" key_type="int32"/>
<struct name="StructComplexMap">
    <member name="my_complex_map" type="nonBasic" nonBasicTypeName="my_map_inner" key_
↪type="int32" mapMaxLength="2"/>
</struct>
<!--Complex sequence-->
<typedef name="long_sequence" sequenceMaxLength="10" type="int32"/>
<struct name="StructComplexSequence">
    <member name="my_sequence_sequence" type="nonBasic" nonBasicTypeName="long_sequence"␣
↪sequenceMaxLength="3"/>
</struct>
```

Enumerations

The <enum> type is defined by its attribute name and a set of <enumerator> child elements. Each <enumerator> is defined by two attributes: a mandatory name and an optional unsigned integer value. Please, refer to *Enumeration* for more information on the <enum> type.

```
<enum name="MyEnum">
    <enumerator name="A" value="0"/>
    <enumerator name="B" value="1"/>
    <enumerator name="C"/>
</enum>
```

Struct

The <struct> element is defined by its name attribute and its <member> child elements. Please, refer to *Structure* for more information on the <struct> type.

```
<struct name="MyStruct">
    <member name="first" type="int32"/>
    <member name="second" type="int64"/>
</struct>
```

Structs can inherit from another struct. This is implemented by defining the value of the baseType attribute, on the child <struct> element to be the value of the name attribute of the parent <struct> element. This is exemplified by the code snippet below.

```
<struct name="ParentStruct">
    <member name="first" type="int32"/>
    <member name="second" type="int64"/>
</struct>
<struct name="ChildStruct" baseType="ParentStruct">
    <member name="third" type="int32"/>
    <member name="fourth" type="int64"/>
</struct>
```

Union

The <union> type is defined by a name attribute, a <discriminator> child element and a set of <case> child elements. The <discriminator> must define its type Each <case> element has one or more <caseDiscriminator> elements, which type must be consistent with the <discriminator> type, and a unique <member> element. Please, refer to *Union* for more information on the <union> type.

```
<union name="MyUnion">
    <discriminator type="byte"/>
    <case>
        <caseDiscriminator value="0"/>
        <caseDiscriminator value="1"/>
        <member name="first" type="int32"/>
    </case>
    <case>
        <caseDiscriminator value="2"/>
        <member name="second" type="nonBasic" nonBasicTypeName="MyStruct"/>
    </case>
    <case>
        <caseDiscriminator value="default"/>
        <member name="third" type="nonBasic" nonBasicTypeName="int64"/>
    </case>
</union>
```

Bitset

The <bitset> element defines the *Bitset* type. It is comprised by a name attribute and a set of <bitfield> child elements. In turn, the <bitfield> element has the mandatory bit_bound attribute, which cannot be higher than 64, and two optional attributes: name and type. A <bitfield> without name attribute is an inaccessible set of bits. Its management type can ease the <bitfield> modification and access. Please, refer to *Bitset* for more information about the <bitset> type.

```
<bitset name="MyBitSet">
    <bitfield name="a" bit_bound="3"/>
    <bitfield name="b" bit_bound="1"/>
    <bitfield bit_bound="4"/>
    <bitfield name="c" bit_bound="10"/>
    <bitfield name="d" bit_bound="12" type="int16"/>
</bitset>
```

Moreover, bitsets can inherit from another bitsets:

```
<bitset name="ParentBitSet">
    <bitfield name="a" bit_bound="10"/>
    <bitfield name="b" bit_bound="15"/>
</bitset>

<bitset name="ChildBitSet" baseType="ParentBitSet">
    <bitfield bit_bound="1"/>
    <bitfield bit_bound="5" type="uint16"/>
</bitset>
```

Bitmask

The <bitmask> element, which corresponds to the *Bitmask* type, is defined by a mandatory name attribute, an optional bit_bound attribute, and several <bit_value> child elements. The bit_bound attribute specifies the number of bits that the bitmask type will manage. The maximum value allowed for the bit_bound is 64. The <bit_value> element must define the name attribute and it might define its position in the bitmask setting the positition attribute. Please, refer to *Bitmask* for more information on the <bitmask> type.

```
<bitmask name="MyBitMask" bit_bound="8">
    <bit_value name="flag0" position="0"/>
    <bit_value name="flag1"/>
    <bit_value name="flag2" position="2"/>
    <bit_value name="flag5" position="5"/>
</bitmask>
```

Loading dynamic types in a *Fast DDS* application

In the *Fast DDS* application that will make use of the *XML Types*, the XML files that define the types must be loaded before trying to instantiate *DynamicPubSubType* objects of these types.

```
// Create a DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Load the XML File
if (ReturnCode_t::RETCODE_OK ==
        DomainParticipantFactory::get_instance()->load_XML_profiles_file("my_profiles.xml
↪"))
{
    // Retrieve the an instance of MyStruct type
    eprosima::fastrtps::types::DynamicType_ptr my_struct_type =
            eprosima::fastrtps::xmlparser::XMLProfileManager::getDynamicTypeByName(
↪"MyStruct")->build();
    // Register MyStruct type
    TypeSupport my_struct_type_support(new␣
↪eprosima::fastrtps::types::DynamicPubSubType(my_struct_type));
    my_struct_type_support.register_type(participant, nullptr);
}
else
{
    std::cout << "Cannot open XML file \"types.xml\". "
              << "Please, set the correct path to the XML file"
              << std::endl;
}
```

6.25.10 Common

The preceding XML profiles define some XML elements that are common to several profiles. This section aims to explain these common elements.

- *LocatorListType*
- *ExternalLocatorListType*
- *PropertiesPolicyType*
- *DurationType*
- *TopicType*
  - *HistoryQoS*
  - *ResourceLimitsQos*
- *QoS*

  - *Data-Sharing*
  - *Deadline*
  - *DisableHeartbeatPiggyback*
  - *DisablePositiveAcks*
  - *Durability*
  - *GroupData*
  - *LatencyBudget*
  - *Lifespan*
  - *Liveliness*
  - *Ownership*
  - *Ownership Strength*
  - *Partition*
  - *PublishMode*
  - *ReliabilityQosPolicy*
  - *TopicData*
  - *UserData*
- *HistoryMemoryPolicy*
- *Allocation Configuration*

LocatorListType

It represents a list of `Locator_t`. LocatorListType is used inside other configuration parameter labels that expect a list of locators, for example, in <defaultUnicastLocatorList>. Therefore, LocatorListType is defined as a set of <locator> elements. The <locator> element has a single child element that defines the transport protocol for which the locator is defined. These are: <udpv4>, <tcpv4>, <udpv6>, and <tcpv6>. The table presented below outlines each possible Locator's field.

Note: *SHM transport* locators cannot be configured as they are automatically handled by SHM.

Name	Description	Values	Default
<port>	RTPS port number of the locator. *Physical port* in UDP, *logical port* in TCP.	`uint16_t`	0
<physical_port>	TCP's *physical port*.	`uint16_t`	0
<address>	IP address of the locator.	`string` (IPv4/IPv6 format or DNS name)	Empty
<unique_lan_id>	The LAN ID uniquely identifies the LAN the locator belongs to (**TCPv4 only**).	`string` (16 bytes)	Empty
<wan_address>	WAN IPv4 address (**TCPv4 only**).	`string` (IPv4 format)	`0.0.0.0`

Example

The following example shows the implementation of one locator of each transport protocol in <defaultUnicastLocatorList>.

```
<defaultUnicastLocatorList>
    <locator>
        <udpv4>
            <!-- Access as physical, typical UDP usage -->
            <port>7400</port>
            <address>192.168.1.41</address>
        </udpv4>
        <udpv4>
            <!-- Access as physical, typical UDP usage -->
            <port>7600</port>
            <address>localhost</address>
        </udpv4>
    </locator>
    <locator>
        <tcpv4>
            <!-- Both physical and logical (port), useful in TCP transports -->
            <physical_port>5100</physical_port>
            <port>7400</port>
            <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
            <wan_address>80.80.99.45</wan_address>
            <address>192.168.1.55</address>
        </tcpv4>
    </locator>
    <locator>
        <udpv6>
            <port>8844</port>
            <address>::1</address>
        </udpv6>
        <udpv6>
            <port>8888</port>
            <address>localhost</address>
        </udpv6>
    </locator>
    <locator>
        <tcpv6>
            <!-- Both physical and logical (port), useful in TCP transports -->
            <physical_port>5100</physical_port>
            <port>7400</port>
            <address>fe80::55e3:290:165:5af8</address>
        </tcpv6>
    </locator>
</defaultUnicastLocatorList>
```

ExternalLocatorListType

It represents a list of external locator entries. Each entry can be a <udpv4> or a <udpv6> tag. These tags can be configured with the following attributes:

Name	Description	Values	Default
externality	Number of hops from the participant's host to the LAN represented by the external locator. Valid values: from 1 to 255.	uint8_t	1
cost	Communication cost relative to other locators on the same externality level. Valid values: from 0 to 255.	uint8_t	0
mask	Number of significant bits on the LAN represented by the external locator. Valid values: from 1 to 31 (UDPv4) or 127 (UDPv6)	uint8_t	24

They should contain the following tags:

Name	Description	Values
<port>	UDP port number of the locator. The UDP port number should be valid.	uint16_t
<address>	IP address of the locator.	string (IPv4/IPv6 format or DNS name)

Example

The following example shows the implementation of one locator of each transport protocol in <default_external_unicast_locators>.

```
<default_external_unicast_locators>
    <udpv4 externality="1" cost="0" mask="24">
        <address>100.100.100.10</address>
        <port>23456</port>
    </udpv4>
    <udpv6 externality="2" cost="0" mask="48">
        <address>::1</address>
        <port>1234</port>
    </udpv6>
</default_external_unicast_locators>
```

PropertiesPolicyType

PropertiesPolicyType defines the <propertiesPolicy> element. It allows the user to define a set of generic properties inside a <properties> element. It is useful at defining extended or custom configuration parameters.

Name	Description	Values	Default
<name>	Name to identify the property.	string	
<value>	Property's value.	string	
<propagate>	Indicates if it is going to be serialized along with the object it belongs to.	bool	false

Example

```
<propertiesPolicy>
    <properties>
        <property>
            <name>Property1Name</name>
            <value>Property1Value</value>
            <propagate>false</propagate>
        </property>
        <property>
            <name>Property2Name</name>
            <value>Property2Value</value>
            <propagate>true</propagate>
        </property>
    </properties>
</propertiesPolicy>
```

DurationType

DurationType expresses a period of time and it is commonly used inside other XML elements, such as in <leaseAnnouncement> or <leaseDuration>. A DurationType is defined by at least one mandatory element of two possible ones: <sec> plus <nanosec>. An infinite value can be specified by using the values DURATION_INFINITY, DURATION_INFINITE_SEC and DURATION_INFINITE_NSEC.

Name	Description	Values	Default
<sec>	Number of seconds.	int32_t	0
<nanosec>	Number of nanoseconds.	uint32_t	0

Example

```
<discovery_config>
    <leaseDuration>
        <sec>DURATION_INFINITY</sec>
    </leaseDuration>
    <leaseAnnouncement>
        <sec>1</sec>
        <nanosec>856000</nanosec>
    </leaseAnnouncement>
</discovery_config>
```

TopicType

This XML element allows the configuration of the specific *HistoryQosPolicy* and *ResourceLimitsQosPolicy* QoS of the Datawriters and DataReaders in which this element is defined inside of. Also, it sets the *TopicQos* configuration with the policies detailed.

Name	Description	Values
<historyQos>	It controls the behavior of *Fast DDS* when the value of an instance changes before it is finally communicated to some of its existing DataReaders.	*HistoryQoS*
<resourceLimitsQos>	It controls the resources that *Fast DDS* can use in order to meet the requirements imposed by the application and other QoS settings.	*ResourceLimitsQos*

Example

```
<data_writer profile_name="dataWriter_topic_example">
    <topic>
        <historyQos>
            <kind>KEEP_LAST</kind>
            <depth>20</depth>
        </historyQos>
        <resourceLimitsQos>
            <max_samples>5</max_samples>
            <max_instances>2</max_instances>
            <max_samples_per_instance>1</max_samples_per_instance>
            <allocated_samples>20</allocated_samples>
            <extra_samples>10</extra_samples>
        </resourceLimitsQos>
    </topic>
</data_writer>
```

HistoryQoS

It controls the behavior of *Fast DDS* when the value of an instance changes before it is finally communicated to some of its existing DataReaders. Please refer to *HistoryQosPolicyKind* for further information on HistoryQoS.

Name	Description	Values	Default
<kind>	*Fast DDS* will only attempt to keep the latest values of the instance and discard the older ones.	*KEEP_LAST*	*KEEP_LAST*
	Fast DDS will attempt to maintain and deliver all the values of the instance to existing DataReaders.	*KEEP_ALL*	
<depth>	It must be consistent with the *ResourceLimitsQos* <max_samples_per_instance> element value. It must be verified that: <depth> <= <max_samples_per_instance>.	uint32_t	

ResourceLimitsQos

It controls the resources that *Fast DDS* can use in order to meet the requirements imposed by the application and other QoS settings. Please refer to *ResourceLimitsQosPolicy* for further information on ResourceLimitsQos.

Name	Description	Values	Default
<max_samples>	It must verify that: <max_samples> >= <max_samples_per_instance>.	uint32_t	5000
<max_instances>	It defines the maximum number of instances.	uint32_t	10
<max_samples_per_instance>	It must verify that: *HistoryQos* <depth> <= <max_samples_per_instance>.	uint32_t	400
<allocated_samples>	It controls the maximum number of samples to be stored.	uint32_t	100
<extra_samples>	The number of extra samples to allocate on the pool.	uint32_t	1

QoS

The Quality of Service (QoS) is used to specify the behavior of the Service, allowing the user to define how each *Entity* will behave. Please refer to the *Policy* section for more information on QoS.

Name	Description	Values
<data_sharing>	See *DataSharingQosPolicy*	*Data-Sharing*
<deadline>	See *DeadlineQosPolicy*.	*Deadline*
<disable_heartbeat_piggyback>	See *DisableHeartbeatPiggyback*.	*DisableHeartbeatPiggyback*
<disablePositiveAcks>	See *DisablePositiveACKsQosPolicy*.	*DisablePositiveAcks*
<durability>	See *DurabilityQosPolicy*.	*Durability*
<groupData>	See *GroupDataQosPolicy*.	*GroupData*
<latencyBudget>	See *LatencyBudgetQosPolicy*.	*LatencyBudget*
<lifespan>	See *LifespanQosPolicy*.	*Lifespan*
<liveliness>	See *LivelinessQosPolicy*.	*Liveliness*
<ownership>	See *OwnershipQosPolicy*.	*Ownership*
<ownershipStrength>	See *OwnershipStrengthQosPolicy*.	*Ownership Strength*
<partition>	See *PartitionQosPolicy*.	*Partition*
<publishMode>	See *PublishModeQosPolicy*.	*PublishMode*
<reliability>	See *ReliabilityQosPolicy*.	*ReliabilityQosPolicy*
<topicData>	See *TopicDataQosPolicy*.	*TopicData*
<userData>	See *UserDataQosPolicy*.	*UserData*

Example

```
<data_writer profile_name="pub_topic_qos">
<qos> <!-- writerQosPoliciesType -->
    <data_sharing>
        <kind>AUTOMATIC</kind>
        <shared_dir>/home</shared_dir>
        <max_domains>10</max_domains>
        <domain_ids>
            <domainId>0</domainId>
            <domainId>11</domainId>
        </domain_ids>
    </data_sharing>

    <deadline>
        <period>
            <sec>1</sec>
        </period>
    </deadline>

    <!-- DataWriter specific QoS -->
    <disable_heartbeat_piggyback>true</disable_heartbeat_piggyback>

    <disablePositiveAcks>
        <enabled>true</enabled>
        <duration>
            <sec>1</sec>
        </duration>
    </disablePositiveAcks>
```

(continues on next page)

(continued from previous page)

```
    <durability>
        <kind>VOLATILE</kind>
    </durability>

    <groupData>
        <value>1.a.2</value>
    </groupData>

    <!-- QoS policy pending implementation -->
    <latencyBudget>
        <duration>
            <sec>1</sec>
        </duration>
    </latencyBudget>

    <lifespan>
        <duration>
            <sec>1</sec>
        </duration>
    </lifespan>

    <liveliness>
        <kind>AUTOMATIC</kind>

        <lease_duration>
            <sec>1</sec>
        </lease_duration>

        <announcement_period>
            <sec>1</sec>
        </announcement_period>
    </liveliness>

    <ownership>
        <kind>EXCLUSIVE</kind>
    </ownership>

    <!-- DataWriter specific QoS -->
    <ownershipStrength>
        <value>50</value>
    </ownershipStrength>

    <partition>
        <names>
            <name>part1</name>
            <name>part2</name>
        </names>
    </partition>

    <!-- DataWriter specific QoS -->
    <publishMode>
```

(continues on next page)

(continued from previous page)

```
        <kind>ASYNCHRONOUS</kind>
    </publishMode>

    <reliability>
        <kind>BEST_EFFORT</kind>
        <max_blocking_time>
            <sec>1</sec>
        </max_blocking_time>
    </reliability>

    <topicData>
        <value>2.b.1</value>
    </topicData>

    <userData>
        <value>3.c.0</value>
    </userData>
</qos>
```

Data-Sharing

Name	Description	Values	Default
<kind>	See *DataSharingKind*	*AUTOMATIC ON OFF*	*AUTOMATIC*
<shared_dir>	Directory used for the memory-mapped files.	string	Empty
<max_domains>	Maximum number of Data-Sharing domain IDs in the local or remote endpoints.	uint32_t	0 (unlimited)
<domain_ids>	List of Data-Sharing domain IDs configured for the current endpoint.	<domainId>	Empty list

Name	Description	Values
<domainId>	Domain ID to be used by the endpoint for Data-Sharing.	uint32_t

Deadline

Name	Description	Values	Default
<period>	See *DeadlineQosPolicy*.	*DurationType*	*c_TimeInfinite*

DisableHeartbeatPiggyback

Name	Description	Values	Default
<disable_heartbeat_piggyback>	See *DisableHeartbeatPiggyback*.	bool	false

Important: This configuration is only available for *DataWriter QoS profile configuration*.

DisablePositiveAcks

Name	Description	Values	Default
<enabled>	See *DisablePositiveACKsQosPolicy*.	bool	false
<duration>	See *DisablePositiveACKsQosPolicy*.	*DurationType*	*c_TimeInfinite*

Durability

Name	Description	Values	Default
<kind>	See *DurabilityQosPolicyKind*.	*VOLATILE* *TRANSIENT_LOCAL* *TRANSIENT* *PERSISTENT*	DataReaders: *VOLATILE* DataWriters: *TRANSIENT_LOCAL*

GroupData

Name	Description	Values	Default
<value>	See *GroupDataQosPolicy*.	string (std::vector<octet>)	Empty

LatencyBudget

Name	Description	Values	Default
<duration>	See *LatencyBudgetQosPolicy*.	*DurationType*	0

Lifespan

Name	Description	Values	Default
<duration>	See *LifespanQosPolicy*.	*DurationType*	*c_TimeInfinite*

Liveliness

Name	Description	Values	Default
<kind>	See *LivelinessQosPolicyKind*.	*AUTOMATIC*	*AUTOMATIC*
		MANUAL_BY_PARTICIPANT	
		MANUAL_BY_TOPIC	
<lease_duration>	See *LivelinessQosPolicy*.	*DurationType*	*c_TimeInfinite*
<announcement_period>	See *LivelinessQosPolicy*.	*DurationType*	*c_TimeInfinite*

Ownership

Name	Description	Values	Default
<kind>	See *OwnershipQosPolicyKind*.	*SHARED EXCLUSIVE*	*SHARED*

Ownership Strength

Name	Description	Values	Default
<value>	See *OwnershipStrengthQosPolicy*.	uint32_t	0

Important: This configuration is only available for *DataWriter QoS profile configuration*.

Partition

Name	Description	Values
<names>	It comprises a set of <name> elements containing the name of each partition. See *PartitionQosPolicy*.	<name>

PublishMode

Name	Description	Values	Default
<kind>	See *PublishModeQosPolicy*.	ASYNCHRONOUS	ASYNCHRONOUS
		SYNCHRONOUS	

Important: This configuration is only available for *DataWriter QoS profile configuration*.

ReliabilityQosPolicy

Name	Description	Values	Default
<kind>	See *ReliabilityQosPolicyKind*.	*BEST_EFFORT*	DataReaders: *BEST_EFFORT*
		RELIABLE	DataWriters: *RELIABLE*
<max_blocking_time>	See *ReliabilityQosPolicy*.	*DurationType*	100 ms

TopicData

Name	Description	Values	Default
<value>	See *TopicDataQosPolicy*.	string (std::vector<octet>)	Empty

UserData

Name	Description	Values	Default
<value>	See *UserDataQosPolicy*.	string (std::vector<octet>)	Empty

HistoryMemoryPolicy

Indicates the way the memory is managed in terms of dealing with the CacheChanges of the *RTPSEndpointQos*.

Name	Description	Values	Default
<historyMemoryPolicy>	Four different options as described in *MemoryManagementPolicy*.	*PREALLOCATED* *PREALLOCATED_WITH_REALLOC* *DYNAMIC* *DYNAMIC_REUSABLE*	*PREALLOCATED*

Example

```
<data_writer profile_name="data_writer_historyMemoryPolicy">
    <!-- ...  -->
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</data_writer>

<data_reader profile_name="data_reader_historyMemoryPolicy">
```

(continues on next page)

(continued from previous page)

```
    <!-- ...  -->
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</data_reader>
```

Allocation Configuration

The <allocation> element allows to control the allocation behavior of internal collections for which the number of elements depends on the number of entities in the system. For instance, there are collections inside a DataWriter which depend on the number of DataReaders matching with it. Please refer to *ParticipantResourceLimitsQos* for a detailed documentation on DomainParticipant allocation, and to *Tuning allocations* for detailed information on how to tune allocation related parameters.

Name	Description	Values	Default
<initial>	Number of elements for which space is initially allocated.	uint32_t	0
<maximum>	Maximum number of elements for which space will be allocated.	uint32_t	0 (Means no limit)
<increment>	Number of new elements that will be allocated when more space is necessary.	uint32_t	1

6.25.11 Example

In this section, there is a full XML example with all possible configuration.

Warning: This example can be used as a quick reference, but it may not be correct due to incompatibility or exclusive properties. **Do not take it as a working example.**

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
    <profiles>
        <transport_descriptors>
            <!-- TCP sample transport descriptor -->
            <transport_descriptor>
                <transport_id>ExampleTransportId1</transport_id>
                <type>TCPv4</type>
                <sendBufferSize>8192</sendBufferSize>
                <receiveBufferSize>8192</receiveBufferSize>
                <maxMessageSize>16384</maxMessageSize>
                <maxInitialPeersRange>100</maxInitialPeersRange>
                <interfaceWhiteList>
                    <address>192.168.1.41</address>
                    <address>127.0.0.1</address>
                </interfaceWhiteList>
                <wan_addr>80.80.55.44</wan_addr>
                <keep_alive_frequency_ms>5000</keep_alive_frequency_ms>
                <keep_alive_timeout_ms>25000</keep_alive_timeout_ms>
                <max_logical_port>200</max_logical_port>
                <logical_port_range>20</logical_port_range>
                <logical_port_increment>2</logical_port_increment>
```

(continues on next page)

(continued from previous page)

```
                <listening_ports>
                    <port>5100</port>
                    <port>5200</port>
                </listening_ports>
                <tls>
                    <password>Password</password>
                    <private_key_file>Key_file.pem</private_key_file>
                    <rsa_private_key_file>RSA_file.pem</rsa_private_key_file>
                    <cert_chain_file>Chain.pem</cert_chain_file>
                    <tmp_dh_file>DH.pem</tmp_dh_file>
                    <verify_file>verify.pem</verify_file>
                    <verify_mode>
                        <verify>VERIFY_PEER</verify>
                    </verify_mode>
                    <options>
                        <option>NO_TLSV1</option>
                        <option>NO_TLSV1_1</option>
                    </options>
                    <verify_paths>
                        <verify_path>Path1</verify_path>
                        <verify_path>Path2</verify_path>
                        <verify_path>Path3</verify_path>
                    </verify_paths>
                    <verify_depth>55</verify_depth>
                    <default_verify_path>true</default_verify_path>
                    <handshake_role>SERVER</handshake_role>
                    <server_name>my_server.com</server_name>
                </tls>
                <calculate_crc>false</calculate_crc>
                <check_crc>false</check_crc>
                <enable_tcp_nodelay>false</enable_tcp_nodelay>
            </transport_descriptor>
            <!-- UDP sample transport descriptor. Several options are common with TCP -->
            <transport_descriptor>
                <transport_id>ExampleTransportId2</transport_id>
                <type>UDPv6</type>
                <TTL>250</TTL>
                <non_blocking_send>false</non_blocking_send>
                <output_port>5101</output_port>
            </transport_descriptor>
            <!-- SHM sample transport descriptor -->
            <transport_descriptor>
                <transport_id>SHM_SAMPLE_DESCRIPTOR</transport_id>
                    <type>SHM</type> <!-- REQUIRED -->
                    <maxMessageSize>524288</maxMessageSize> <!-- OPTIONAL uint32 valid␣
↪of all transports-->
                    <segment_size>1048576</segment_size> <!-- OPTIONAL uint32 SHM only-->
                    <port_queue_capacity>1024</port_queue_capacity> <!-- OPTIONAL uint32␣
↪SHM only-->
                    <healthy_check_timeout_ms>250</healthy_check_timeout_ms> <!--␣
↪OPTIONAL uint32 SHM only-->
                    <rtps_dump_file>test_file.dump</rtps_dump_file> <!-- OPTIONAL string␣
↪SHM only-->
```

(continues on next page)

(continued from previous page)

```
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="participant_profile_example">
        <domainId>4</domainId>
        <rtps>
            <name>Participant Name</name> <!-- String -->

            <defaultUnicastLocatorList>
                <locator>
                    <udpv4>
                        <!-- Access as physical, like UDP -->
                        <port>7400</port>
                        <address>localhost</address>
                    </udpv4>
                </locator>
                <locator>
                    <tcpv4>
                        <!-- Both physical and logical (port), like TCP -->
                        <physical_port>5100</physical_port>
                        <port>7400</port>
                        <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.55</address>
                    </tcpv4>
                </locator>
                <locator>
                    <udpv6>
                        <port>8844</port>
                        <address>::1</address>
                    </udpv6>
                </locator>
            </defaultUnicastLocatorList>

            <defaultMulticastLocatorList>
                <locator>
                    <udpv4>
                        <!-- Access as physical, like UDP -->
                        <port>7400</port>
                        <address>192.168.1.41</address>
                    </udpv4>
                </locator>
                <locator>
                    <tcpv4>
                        <!-- Both physical and logical (port), like TCP -->
                        <physical_port>5100</physical_port>
                        <port>7400</port>
                        <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.55</address>
                    </tcpv4>
                </locator>
```

(continues on next page)

(continued from previous page)

```
                    <locator>
                        <udpv6>
                            <port>8844</port>
                            <address>::1</address>
                        </udpv6>
                    </locator>
                </defaultMulticastLocatorList>

                <default_external_unicast_locators>
                    <!-- EXTERNAL_LOCATOR_LIST -->
                    <udpv4 externality="1" cost="0" mask="24">
                        <address>100.100.100.10</address>
                        <port>23456</port>
                    </udpv4>
                    <udpv6 externality="1" cost="1" mask="48">
                        <address>::1</address>
                        <port>1234</port>
                    </udpv6>
                </default_external_unicast_locators>

                <ignore_non_matching_locators>true</ignore_non_matching_locators>
                <sendSocketBufferSize>8192</sendSocketBufferSize>
                <listenSocketBufferSize>8192</listenSocketBufferSize>

                <builtin>
                    <discovery_config>
                        <discoveryProtocol>NONE</discoveryProtocol>
                        <discoveryServersList>
                            <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.74">
                                <metatrafficUnicastLocatorList>
                                    <locator>
                                        <udpv4>
                                            <address>192.168.10.57</address>
                                            <port>56542</port>
                                        </udpv4>
                                    </locator>
                                </metatrafficUnicastLocatorList>
                                <metatrafficMulticastLocatorList>
                                    <locator>
                                        <udpv4>
                                            <address>192.168.10.58</address>
                                            <port>24565</port>
                                        </udpv4>
                                    </locator>
                                </metatrafficMulticastLocatorList>
                            </RemoteServer>
                            <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.75">
                                <metatrafficUnicastLocatorList>
                                    <locator>
                                        <udpv4>
                                            <address>192.168.10.59</address>
                                            <port>56543</port>
```

(continues on next page)

(continued from previous page)

```
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                            <metatrafficMulticastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.60</address>
                                        <port>34565</port>
                                    </udpv4>
                                </locator>
                            </metatrafficMulticastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                    <ignoreParticipantFlags>FILTER_DIFFERENT_PROCESS|FILTER_SAME_
↪PROCESS</ignoreParticipantFlags>
                    <EDP>SIMPLE</EDP>
                    <simpleEDP>
                        <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
                        <PUBREADER_SUBWRITER>true</PUBREADER_SUBWRITER>
                    </simpleEDP>
                    <leaseDuration>
                        <sec>DURATION_INFINITY</sec>
                    </leaseDuration>
                    <leaseAnnouncement>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </leaseAnnouncement>
                    <initialAnnouncements>
                        <count>10</count>
                        <period>
                            <nanosec>50</nanosec>
                        </period>
                    </initialAnnouncements>
                    <clientAnnouncementPeriod>
                        <nanosec>250000000</nanosec>
                    </clientAnnouncementPeriod>
                    <static_edp_xml_config>filename1.xml</static_edp_xml_config>
                    <static_edp_xml_config>filename2.xml</static_edp_xml_config>
                    <static_edp_xml_config>filename3.xml</static_edp_xml_config>
                </discovery_config>

                <avoid_builtin_multicast>true</avoid_builtin_multicast>
                <use_WriterLivelinessProtocol>false</use_WriterLivelinessProtocol>

                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <!-- Access as physical, like UDP -->
                            <port>7400</port>
                            <address>192.168.1.41</address>
                        </udpv4>
                    </locator>
```

(continues on next page)

(continued from previous page)

```
                <locator>
                    <tcpv4>
                        <!-- Both physical and logical (port), like TCP -->
                        <physical_port>5100</physical_port>
                        <port>7400</port>
                        <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.55</address>
                    </tcpv4>
                </locator>
                <locator>
                    <udpv6>
                        <port>8844</port>
                        <address>::1</address>
                    </udpv6>
                </locator>
            </metatrafficUnicastLocatorList>

            <metatrafficMulticastLocatorList>
                <locator>
                    <udpv4>
                        <!-- Access as physical, like UDP -->
                        <port>7400</port>
                        <address>192.168.1.41</address>
                    </udpv4>
                </locator>
                <locator>
                    <tcpv4>
                        <!-- Both physical and logical (port), like TCP -->
                        <physical_port>5100</physical_port>
                        <port>7400</port>
                        <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.55</address>
                    </tcpv4>
                </locator>
                <locator>
                    <udpv6>
                        <port>8844</port>
                        <address>::1</address>
                    </udpv6>
                </locator>
            </metatrafficMulticastLocatorList>

            <initialPeersList>
                <locator>
                    <udpv4>
                        <!-- Access as physical, like UDP -->
                        <port>7400</port>
                        <address>192.168.1.41</address>
                    </udpv4>
                </locator>
```

(continues on next page)

(continued from previous page)

```
                        <locator>
                            <tcpv4>
                                <!-- Both physical and logical (port), like TCP -->
                                <physical_port>5100</physical_port>
                                <port>7400</port>
                                <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                                <wan_address>80.80.99.45</wan_address>
                                <address>192.168.1.55</address>
                            </tcpv4>
                        </locator>
                        <locator>
                            <udpv6>
                                <port>8844</port>
                                <address>::1</address>
                            </udpv6>
                        </locator>
                    </initialPeersList>

                    <metatraffic_external_unicast_locators>
                        <udpv4 externality="1" cost="0" mask="24">
                            <address>100.100.100.10</address>
                            <port>34567</port>
                        </udpv4>
                    </metatraffic_external_unicast_locators>

                    <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</
↪readerHistoryMemoryPolicy>
                    <writerHistoryMemoryPolicy>PREALLOCATED</writerHistoryMemoryPolicy>
                    <readerPayloadSize>512</readerPayloadSize>
                    <writerPayloadSize>512</writerPayloadSize>
                    <mutation_tries>55</mutation_tries>
                    <typelookup_config>
                        <use_client>true</use_client>
                        <use_server>true</use_server>
                    </typelookup_config>
                </builtin>

                <port>
                    <portBase>7400</portBase>
                    <domainIDGain>200</domainIDGain>
                    <participantIDGain>10</participantIDGain>
                    <offsetd0>0</offsetd0>
                    <offsetd1>1</offsetd1>
                    <offsetd2>2</offsetd2>
                    <offsetd3>3</offsetd3>
                </port>

                <participantID>99</participantID>

                <userTransports>
                    <transport_id>ExampleTransportId1</transport_id>
                    <transport_id>ExampleTransportId2</transport_id>
```

(continues on next page)

(continued from previous page)

```
                </userTransports>

                <useBuiltinTransports>false</useBuiltinTransports>

                <builtinTransports>DEFAULT</builtinTransports>

                <propertiesPolicy>
                    <properties>
                        <property>
                            <name>Property1Name</name>
                            <value>Property1Value</value>
                            <propagate>false</propagate>
                        </property>
                        <property>
                            <name>Property2Name</name>
                            <value>Property2Value</value>
                            <propagate>false</propagate>
                        </property>
                    </properties>
                </propertiesPolicy>

                <allocation>
                    <remote_locators>
                        <max_unicast_locators>4</max_unicast_locators> <!-- uint32 -->
                        <max_multicast_locators>1</max_multicast_locators> <!-- uint32 --
↪>
                    </remote_locators>
                    <total_participants>
                        <initial>0</initial>
                        <maximum>0</maximum>
                        <increment>1</increment>
                    </total_participants>
                    <total_readers>
                        <initial>0</initial>
                        <maximum>0</maximum>
                        <increment>1</increment>
                    </total_readers>
                    <total_writers>
                        <initial>0</initial>
                        <maximum>0</maximum>
                        <increment>1</increment>
                    </total_writers>
                    <max_partitions>256</max_partitions>
                    <max_user_data>256</max_user_data>
                    <max_properties>512</max_properties>
                    <send_buffers>
                        <preallocated_number>127</preallocated_number>
                        <dynamic>true</dynamic>
                    </send_buffers>
                </allocation>
            </rtps>
        </participant>
```

(continues on next page)

(continued from previous page)

```
        <data_writer profile_name="datawriter_profile_example">
            <topic>
                <historyQos>
                    <kind>KEEP_LAST</kind>
                    <depth>20</depth>
                </historyQos>
                <resourceLimitsQos>
                    <max_samples>5</max_samples>
                    <max_instances>2</max_instances>
                    <max_samples_per_instance>1</max_samples_per_instance>
                    <allocated_samples>20</allocated_samples>
                    <extra_samples>10</extra_samples>
                </resourceLimitsQos>
            </topic>
            <qos> <!-- dataWriterQosPoliciesType -->
                <data_sharing>
                    <kind>AUTOMATIC</kind>
                    <shared_dir>/home</shared_dir>
                    <max_domains>10</max_domains>
                    <domain_ids>
                        <domainId>0</domainId>
                        <domainId>11</domainId>
                    </domain_ids>
                </data_sharing>
                <deadline>
                    <period>
                        <sec>1</sec>
                    </period>
                </deadline>
                <disable_heartbeat_piggyback>true</disable_heartbeat_piggyback>
                <disablePositiveAcks>
                    <enabled>true</enabled>
                    <duration>
                        <sec>1</sec>
                    </duration>
                </disablePositiveAcks>
                <durability>
                    <kind>VOLATILE</kind>
                </durability>
                <!-- QoS policy pending implementation -->
                <latencyBudget>
                    <duration>
                        <sec>1</sec>
                    </duration>
                </latencyBudget>
                <lifespan>
                    <duration>
                        <sec>5</sec>
                    </duration>
                </lifespan>
                <liveliness>
```

(continues on next page)

(continued from previous page)

```
                    <kind>AUTOMATIC</kind>
                    <lease_duration>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </lease_duration>
                    <announcement_period>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </announcement_period>
                </liveliness>
                <ownership>
                    <kind>EXCLUSIVE</kind>
                </ownership>
                <ownershipStrength>
                    <value>50</value>
                </ownershipStrength>
                <partition>
                    <names>
                        <name>part1</name>
                        <name>part2</name>
                    </names>
                </partition>
                <publishMode>
                    <kind>ASYNCHRONOUS</kind>
                </publishMode>
                <reliability>
                    <kind>BEST_EFFORT</kind>
                    <max_blocking_time>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </max_blocking_time>
                </reliability>
            </qos>

            <times>
                <initialHeartbeatDelay>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </initialHeartbeatDelay>
                <heartbeatPeriod>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </heartbeatPeriod>
                <nackResponseDelay>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </nackResponseDelay>
                <nackSupressionDuration>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </nackSupressionDuration>
            </times>
```

(continues on next page)

(continued from previous page)

```
        <unicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </unicastLocatorList>

        <multicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </multicastLocatorList>
```

(continues on next page)

(continued from previous page)

```
            <external_unicast_locators>
                <udpv4 externality="1" cost="0" mask="24">
                    <address>100.100.100.10</address>
                    <port>12345</port>
                </udpv4>
            </external_unicast_locators>

            <ignore_non_matching_locators>true</ignore_non_matching_locators>
            <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

            <propertiesPolicy>
                <properties>
                    <property>
                        <name>Property1Name</name>
                        <value>Property1Value</value>
                        <propagate>false</propagate>
                    </property>
                    <property>
                        <name>Property2Name</name>
                        <value>Property2Value</value>
                        <propagate>false</propagate>
                    </property>
                </properties>
            </propertiesPolicy>

            <userDefinedID>45</userDefinedID>
            <entityID>76</entityID>

            <matchedSubscribersAllocation>
                <initial>3</initial>
                <maximum>3</maximum>
                <increment>0</increment>
            </matchedSubscribersAllocation>
        </data_writer>

        <data_reader profile_name="datareader_profile_example">
            <topic>
                <historyQos>
                    <kind>KEEP_LAST</kind>
                    <depth>20</depth>
                </historyQos>
                <resourceLimitsQos>
                    <max_samples>5</max_samples>
                    <max_instances>2</max_instances>
                    <max_samples_per_instance>1</max_samples_per_instance>
                    <allocated_samples>20</allocated_samples>
                    <extra_samples>10</extra_samples>
                </resourceLimitsQos>
            </topic>
            <qos> <!-- dataReaderQosPoliciesType -->
                <data_sharing>
```

(continues on next page)

(continued from previous page)

```
                    <kind>AUTOMATIC</kind>
                    <shared_dir>/home</shared_dir>
                    <max_domains>10</max_domains>
                    <domain_ids>
                        <domainId>0</domainId>
                        <domainId>11</domainId>
                    </domain_ids>
                </data_sharing>
                <deadline>
                    <period>
                        <sec>1</sec>
                    </period>
                </deadline>
                <disablePositiveAcks>
                    <enabled>true</enabled>
                    <duration>
                        <sec>1</sec>
                    </duration>
                </disablePositiveAcks>
                <durability>
                    <kind>PERSISTENT</kind>
                </durability>
                <!-- QoS policy pending implementation -->
                <latencyBudget>
                    <duration>
                        <sec>1</sec>
                    </duration>
                </latencyBudget>
                <lifespan>
                    <duration>
                        <sec>5</sec>
                    </duration>
                </lifespan>
                <liveliness>
                    <kind>MANUAL_BY_PARTICIPANT</kind>
                    <lease_duration>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </lease_duration>
                    <announcement_period>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </announcement_period>
                </liveliness>
                <ownership>
                    <kind>EXCLUSIVE</kind>
                </ownership>
                <partition>
                    <names>
                        <name>part1</name>
                        <name>part2</name>
                    </names>
```

(continues on next page)

(continued from previous page)

```
            </partition>
            <reliability>
                <kind>BEST_EFFORT</kind>
                <max_blocking_time>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </max_blocking_time>
            </reliability>
        </qos>

        <times>
            <initialAcknackDelay>
                <sec>1</sec>
                <nanosec>856000</nanosec>
            </initialAcknackDelay>
            <heartbeatResponseDelay>
                <sec>1</sec>
                <nanosec>856000</nanosec>
            </heartbeatResponseDelay>
        </times>

        <unicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </unicastLocatorList>

        <multicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
```

(continues on next page)

(continued from previous page)

```
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </multicastLocatorList>

        <external_unicast_locators>
            <udpv4 externality="1" cost="0" mask="24">
                <address>100.100.100.10</address>
                <port>12345</port>
            </udpv4>
        </external_unicast_locators>

        <ignore_non_matching_locators>true</ignore_non_matching_locators>
        <expectsInlineQos>true</expectsInlineQos>
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

        <propertiesPolicy>
            <properties>
                <property>
                    <name>Property1Name</name>
                    <value>Property1Value</value>
                    <propagate>false</propagate>
                </property>
                <property>
                    <name>Property2Name</name>
                    <value>Property2Value</value>
                    <propagate>false</propagate>
                </property>
            </properties>
        </propertiesPolicy>

        <userDefinedID>55</userDefinedID>
        <entityID>66</entityID>

        <matchedPublishersAllocation>
            <initial>1</initial>
```

(continues on next page)

(continued from previous page)

```
                <maximum>1</maximum>
                <increment>0</increment>
            </matchedPublishersAllocation>
        </data_reader>

        <topic profile_name="topic_profile_example">
            <historyQos>
                <kind>KEEP_LAST</kind>
                <depth>20</depth>
            </historyQos>
            <resourceLimitsQos>
                <max_samples>5</max_samples>
                <max_instances>2</max_instances>
                <max_samples_per_instance>1</max_samples_per_instance>
                <allocated_samples>20</allocated_samples>
                <extra_samples>10</extra_samples>
            </resourceLimitsQos>
        </topic>
    </profiles>

    <library_settings>
        <intraprocess_delivery>USER_DATA_ONLY</intraprocess_delivery>
    </library_settings>

    <log>
        <use_default>false</use_default>

        <consumer>
            <class>StdoutConsumer</class>
        </consumer>

        <consumer>
            <class>StdoutErrConsumer</class>
            <property>
                <name>stderr_threshold</name>
                <value>Log::Kind::Warning</value>
            </property>
        </consumer>

        <consumer>
            <class>FileConsumer</class>
            <property>
                <name>filename</name>
                <value>execution.log</value>
            </property>
            <property>
                <name>append</name>
                <value>TRUE</value>
            </property>
        </consumer>
    </log>

```

(continues on next page)

(continued from previous page)

```
    <types>
        <type> <!-- Types can be defined in its own type of tag or sharing the same tag -->
            <enum name="MyAloneEnumType">
                <enumerator name="A" value="0"/>
                <enumerator name="B" value="1"/>
                <enumerator name="C"/>
            </enum>
        </type>
        <type>
            <!-- All possible members struct type -->
            <struct name="MyFullStruct">
                <!-- Primitives & basic -->
                <member name="my_bool" type="boolean"/>
                <member name="my_char" type="char8"/>
                <member name="my_wchar" type="char16"/>
                <member name="my_byte" type="byte"/>
                <member name="my_octet" type="octet"/>
                <member name="my_uint8" type="uint8"/>
                <member name="my_short" type="int16"/>
                <member name="my_long" type="int32"/>
                <member name="my_unsignedshort" type="uint16"/>
                <member name="my_unsignedlong" type="uint32"/>
                <member name="my_longlong" type="int64"/>
                <member name="my_unsignedlonglong" type="uint64"/>
                <member name="my_float" type="float32"/>
                <member name="my_double" type="float64"/>
                <member name="my_longdouble" type="float128"/>
                <member name="my_string" type="string"/>
                <member name="my_wstring" type="wstring"/>

                <!-- string my_boundedString[41925] -->
                <member name="my_boundedString" type="string" stringMaxLength="41925"/>
                <!-- wstring my_boundedWString[41925] -->
                <member name="my_boundedWString" type="wstring" stringMaxLength="41925"/>

                <!-- short short_sequence[5]; -->
                <member name="short_sequence" sequenceMaxLength="5" type="int16"/>

                <!-- long long_array[2][3][4]; -->
                <member name="long_array" arrayDimensions="2,3,4" type="int32"/>

                <!-- map<long,long,2> my_map_inner -->
                <member name="my_map" type="int32" key_type="int32" mapMaxLength="2"/>
            </struct>

            <typedef name="inner_map" type="char8" key_type="int16"/>
            <struct name="MyComplexStruct">
                <!-- Complex types -->
                <member name="my_other_struct" type="nonBasic" nonBasicTypeName="MyFullStruct"/>
                <!-- map<long,map<long,long,2>,2> my_map_map; -->
```

(continues on next page)

(continued from previous page)

```
847             <member name="my_map_map" type="nonBasic" nonBasicTypeName="inner_map"␣
↪key_type="int32" mapMaxLength="2"/>
848         </struct>
849
850         <enum name="MyEnum">
851             <enumerator name="A" value="0"/>
852             <enumerator name="B" value="1"/>
853             <enumerator name="C"/>
854         </enum>
855
856         <typedef name="MyAlias1" type="nonBasic" nonBasicTypeName="MyEnum"/>
857         <typedef name="MyAlias2" type="int32" arrayDimensions="2,2"/>
858
859         <struct name="MyStruct">
860             <member name="first" type="int32"/>
861             <member name="second" type="int64"/>
862         </struct>
863
864         <struct name="OtherInheritedStruct" baseType="MyStruct">
865             <member name="my_enum" type="nonBasic" nonBasicTypeName="MyEnum"/>
866             <member name="my_struct" type="nonBasic" nonBasicTypeName="MyFullStruct"␣
↪arrayDimensions="5"/>
867         </struct>
868
869         <union name="MyUnion1">
870             <discriminator type="byte"/>
871             <case>
872                 <caseDiscriminator value="0"/>
873                 <caseDiscriminator value="1"/>
874                 <member name="first" type="int32"/>
875             </case>
876             <case>
877                 <caseDiscriminator value="2"/>
878                 <member name="second" type="nonBasic" nonBasicTypeName="MyStruct"/>
879             </case>
880             <case>
881                 <caseDiscriminator value="default"/>
882                 <member name="third" type="int64"/>
883             </case>
884         </union>
885
886         <bitset name="MyBitSet">
887             <bitfield name="a" bit_bound="3"/>
888             <bitfield name="b" bit_bound="10"/>
889             <bitfield name="c" bit_bound="12" type="int16"/>
890         </bitset>
891
892         <bitset name="OtherInheritedBitSet" baseType="MyBitSet">
893             <bitfield bit_bound="8"/>
894             <bitfield bit_bound="15" type="byte"/>
895         </bitset>
896
```

(continues on next page)

(continued from previous page)

```
            <bitmask name="MyBitMask" bit_bound="8">
                <bit_value name="flag0" position="0"/>
                <bit_value name="flag1"/>
            </bitmask>
        </type>
    </types>
</dds>
```

6.26 Environment variables

This is the list of environment variables that affect the behavior of *Fast DDS*:

6.26.1 FASTRTPS_DEFAULT_PROFILES_FILE

Defines the location of the default profile configuration XML file. If this variable is set and its value corresponds with an existing file, *Fast DDS* will load its profiles. For more information about XML profiles, please refer to *XML profiles*.

Linux

```
export FASTRTPS_DEFAULT_PROFILES_FILE=/home/user/profiles.xml
```

Windows

```
set FASTRTPS_DEFAULT_PROFILES_FILE=C:\profiles.xml
```

6.26.2 SKIP_DEFAULT_XML

Skips looking for a default profile configuration XML file. If this variable is set to *1*, *Fast DDS* will load the configuration parameters directly from the classes' definitions without looking for the *DEFAULT_FASTRTPS_PROFILES.xml* in the working directory. For more information about XML profiles, please refer to *XML profiles*.

Linux

```
export SKIP_DEFAULT_XML=1
```

Windows

```
set SKIP_DEFAULT_XML=1
```

6.26.3 FASTDDS_BUILTIN_TRANSPORTS

Setting this variable allows to modify the builtin transports that are initialized during the *DomainParticipant* creation. It is a simple way of changing the default configuration of the *Transport Layer* and it directly affects how DDS entities communicate between them.

All existing values, along with a brief description, are shown below:

Builtin Transports Options	Description
NONE	No transport will be instantiated. Hence, the user must manually add the desired transports. Otherwise, the participant creation will fail.
DEFAULT	UDPv4 and SHM transports will be instantiated. SHM transport has priority over the UDPv4 transport. Meaning that SHM will always be used when possible.
DEFAULTv6	UDPv6 and SHM transports will be instantiated. SHM transport has priority over the UDPv4 transport. Meaning that SHM will always be used when possible.
SHM	Only a SHM transport will be instantiated.
UDPv4	Only a UDPv4 transport will be instantiated.
UDPv6	Only a UDPv6 transport will be instantiated.
LARGE_DATA	UDPv4, TCPv4, and SHM transports will be instantiated. However, UDP will only be used for multicast announcements during the participant discovery phase (see *Discovery phases*) while the participant liveliness and the application data delivery occurs over TCP or SHM. This configuration is useful when working with large data.(See *Large Data mode and Fast DDS over TCP*).

Note: The environment variable is only used in the case where *use_builtin_transports* is set to TRUE. In any other case, the environment variable has no effect.

Note: TCPv4 transport is initialized with the following configuration:

- *calculate_crc*, *check_crc* and *apply_security* are set to false.
- *enable_tcp_nodelay* is set to true.

6.26.4 ROS_DISCOVERY_SERVER

Warning: The environment variable is only used in the case where *discovery protocol* is set to *SIMPLE*, *SERVER*, or *BACKUP*. In any other case, the environment variable has no effect.

Setting this variable configures the *DomainParticipant* to connect to one or more *servers* using the *Discovery Server* discovery mechanism.

- If ROS_DISCOVERY_SERVER is defined, and the DomainParticipant's *discovery protocol* is set to *SIMPLE*, then Fast DDS will instead configure it as *CLIENT* of the given *server*.
- If ROS_DISCOVERY_SERVER is defined, and the DomainParticipant's *discovery protocol* is set to *SERVER* or *BACKUP*, then the variable is used to add remote *servers* to the given *server*, leaving the *discovery protocol* as *SERVER* or *BACKUP* respectively.

- The value of the variable must list the locator of the server in the form of:
  - An IPv4 address like `192.168.2.23`. The UDP protocol is used by default. The UDP port can be appended using *:* as in `192.168.2.23:35665`.
  - An IPv6 address that follows RFC3513 address convention like `1080::8:800:200C:417A`. Again, it uses the UDP protocol by default. An UDP port can be appended like in `[1080::8:800:200C:417A]:35665`. Note the use of square brackets to avoid ambiguities.
  - TCPv4 specifier + IPv4 address like `TCPv4:[127.0.0.1]`. The TCP protocol is used to communicate with the server. The TCP port can be appended using *:* as in `TCPv4:[127.0.0.1]:42100`.
  - TCPv6 specifier + IPv6 address like `TCPv6:[::1]`. The TCP protocol is used to communicate with the server. The TCP port can be appended using *:* as in `TCPv6:[::1]:42100`.
  - A DNS name can be specified. This name will be used to query known hosts and available DNS servers to try to resolve valid IP addresses. Several formats are acceptable:
    - Plain domain name: `eprosima.com`. This will include all available IP addresses.
    - Domain name + port: `eprosima.com:35665`. As above but using a specific port.
    - UDPv4 specifier + domain name: `UDPv4:[eprosima.com]`. Only the first IPv4 address resolved will be used.
    - UDPv4 specifier + domain name + port: `UDPv4:[eprosima.com]:35665`. As above but using a specific port.
    - UDPv6 specifier + domain name: `UDPv6:[<dns>]`. Only the first IPv6 address resolved will be used.
    - UDPv6 specifier + domain name + port: `UDPv6:[<dns>]:35665`. As above but using a specific port.
    - TCPv4 specifier + domain name: `TCPv4:[eprosima.com]`. Only the first IPv4 address resolver will be used.
    - TCPv4 specifier + domain name + port: `TCPv4:[eprosima.com]:42100`. As above but using a specific port.
    - TCPv6 specifier + domain name: `TCPv6:[<dns>]`. Only the first IPv4 address resolver will be used.
    - TCPv6 specifier + domain name + port: `TCPv6:[<dns>]:42100`. As above but using a specific port.
- If no port is specified when using default UDP transport, the default port 11811 is used.
- If no port is specified when using TCP transport, the default port 42100 is used.
- To set more than one *server*'s address, they must be separated by semicolons.
- The server's ID is determined by their position in the list. Two semicolons together means the corresponding ID is free.
- When using IPv6 with DNS, the specified domain name space (<*dns*>) must be able to resolve to an IPv6 address. Otherwise an error will be raised.

The following example shows how to set the address of two remote discovery servers with addresses '84.22.259.329:8888' and 'localhost:1234' and IDs 0 and 2 respectively.

Linux
`export ROS_DISCOVERY_SERVER="84.22.259.329:8888;;localhost:1234"`
Windows
`set ROS_DISCOVERY_SERVER=84.22.259.329:8888;;localhost:1234`

Important: IP addresses specified in `ROS_DISCOVERY_SERVER` must be either valid IPv4/IPv6 addresses or domain names. If a name can be resolved into several addresses, it is possible to either use them all or restrict the selection to the first IPv4 using the *UDPv4:* or *TCPv4* prefixes or to the first IPv6 address using the *UDPv6:* or *TCPv6* prefixes.

Important: This environment variable is meant to be used in combination with *Fast DDS discovery CLI*. The *server*'s ID is used by *Fast DDS* to derived the `GuidPrefix_t` of the *server*. If the *server* is not instantiated using the CLI, the *server*'s GUID prefix should adhere to the same schema as the one generated from the CLI. Else, the *clients* configured with this environment variable will not be able to establish a connection with the *server*, thus not being able to connect to other *clients* either. The *server*'s GUID prefixes generated by the CLI comply with the following schema: `44.53.<server-id-in-hex>.5f.45.50.52.4f.53.49.4d.41`. This prefix schema has been chosen for its ASCII translation: `DS<id_in_hex>_EPROSIMA`.

Important: This environment variable can be changed at runtime adding new remote servers to a *SERVER*, *BACKUP* or *CLIENT* (that has been initialized with this environment variable previously) if loaded from an environment file using *FASTDDS_ENVIRONMENT_FILE*.

6.26.5 ROS_SUPER_CLIENT

If the `DomainParticipant`'s *discovery protocol* is set to *SIMPLE*, and `ROS_SUPER_CLIENT` is set to TRUE, the participant is automatically promoted to a *SUPER_CLIENT*.

Important: This environment variable is meant to be used in combination with `ROS_DISCOVERY_SERVER` to promote a participant from *SIMPLE* to *SUPER_CLIENT*. The participant will have the *servers* list defined in `ROS_DISCOVERY_SERVER`.

The possible values are: **TRUE**, **true**, **True**, **1**, **FALSE**, **false**, **False**, **0**.

Important: If the variable is not set, the default behavior of Fast DDS is equivalent to the case in which the variable is set to false.

The following example shows how to set the environment variable to true.

Linux
`export ROS_SUPER_CLIENT=TRUE`
Windows
`set ROS_SUPER_CLIENT=TRUE`

6.26.6 FASTDDS_STATISTICS

Warning: The environment variable is only used in the case where the CMake option *FASTDDS_STATISTICS* has been enabled. In any other case, the environment variable has no effect. Please, refer to *CMake options* for more information.

Setting this variable configures the *DomainParticipant* to enable the statistics DataWriters which topics are contained in the list set in this environment variable. The elements of the list should be separated by semicolons and match the *statistics topic name aliases*.

For example, to enable the statistics DataWriters that report the latency measurements, the environment variable should be set as follows:

Linux
`export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC"`
Windows
`set FASTDDS_STATISTICS=HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC`

Important: This environment variable can be used together with the XML profiles (for more information please refer to *Automatically enabling statistics DataWriters*). The statistics DataWriters that will be enabled is the union between the ones specified in the XML file (if loaded) and the ones stated in the environment variable (if set).

6.26.7 FASTDDS_ENVIRONMENT_FILE

Setting this environment variable to an existing `json` file allows to load the environment variables from the file instead of from the environment. This allows to change the value of some environment variables at run time with just modifying and saving the changes to the file. The environment value can be either an absolute or relative path. The file format is as follows:

```
{
    "environment_variable_name_1": "environment_variable_value_1",
    "environment_variable_name_2": "environment_variable_value_2"
```

(continues on next page)

(continued from previous page)

```
}
```

Important: The environment variables set in the environment file have precedence over the environment.

Warning: Currently only `ROS_DISCOVERY_SERVER` environment variable allows for changes at run time. (see *Modifying remote servers list at run time*)

6.27 PropertyPolicyQos Options

This section contains the list of *PropertyPolicyQos* that can be set with *Fast DDS*:

6.27.1 Non consolidated QoS

The *PropertyPolicyQos Options* are used to develop new *eProsima Extensions* QoS. Before consolidating a new QoS Policy, it is usually set using this generic QoS Policy. Consequently, this section is prone to frequent updates so the user is advised to check latest changes after upgrading to a different release version.

`DataWriter operating mode` QoS Policy

By default, *Fast DDS DataWriters* are enabled using `push mode`. This implies that they will add new samples into their queue, and then immediately deliver them to matched readers. For writers that produce non periodic bursts of data, this may imply saturating the network with a lot of packets, increasing the possibility of losing them on unreliable (i.e. UDP) transports. Depending on their QoS, DataReaders may also have to ignore some received samples, so they will have to be resent.

Configuring the DataWriters on `pull mode` offers an alternative by letting each reader pace its own data stream. It works by the writer notifying the reader what it is available, and waiting for it to request only as much as it can handle. At the cost of greater latency, this model can deliver reliability while using far fewer packets than `push mode`.

DataWriters periodically announce the state of their queue by means of a heartbeat. Upon reception of the heartbeat, DataReaders will request the DataWriter to send the samples they want to process. Consequently, the publishing rate can be tuned setting the heartbeat period accordingly. See *Tuning Heartbeat Period* for more details.

PropertyPolicyQos name	PropertyPolicyQos value	Default value
`"fastdds.push_mode"`	`"true"/"false"`	`"true"`

C++

```
DataWriterQos wqos;

// Enable pull mode
wqos.properties().properties().emplace_back(
    "fastdds.push_mode",
    "false");
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
<data_writer profile_name="pull_mode_datawriter_xml_profile">
    <propertiesPolicy>
        <properties>
            <!-- Enable pull mode -->
            <property>
                <name>fastdds.push_mode</name>
                <value>false</value>
            </property>
        </properties>
    </propertiesPolicy>
</data_writer>
</profiles>
```

Note:

- Communication to readers running on the same process (*Intra-process delivery*) will always use `push mode`.
- Communication to *BEST_EFFORT_RELIABILITY_QOS* readers will always use `push mode`.

Warning:

- It is inconsistent to enable the `pull mode` and also set the *ReliabilityQosPolicyKind* to *BEST_EFFORT_RELIABILITY_QOS*.
- It is inconsistent to enable the `pull mode` and also set the *heartbeatPeriod* to *c_TimeInfinite*.

Unique network flows QoS Policy

Warning: This section is still under work.

Statistics Module Settings

Fast DDS Statistics Module uses the *PropertyPolicyQos* to indicate the statistics DataWriters that are enabled automatically (see *Automatically enabling statistics DataWriters*). In this case, the property value is a semicolon separated list containing the *statistics topic name aliases* of those DataWriters that the user wants to enable.

PropertyPolicyQos name	PropertyPolicyQos value	Default value
"fastdds.statistics"	Semicolon separated list of *statistics topic name aliases*	""

C++

```
DomainParticipantQos pqos;

// Activate Fast DDS Statistics module
pqos.properties().properties().emplace_back("fastdds.statistics",
        "HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC
↪");
```

XML

```
<participant profile_name="statistics_domainparticipant_conf_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate Fast DDS Statistics Module -->
                <property>
                    <name>fastdds.statistics</name>
                    <value>HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
↪PHYSICAL_DATA_TOPIC</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Physical Data in Discovery Information

It is possible to include the information conveyed in the *PHYSICAL_DATA_TOPIC* into the participant discovery message, a.k.a *DATA[p]* (see *Discovery phases*). This is done by setting the following properties within the *PropertyPolicyQos*:

PropertyPolicyQos name	PropertyPolicyQos value	Default value without FASTDDS_STATISTICS	Default value with FASTDDS_STATISTICS
"fastdds.physical_data.host"	Name of the host computer in which the application runs	Not set	""
"fastdds.physical_data.user"	Name of the user running the application	Not set	""
"fastdds.physical_data.process"	Name of the process running the application	Not set	""

Whenever any of these properties is defined within the *DomainParticipantQos*, the *DomainParticipant* *DATA[p]* will contained the set value. Furthermore, if any of these properties is set to a value of "", which is the default when

FASTDDS_STATISTICS is defined (see *CMake options*), *Fast DDS* will automatically populate the value using the following convention:

- "fastdds.physical_data.host": Host name as returned by asio::ip::host_name(), followed by ":<default data sharing domain id>"
- "fastdds.physical_data.user": Name of the user running the application, or "unknown" if it could not be retrieved.
- "fastdds.physical_data.process": The process ID of the process in which the application is running.

All the previous entails that adding physical information to the *DATA[p]* can be done regardless of whether FASTDDS_STATISTICS is defined, and that it is possible to let *Fast DDS* set some default values into the reported host, user, and process:

1. If FASTDDS_STATISTICS is defined, and the user does not specify otherwise, *Fast DDS* will set default values to the physical properties of the *DATA[p]*.
2. If FASTDDS_STATISTICS is defined, and the user sets values to the properties, the user settings are honored.
3. If FASTDDS_STATISTICS is defined, and the user removes the physical properties from the *DomainParticipantQos*, then no physical information is transmitted in the *DATA[p]*.
4. If FASTDDS_STATISTICS is not defined, it is still possible to transmit physical information in the *DATA[p]* by setting the aforementioned properties:
   a) If set to "", then *Fast DDS* will populate their value according to the described rules.
   b) If set to something other than "", then the set value will be transmitted in the *DATA[p]* as-is.

In case FASTDDS_STATISTICS is defined, and the reporting of statistics over the DISCOVERY_TOPIC is enabled (see *Statistics Module Settings*), then the physical information included in the *DATA[p]* is also transmitted over the DISCOVERY_TOPIC (see *PHYSICAL_DATA_TOPIC*) whenever one *DomainParticipant* discovers another one.

C++

```
/* Create participant which announces default physical properties */
DomainParticipantQos pqos_default_physical;
// NOTE: If FASTDDS_STATISTICS is defined, then setting the properties to "" is not␣
↪necessary
pqos_default_physical.properties().properties().emplace_back("fastdds.physical_data.host
↪", "");
pqos_default_physical.properties().properties().emplace_back("fastdds.physical_data.user
↪", "");
pqos_default_physical.properties().properties().emplace_back("fastdds.physical_data.
↪process", "");
DomainParticipant* participant_with_physical = DomainParticipantFactory::get_instance()->
↪create_participant(0,
                pqos_default_physical);

/* Create participant which announces custom physical properties */
DomainParticipantQos pqos_custom_physical;
// NOTE: If FASTDDS_STATISTICS is defined, then clear the properties before setting them
// pqos_custom_physical.properties().properties().clear()
pqos_custom_physical.properties().properties().emplace_back("fastdds.physical_data.host",
↪ "custom_hostname");
pqos_custom_physical.properties().properties().emplace_back("fastdds.physical_data.user",
↪ "custom_username");
pqos_custom_physical.properties().properties().emplace_back("fastdds.physical_data.
↪process", "custom_process");
```

(continues on next page)

(continued from previous page)

```
DomainParticipant* participant_custom_physical = DomainParticipantFactory::get_
↪instance()->create_participant(0,
                pqos_custom_physical);

/* Create participant which does not announce physical properties */
DomainParticipantQos pqos_no_physical;
pqos_no_physical.properties().properties().clear();
DomainParticipant* participant_without_physical = DomainParticipantFactory::get_
↪instance()->create_participant(
    0, pqos_no_physical);

/* Load physical properties from default XML file */
DomainParticipantFactory::get_instance()->load_profiles();
DomainParticipantQos pqos_default_xml_physical =
        DomainParticipantFactory::get_instance()->get_default_participant_qos();
DomainParticipant* participant_default_xml_physical =
        DomainParticipantFactory::get_instance()->create_participant(0, pqos_default_xml_
↪physical);

/* Load physical properties from specific XML file */
DomainParticipantFactory::get_instance()->load_XML_profiles_file("somefile.xml");
DomainParticipantFactory::get_instance()->load_profiles();
DomainParticipantQos pqos_custom_xml_physical =
        DomainParticipantFactory::get_instance()->get_default_participant_qos();
DomainParticipant* participant_custom_xml_physical =
        DomainParticipantFactory::get_instance()->create_participant(0, pqos_custom_xml_
↪physical);
```

XML

```
<?xml version="1.0" encoding="utf-8"?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="statistics_participant" is_default_profile="true">
            <rtps>
                <propertiesPolicy>
                    <properties>
                        <property>
                            <name>fastdds.physical_data.host</name>
                            <value>custom_hostname</value>
                        </property>
                        <property>
                            <name>fastdds.physical_data.user</name>
                            <value>custom_username</value>
                        </property>
                        <property>
                            <name>fastdds.physical_data.process</name>
                            <value>custom_process</value>
                        </property>
                    </properties>
                </propertiesPolicy>
            </rtps>
```

(continues on next page)

(continued from previous page)

```
        </participant>
    </profiles>
</dds>
```

Important: The properties set using XML override those in the default QoS, which means that it is possible to set the physical properties using XML regardless of whether `FASTDDS_STATISTICS` is defined. However, it is not possible to remove the properties using XML, meaning that an application using *Fast DDS* with `FASTDDS_STATISTICS` enabled which does not want for the physical information to be transmitted in the `DomainParticipant` *DATA[p]* must remove the properties using the aforementioned *C++* API.

Endpoint Partitions

Fast DDS uses this *PropertyPolicyQos* to define which partitions does an endpoint belong to. This property follows the same logic regarding matching as the *PartitionQosPolicy* that can be defined for Publishers and Subscribers.

This property's value is a semicolon separated list containing the partition names the user wants this endpoint to belong to.

Important: If both a Publisher and one of its DataWriters have conflicting partition configuration, this is, a DataWriter has this property defined while the Publisher has the *PartitionQosPolicy* defined, the DataWriter configuration takes precedence and the Publisher *PartitionQosPolicy* is ignored for this endpoint. This applies to Subscribers and their DataReaders as well.

This property will be automatically set when creating DataReaders and DataWriters using the create_with_profile functions. It cannot be changed after the entity has been created.

PropertyPolicyQos name	PropertyPolicyQos value	Default value
`"partitions"`	Semicolon separated list of partition names	`""`

C++

```
DataWriterQos wqos;

// Add partitions
wqos.properties().properties().emplace_back(
    "partitions",
    "part1;part2");

DataReaderQos rqos;

// Add partitions
rqos.properties().properties().emplace_back(
    "partitions",
    "part1;part2");
```

XML

```
<data_writer profile_name="pub_partition_example">
    <qos>
        <partition>
            <names>
                <name>part1</name>
                <name>part2</name>
            </names>
        </partition>
    </qos>
</data_writer>

<data_reader profile_name="sub_partition_example">
    <qos>
        <partition>
            <names>
                <name>part1</name>
                <name>part2</name>
            </names>
        </partition>
    </qos>
</data_reader>
```

Static Discovery's Exchange Format

Static Discovery exchanges data in the Participant Discovery Phase (PDP). Currently there are two different exchange formats which can be selected using the property `dds.discovery.static_edp.exchange_format`.

PropertyPolicyQos value	Description	Default
"v1"	Standard exchange format for Static Discovery.	
"v1_Reduced"	Format which reduces the necessary network bandwidth to transmit Static Discovery's information in the Participant Discovery Phase (PDP).	

C++

```
DomainParticipantQos participant_qos;
participant_qos.properties().properties().emplace_back(
    "dds.discovery.static_edp.exchange_format",
    "v1_Reduced"
    );
```

XML

```
<participant profile_name="participant_xml_conf_static_discovery_format_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <property>
                    <name>dds.discovery.static_edp.exchange_format</name>
                    <value>v1_Reduced</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

SHM transport meta-traffic enforcement

A DomainParticipant will by default configure both a *UDP Transport* and a *Shared Memory Transport*. When a participant on another process in the same host is discovered, the endpoint discovery might be done using either transport.

Avoiding Shared Memory communication for discovery traffic can save valuable resources. The behavior regarding this can be configured using the property `fastdds.shm.enforce_metatraffic`.

Property-PolicyQos value	Description	Default
`"none"`	Use other transports for meta-traffic.	
`"unicast"`	Enable SHM transport unicast communications.	
`"all"`	Enable SHM transport unicast and multicast communications. This will enable discovery between SHM only participants and participants having several transports.	

Note: When SHM is the only transport configured for a participant, the setting of this property is ignored, and considered to be `"all"`.

C++

```
DomainParticipantQos participant_qos;

// SHM transport will listen for unicast meta-traffic
participant_qos.properties().properties().emplace_back(
    "fastdds.shm.enforce_metatraffic",
    "unicast");
```

XML

```
<participant profile_name="participant_xml_conf_shm_enforce_metatraffic_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <property>
                    <name>fastdds.shm.enforce_metatraffic</name>
                    <value>unicast</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

TCP Non-blocking send

TCP transport will by default configure a *TCP Transport* with blocking send calls. When `fastdds.tcp_transport.non_blocking_send` property is set to `true`, send operations will return immediately if the send buffer is full, but no error will be returned to the upper layer. This means that the application will behave as if the packet is sent and lost.

When set to `false`, send operations will block until the network buffer has space for the packet.

PropertyPolicyQos value	Description	Default
`"false"`	Block on send operations when send buffer is full.	
`"true"`	Do not block on send operations when send buffer is full.	

C++

```
DomainParticipantQos participant_qos;

// TCP transport will use non-blocking send
participant_qos.properties().properties().emplace_back(
    "fastdds.tcp_transport.non_blocking_send",
    "true");
```

XML

```
<participant profile_name="participant_xml_conf_tcp_non_blocking_send">
    <rtps>
        <propertiesPolicy>
            <properties>
                <property>
                    <name>fastdds.tcp_transport.non_blocking_send</name>
                    <value>true</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

6.27.2 Flow Controller Settings

When using *Flow Controllers*, the DataWriter may need specific parameters to be set. Properties related with this feature lie on the `fastdds.sfc` namespace.

- Property `fastdds.sfc.priority` is used to set the priority of the DataWriter for *HIGH_PRIORITY* and *PRIORITY_WITH_RESERVATION* flow controllers. Allowed values are from -10 (highest priority) to 10 (lowest priority). If the property is not present, it will be set to the lowest priority.
- Property `fastdds.sfc.bandwidth_reservation` is used to set the percentage of the bandwidth that the DataWriter is requesting for *PRIORITY_WITH_RESERVATION* flow controllers. Allowed values are from 0 to 100, and express a percentage of the total flow controller limit. If the property is not present, it will be set to 0 (no bandwidth is reserved for the DataWriter).

6.27.3 Persistence Service Settings

Warning: This section is still under work.

6.27.4 Security Plugins Settings

As described in the *Security* section, the security plugins admit a set of settings that can be configured.

Authentication plugin settings

The *DDS:Auth:PKI-DH* authentication plugin, can be activated setting the *DomainParticipantQos* `properties()` `dds.sec.auth.plugin` with the value `builtin.PKI-DH`. The following table outlines the properties used for the *DDS:Auth:PKI-DH* plugin configuration.

PropertyPolicyQos name	PropertyPolicyQos value
`identity_ca`	URI to the X.509 v3 certificate of the Identity CA in PEM format. Supported URI schemes: file.
`identity_certificate`	URI to a X.509 v3 certificate signed by the Identity CA in PEM format containing the signed public key for the Participant. Supported URI schemes: file.
`identity_crl` *(optional)*	URI to a X.509 Certificate Revocation List (CRL). Supported URI schemes: file.
`private_key`	URI to access the private Private Key for the Participant. Supported URI schemes: file, *PKCS#11*.
`password` *(optional)*	A password used to decrypt the *private_key*. If the *password* property is not present, then the value supplied in the *private_key* property must contain the decrypted private key. The *password* property is ignored if the *private_key* is given in PKCS#11 scheme.

Note: All properties listed above have the `dds.sec.auth.builtin.PKI-DH."` prefix. For example: `dds.sec.auth.builtin.PKI-DH.identity_ca`. For examples and further information, please refer to the *Authentication plugin: DDS:Auth:PKI-DH* section.

Authentication handshake settings

The authentication phase starts when discovery information is received from the remote *DomainParticipants*. At this moment, the participant sends a handshake request until a handshake response is received from the remote participant. Some parameters are involved in the behavior of this exchange:

- `max_handshake_requests` controls the maximum number of handshake requests to be sent.
- `initial_handshake_resend_period` represents the initial waiting time (in milliseconds) for the first handshake request that has to be resent.
- `handshake_resend_period_gain` is the gain against which the period is multiplied between two handshake requests.

Hence, the period of time to wait for sending a new handshake request is computed at each iteration as the period between the last two handshake requests multiplied by the gain (so that the period increases).

The following table lists the settings to configure the authentication handshake behavior within the `dds.sec.auth.builtin.PKI-DH` plugin:

PropertyPolicyQos name	PropertyPolicyQos value	PropertyPolicyQos bounds	Default value
`max_handshake_requests`	`<int>`	`[1, max)`	`10`
`initial_handshake_resend_period`	`<int>`	`[1, max)`	`125`
`handshake_resend_period_gain`	`<double>`	`(1.0, max)`	`1.5`

Note: All listed properties have the `dds.sec.auth.builtin.PKI-DH.` prefix. For example: `dds.sec.auth.builtin.PKI-DH.max_handshake_requests`.

The following is an example of how to set the properties of DomainParticipantQoS for the authentication handshake configuration.

C++

```
DomainParticipantQos pqos;

pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.max_handshake_requests",
    "5");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.initial_handshake_resend_period",
    "250");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.handshake_resend_period_gain",
    "1.5");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_auth_handshake_props_xml_
↪profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.max_handshake_requests</name>
                    <value>5</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.initial_handshake_resend_period</
↪name>
                    <value>250</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.handshake_resend_period_gain</
↪name>
                    <value>1.5</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Cryptographic plugin settings

The *DDS:Crypto:AES-GCM-GMAC* authentication plugin, can be activated setting the *DomainParticipantQos properties()* `dds.sec.crypto.plugin` with the value `builtin.AES-GCM-GMAC`. Moreover, this plugin needs the activation of the *Authentication plugin: DDS:Auth:PKI-DH*. The *DDS:Crypto:AES-GCM-GMAC* plugin is configured using the *Access control plugin: DDS:Access:Permissions*, i.e the cryptography plugin is configured through the properties and configuration files of the access control plugin. For further information and examples in this regard please refer to *Cryptographic plugin: DDS:Crypto:AES-GCM-GMAC*.

Logging plugin settings

The *DDS:Logging:DDS_LogTopic* authentication plugin, can be activated setting the *DomainParticipantQos properties()* `dds.sec.log.plugin` with the value `builtin.DDS_LogTopic`. The following table outlines the properties used for the DDS:Logging:DDS_LogTopic plugin configuration. For further information and examples follow the dedicated documentation: *Logging plugin: DDS:Logging:DDS_LogTopic*.

6.27.5 Logging Module Settings

Warning: This section is still under work.

6.27.6 Ignore Local Endpoints

By default, Fast DDS will automatically match all the endpoints (meaning *DataReaders* and *DataWriters*) belonging to a given *DomainParticipant* as soon as they share the same *Topic* and have compatible Qos. This however can result in undesired feedback whenever an application creates a *DataReader* and a *DataWriter* under the same *DomainParticipant* on a shared *Topic*. Although this feedback can be filtered out at the application level upon data reception by filtering out messages coming from a *DataWriter* belonging to the same *DomainParticipant* on the *DataReader* receiving the data (by looking at the *GuidPrefix_t*), this entails for a data sample to go all the way to the *DataReaderListener* just to be discarded by an overcomplicated application business logic. For this reason, Fast DDS offers the possibility of instructing the *DomainParticipant* to avoid the matching of local endpoints through the following property:

PropertyPolicyQos name	PropertyPolicyQos value	Default value
`"fastdds.ignore_local_endpoints"`	`"true"/"false"`	`"false"`

C++

```
DomainParticipantQos participant_qos;

// Avoid local matching of this participant's endpoints
participant_qos.properties().properties().emplace_back(
    "fastdds.ignore_local_endpoints",
    "true");
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <participant profile_name="ignore_local_endpoints_domainparticipant_xml_profile">
            <rtps>
                <propertiesPolicy>
                    <properties>
                        <!-- Avoid local matching of this participant's endpoints -->
                        <property>
                            <name>fastdds.ignore_local_endpoints</name>
                            <value>true</value>
                        </property>
                    </properties>
                </propertiesPolicy>
```

(continues on next page)

(continued from previous page)

```
            </rtps>
        </participant>
    </profiles>
</dds>
```

Note: An invalid value of `fastdds.ignore_local_endpoints` results in the default behaviour.

6.28 Dynamic Topic Types

eProsima Fast DDS provides a dynamic way to define and use topic types and topic data. Our implementation follows the *OMG Extensible and Dynamic Topic Types for DDS interface*. For more information, you can read the specification for DDS-XTypes V1.2.

The dynamic topic types offer the possibility to work over RTPS without the restrictions related to the IDLs. Using them, the users can declare the different types that they need and manage the information directly, avoiding the additional step of updating the IDL file and the generation of *C++* classes.

6.28.1 Overview of Dynamic Types

This section describes the classes related to dynamic types that are used through the rest of the documentation. At the bottom of the section you can also find a short example using the functionality.

Involved classes

The following class diagram describes the relationship among the classes related to dynamic types. Please, refer to the description of each class to find its purpose and the nature of the relationship with the rest of the classes.

Fig. 13: Dynamic types class diagram

- *DynamicType*
- *DynamicTypeBuilderFactory*
- *DynamicTypeBuilder*
- *TypeDescriptor*
- *DynamicTypeMember*
- *MemberDescriptor*
- *DynamicData*
- *DynamicDataFactory*
- *DynamicPubSubType*

DynamicType

Base class of all types declared dynamically. It represents a dynamic data type that can be used to create *DynamicData* values. By design, the structure of a dynamic type (its member fields) cannot be modified once the type is created.

DynamicTypeBuilderFactory

Singleton class that is in charge of the creation and the management of every DynamicType and *DynamicTypeBuilder*. It declares functions to create builders for each kind of supported types. Given a builder for a specific type, it can also create the corresponding DynamicType. Some simpler types can be created directly, avoiding the step of creating a DynamicTypeBuilder. Please, refer to the *Supported Types* documentation for details about which ones support this option.

Every object created by the factory must be deleted to avoid memory leaking. Refer to the *Memory management* section for details.

DynamicTypeBuilder

Intermediate class used to configure a DynamicType before it is created. By design, the structure of a DynamicType (its member fields) cannot be modified once the object is created. Therefore, all its structure must be defined prior to its creation. The builder is the object used to set up this structure.

Once defined, the DynamicTypeBuilderFactory is used to create the DynamicType from the information contained in the builder. As a shortcut, the builder exposes a function `build()` that internally uses the DynamicTypeBuilderFactory to return a fully constructed DynamicType. The types created with `build()` are still subject to the *Memory management* restrictions, and must be deleted by the DynamicTypeBuilderFactory.

Builders can be reused after the creation of a DynamicType, as the changes applied to the builder do not affect to types created previously.

TypeDescriptor

Stores the information about one type with its relationships and restrictions. This is the class that describes the inner structure of a DynamicType. The DynamicTypeBuilder has an internal instance of TypeDescriptor that modifies during the type building process. When the DynamicType is created, the DynamicTypeBuilderFactory uses the information of the TypeDescriptor in the builder to create the DynamicType. During the creation, the TypeDescriptor is copied to the DynamicType, so that it becomes independent from the DynamicTypeBuilder, and the builder can be reused for another type.

DynamicTypeMember

Represents a data member of a DynamicType that is also a DynamicType. Compound types (dynamic types that are composed of other dynamic types) have a DynamicTypeMember for every child DynamicType added to it.

MemberDescriptor

Just as a TypeDescriptor describes the inner structure of a DynamicType, a MemberDescriptor stores all the information needed to manage a DynamicTypeMember, like their name, their unique ID, or the default value after the creation. This information is copied to the *DynamicData* on its creation.

DynamicData

While a DynamicType *describes* a type, DynamicData represents a data instance of a DynamicType. It provides functions to access and modify the data values in the instance.

There are two ways to work with DynamicData:

- Activating the macro `DYNAMIC_TYPES_CHECKING`, which creates a variable for each primitive kind to help the debug process.
- Without this macro, the size of the DynamicData is reduced, using only the minimum needed internal values, but it makes the code harder to debug.

DynamicDataFactory

Singleton class that is in charge of the creation and the management of every DynamicData. It can take a DynamicType and create an instance of a corresponding DynamicData. Every data object created by the factory must be deleted to avoid memory leaking. Refer to the *Memory management* section for details.

It also allows to create a `TypeIdentifier` and a (Minimal and Complete) `TypeObject` from a `TypeDescriptor`.

DynamicPubSubType

This class is an adapter that allows using DynamicData on Fast DDS. It inherits from `TopicDataType` and implements the functions needed to communicate the DynamicData between Publishers and Subscribers.

Minimum example

This is a short example to illustrate the use of the dynamic types and how the classes describe above interact with each other. While the code snippet can be used as a quick reference for code building, the sequence diagram below provides a visual interpretation of the actions.

```
// Create a builder for a specific type
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_enum_
↪builder();

// Use the builder to configure the type
builder->add_empty_member(0, "DEFAULT");
builder->add_empty_member(1, "FIRST");
builder->add_empty_member(2, "SECOND");

// Create the data type using the builder
// The builder will internally use the DynamicTypeBuilderFactory to create the type
DynamicType_ptr type = builder->build();
```

(continues on next page)

(continued from previous page)

```
// Create a new data instance of the create data type
DynamicData_ptr data (DynamicDataFactory::get_instance()->create_data(type));

// Now we can set or read data values
data->set_int32_value(1);

// No need of deleting the objects, since we used the
// automanaged smart pointers
```

Fig. 14: Sequence diagram of the code above

6.28.2 Supported Types

In order to provide maximum flexibility and capability to the defined dynamic types, eProsima Fast DDS supports several member types, ranging from simple primitives to nested structures.

This section describes the basic (not nested) supported types. For more complex structures and examples, please, refer to *Complex Types*.

- *Primitive Types*
- *String and WString*
- *Alias*
- *Enumeration*
- *Bitmask*
- *Structure*
- *Bitset*
- *Union*
- *Sequence*
- *Array*
- *Map*

Primitive Types

This section includes every simple kind:

BOOLEAN	INT64
BYTE	UINT16
CHAR8	UINT32
CHAR16	UINT64
INT16	FLOAT32
INT32	FLOAT64
FLOAT128	

By definition, primitive types are self-described and can be created without configuration parameters. Therefore, *DynamicTypeBuilderFactory* exposes several functions to allow users create the dynamic type avoiding the *DynamicTypeBuilder* step. The DynamicTypeBuilder can still be used to create dynamic data of primitive types, as shown on the example below. The DynamicData class has a specific `get()` and `set()` functions for each primitive type of the list.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_int32_builder();
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicData* data = DynamicDataFactory::get_instance()->create_data(created_type);
data->set_int32_value(1);

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_type();
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pType);
data2->set_int32_value(1);
```

String and WString

Strings are pretty similar to primitive types, the main difference being that they need to set the size of the `buffer` that they can manage. By default this size is set to 255 characters.

DynamicTypeBuilderFactory exposes the functions `create_string_type()` and `create_wstring_type()` to allow users create the DynamicTypes avoiding the DynamicTypeBuilder step. The DynamicTypeBuilder can still be used to create String type dynamic data, as shown on the example below.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_string_builder(100);
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicData* data = DynamicDataFactory::get_instance()->create_data(created_type);
data->set_string_value("Dynamic String");

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_string_
↪type(100);
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pType);
data2->set_string_value("Dynamic String");
```

Alias

Alias types provide an alternative name to an already existing type. Once the DynamicData is created, users can access its information as if they were working with the base type.

DynamicTypeBuilderFactory exposes the function `create_alias_type()` to allow users create the Alias types avoiding the DynamicTypeBuilder step. The DynamicTypeBuilder can still be used to create Alias, as shown on the example below.

```
// Create the base type
DynamicTypeBuilder_ptr base_builder = DynamicTypeBuilderFactory::get_instance()->create_
↪string_builder(100);
```

(continues on next page)

(continued from previous page)

```
DynamicType_ptr base_type = DynamicTypeBuilderFactory::get_instance()->create_type(base_
↪builder.get());

// Create alias using Builders
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪builder(base_type,
                "alias");
DynamicData* data = DynamicDataFactory::get_instance()->create_data(builder.get());
data->set_string_value("Dynamic Alias String");

// Create alias type directly
DynamicType_ptr pAliasType = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪type(base_type, "alias");
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pAliasType);
data2->set_string_value("Dynamic Alias String");
```

Enumeration

An enumeration contains a set of supported values and a selected value among those supported. The supported values must be configured using the DynamicTypeBuilder, using the `add_member()` function for each supported value. The input to this function is the index and the name of the value we want to add.

The DynamicData class has functions `get_enum_value()` and `set_enum_value()` to work with value index or value name name strings.

```
// Add enumeration values using the DynamicTypeBuilder
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_enum_
↪builder();
builder->add_empty_member(0, "DEFAULT");
builder->add_empty_member(1, "FIRST");
builder->add_empty_member(2, "SECOND");

// Create the data instance
DynamicData* data = DynamicDataFactory::get_instance()->create_data(builder.get());

// Access value using the name
std::string sValue = "SECOND";
data->set_enum_value(sValue);
std::string sStoredValue;
data->get_enum_value(sStoredValue, MEMBER_ID_INVALID);

// Access value using the index
uint32_t uValue = 2;
data->set_enum_value(uValue);
uint32_t uStoredValue;
data->get_enum_value(uStoredValue, MEMBER_ID_INVALID);
```

Bitmask

Bitmasks are similar to *enumeration* types, but their members work as bit flags that can be individually turned on and off. Bit operations can be applied when testing or setting a bitmask value. DynamicData has the special functions `get_bitmask_value()` and `set_bitmask_value()` which allow to retrieve or modify the full value instead of accessing each bit.

Bitmasks can be bound to any number of bits up to 64.

```
uint32_t limit = 5; // Stores as "octet"

// Add bitmask flags using the DynamicTypeBuilder
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪bitmask_builder(limit);
builder->add_empty_member(0, "FIRST");
builder->add_empty_member(1, "SECOND");

// Create the data instance
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(builder.get()));

// Access the mask values using the name
data->set_bool_value(true, "FIRST");                // Set the "FIRST" bit
bool bSecondValue = data->get_bool_value("SECOND"); // Get the "SECOND" bit

// Access the mask values using the index
data->set_bool_value(true, 1);                      // Set the "SECOND" bit
bool bFirstValue = data->get_bool_value(0);         // Get the "FIRST" bit

// Get the complete bitmask as integer
uint64_t fullValue;
data->get_bitmask_value(fullValue);
```

Structure

Structures are the common complex types, they allow to add any kind of members inside them. They do not have any value, they are only used to contain other types.

To manage the types inside the structure, users can call the `get()` and `set()` functions according to the kind of the type inside the structure using their `ids`. If the structure contains a complex value, it should be used with `loan_value` to access to it and `return_loaned_value` to release that pointer. DynamicData manages the counter of loaned values and users can not loan a value that has been loaned previously without calling `return_loaned_value` before.

The `ids` must be consecutive starting by zero, and the DynamicType will change that Id if it doesn't match with the next value. If two members have the same Id, after adding the second one, the previous will change its Id to the next value. To get the Id of a member by name, DynamicData exposes the function `get_member_id_by_name()`.

```
// Build a structure with two fields ("first" as int32, "other" as uint64) using␣
↪DynamicTypeBuilder
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type());
builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->create_uint64_
↪type());
```

(continues on next page)

(continued from previous page)

```
DynamicType_ptr struct_type(builder->build());

// Create the data instance
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(struct_type));

// Access struct members
data->set_int32_value(5, 0);
data->set_uint64_value(13, 1);
```

Structures allow inheritance, exactly with the same OOP meaning. To inherit from another structure, we must create the structure calling the `create_child_struct_builder()` of the factory. This function is shared with bitsets and will deduce our type depending on the parent's type.

```
DynamicTypeBuilder_ptr child_builder =
        DynamicTypeBuilderFactory::get_instance()->create_child_struct_builder(builder.
↪get());
```

Bitset

Bitset types are similar to *structure* types, but their members are merely *bitfields*, which are stored optimally. In the static version of bitsets, each bit uses just one bit in memory (with platform limitations) without alignment considerations. A bitfield can be anonymous (cannot be addressed) to skip unused bits within a bitset.

Each bitfield in a bitset can be modified through their minimal needed primitive representation.

Number of bits	Primitive
1	BOOLEAN
2-8	UINT8
9-16	UINT16
17-32	UINT32
33-64	UINT64

Each bitfield (or member) works like its primitive type with the only difference that the internal storage only modifies the involved bits instead of the full primitive value.

Bit_bound and position of the bitfield can be set using annotations (useful when converting between static and dynamic bitsets).

```
// Create bitfields with the appropriate type for their size
DynamicTypeBuilder_ptr base_type_byte_builder =
        DynamicTypeBuilderFactory::get_instance()->create_byte_builder();
auto base_type_byte = base_type_byte_builder->build();

DynamicTypeBuilder_ptr base_type_uint32_builder =
        DynamicTypeBuilderFactory::get_instance()->create_uint32_builder();
auto base_type_uint32 = base_type_uint32_builder->build();

// Create the bitset with two bitfields
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪bitset_builder();
builder->add_member(0, "byte", base_type_byte);
```

(continues on next page)

(continued from previous page)

```
builder->add_member(1, "uint32", base_type_uint32);

// Apply members' annotations
builder->apply_annotation_to_member(0, ANNOTATION_POSITION_ID, "value", "0");   // "byte
↪" starts at position 0
builder->apply_annotation_to_member(0, ANNOTATION_BIT_BOUND_ID, "value", "2");  // "byte
↪" is 2 bit length
builder->apply_annotation_to_member(1, ANNOTATION_POSITION_ID, "value", "10");  //
↪"uint32" starts at position 10 (8 bits empty)
builder->apply_annotation_to_member(1, ANNOTATION_BIT_BOUND_ID, "value", "20"); //
↪"uint32" is 20 bits length

// Create the data instance
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(builder.get()));

// Access values
data->set_byte_value(234, 0);
data->set_uint32_value(2340, 1);
octet bValue;
uint32_t uValue;
data->get_byte_value(bValue, 0);
data->get_uint32_value(uValue, 1);
```

Bitsets allows inheritance, exactly with the same OOP meaning. To inherit from another bitset, we must create the bitset calling the `create_child_struct_builder` of the factory. This function is shared with structures and will deduce our type depending on the parent's type.

```
DynamicTypeBuilder_ptr child_builder =
        DynamicTypeBuilderFactory::get_instance()->create_child_struct_builder(builder.
↪get());
```

Union

Unions are a special kind of structures where only one of the members is active at the same time. To control these members, users must set the `discriminator` type that is going to be used to select the current member calling the `create_union_builder` function. The `discriminator` itself is a DynamicType of any primitive type, string type or union type.

Every member that is going to be added needs at least one `union_case_index` to set how it is going to be selected and, optionally, if it is the default value of the union.

```
// Create the union DynamicTypeBuilder with an int32 discriminator
DynamicType_ptr discriminator = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_union_
↪builder(discriminator);

// Add the union members. "firts" will be the default value
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type(), "", { 0 },
        true);
builder->add_member(0, "second", DynamicTypeBuilderFactory::get_instance()->create_int64_
↪type(), "", { 1 },
```

(continues on next page)

(continued from previous page)

```
        false);

// Create the data instance
DynamicType_ptr union_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(union_type));

// Access the values using the member index
data->set_int32_value(9, 0);
data->set_int64_value(13, 1);

// Get the label of the currently selected member
uint64_t unionLabel;
data->get_union_label(unionLabel);
```

Sequence

A complex type that manages its members as a list of items allowing users to insert, remove or access to a member of the list. To create this type users need to specify the type that it is going to store and optionally the size limit of the list.

To ease the memory management of this type, DynamicData has these functions:

- `insert_sequence_data()`: Creates a new element at the end of the list and returns the `id` of the new element.
- `remove_sequence_data()`: Removes the element of the given index and refreshes the `ids` to keep the consistency of the list.
- `clear_data()`: Removes all the elements of the list.

```
// Create a DynamicTypeBuilder for a sequence of two elements of type inte32
uint32_t length = 2;
DynamicType_ptr base_type = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicTypeBuilder_ptr builder =
        DynamicTypeBuilderFactory::get_instance()->create_sequence_builder(base_type,␣
↪length);

// Create the data instance
DynamicType_ptr sequence_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(sequence_type));

// Insert and remove elements
MemberId newId, newId2;
data->insert_int32_value(10, newId);
data->insert_int32_value(12, newId2);
data->remove_sequence_data(newId);
```

Array

Arrays are pretty similar to sequences with two main differences: they can have multiple dimensions and they do not need their elements to be stored consecutively.

An array needs to know the number of dimensions it is managing. For that, users must provide a vector with as many elements as dimensions in the array. Each element in the vector represents the size of the given dimension. If the value of an element is set to zero, the default value applies (`100`).

Id values on the `set()` and `get()` functions of DynamicData correspond to the array index. To ease the management of array elements, every `set()` function in DynamicData class creates the item if the given index is empty.

To ease the memory management of this type, DynamicData has these functions:

- `insert_array_data()`: Creates a new element at the end of the array and returns the `id` of the new element.
- `remove_array_data()`: Clears the element of the given index.
- `clear_data()`: Removes all the elements of the array.
- `get_array_index()`: Returns the position id giving a vector of indexes on every dimension that the arrays support, which is useful in multidimensional arrays.

```
// Create an array DynamicTypeBuilder for a 2x2 elements of type int32
std::vector<uint32_t> lengths = { 2, 2 };
DynamicType_ptr base_type = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicTypeBuilder_ptr builder =
        DynamicTypeBuilderFactory::get_instance()->create_array_builder(base_type,␣
↪lengths);

// Create the data instance
DynamicType_ptr array_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(array_type));

// Access elements in the multidimensional array
MemberId pos = data->get_array_index({1, 0});
data->set_int32_value(11, pos);
data->set_int32_value(27, pos + 1);
data->clear_array_data(pos);
```

Map

Maps contain a list of 'key-value' pair types, allowing users to insert, remove or modify the element types of the map. The main difference with sequences is that the map works with pairs of elements and creates copies of the key element to block the access to these elements.

To create a map, users must set the types of the key and the value elements, and, optionally, the size limit of the map.

To ease the memory management of this type, *DynamicData* has these functions:

- `insert_map_data()`: Inserts a new key value pair and returns the ids of the newly created key and value elements.
- `remove_map_data()`: Uses the given id to find the key element and removes the key and the value elements from the map.
- `clear_data()`: Removes all the elements from the map.

```
// Create DynamicTypeBuilder for a map of two pairs of {key:int32, value:int32}
uint32_t length = 2;
DynamicType_ptr base = DynamicTypeBuilderFactory::get_instance()->create_int32_type();
DynamicTypeBuilder_ptr builder =
        DynamicTypeBuilderFactory::get_instance()->create_map_builder(base, base,␣
↪length);

// Create the data instance
DynamicType_ptr map_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(map_type));

// Add a new element to the map with key 1
DynamicData_ptr key(DynamicDataFactory::get_instance()->create_data(base));
MemberId keyId;
MemberId valueId;
key->set_int32_value(1);
data->insert_map_data(key.get(), keyId, valueId);

// Add a new element to the map with key 2
// insert_map_data creates a copy of the key, so the same instance can be reused
MemberId keyId2;
MemberId valueId2;
key->set_int32_value(2);
data->insert_map_data(key.get(), keyId2, valueId2);

// Set the value to the element with key 2, using the returned value Id
data->set_int32_value(53, valueId2);

// Remove elements from the map
data->remove_map_data(keyId);
data->remove_map_data(keyId2);
```

6.28.3 Complex Types

If the application's data model is complex, it is possible to combine the *basic types* to create complex types, including nested composed types (structures within structures within unions). Types can also be extended using inheritance, improving the flexibility of the definition of the data types to fit the model.

The following subsections describe these *complex types* and their use.

- *Nested structures*
- *Structure inheritance*
- *Alias of an alias*
- *Unions with complex types*

Nested structures

Structures can contain other structures as members. The access to these compound members is restricted and managed by the *DynamicData* instance. Users must request access calling `loan_value` before using them, and release them with `return_loaned_value` once they finished. The loan operation will fail if the member is already loaned and has not been released yet.

```
// Create a struct type
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type());
builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->create_uint64_
↪type());
DynamicType_ptr struct_type = builder->build();

// Create a struct type with the previous struct as member
DynamicTypeBuilder_ptr parent_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_struct_builder();
parent_builder->add_member(0, "child_struct", struct_type);
parent_builder->add_member(1, "second", DynamicTypeBuilderFactory::get_instance()->
↪create_int32_type());
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(parent_builder.
↪get()));

// Access the child struct with the loan operations
DynamicData* child_data = data->loan_value(0);
child_data->set_int32_value(5, 0);
child_data->set_uint64_value(13, 1);
data->return_loaned_value(child_data);
```

Structure inheritance

To inherit a structure from another one, use the `create_child_struct_type` function from *DynamicTypeBuilderFactory*. The resultant type contains all members from the base class and the new ones added to the child.

Structures support several levels of inheritance, so the base class can be another derived type itself.

```
// Create a base struct type
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type());
builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->create_uint64_
↪type());

// Create a struct type derived from the previous struct
DynamicTypeBuilder_ptr child_builder =
        DynamicTypeBuilderFactory::get_instance()->create_child_struct_builder(builder.
↪get());

// Add new members to the derived type
builder->add_member(2, "third", DynamicTypeBuilderFactory::get_instance()->create_uint64_
↪type());
```

(continues on next page)

(continued from previous page)

```
// Create the data instance
DynamicType_ptr struct_type = child_builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(struct_type));

// The derived type includes the members defined on the base type
data->set_int32_value(5, 0);
data->set_uint64_value(13, 1);
data->set_uint64_value(47, 2);
```

Alias of an alias

Alias types support recursion, simply use an alias name as base type for `create_alias_type()`.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_string_builder(100);
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪builder(
    created_builder.get(), "alias");
DynamicTypeBuilder_ptr builder2 = DynamicTypeBuilderFactory::get_instance()->create_
↪alias_builder(
    builder.get(), "alias2");
DynamicData* data(DynamicDataFactory::get_instance()->create_data(builder2->build()));
data->set_string_value("Dynamic Alias 2 String");

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_string_
↪type(100);
DynamicType_ptr pAliasType = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪type(pType, "alias");
DynamicType_ptr pAliasType2 =
        DynamicTypeBuilderFactory::get_instance()->create_alias_type(pAliasType, "alias2
↪");
DynamicData* data2(DynamicDataFactory::get_instance()->create_data(pAliasType));
data2->set_string_value("Dynamic Alias 2 String");
```

Unions with complex types

Unions support complex type fields. The access to these complex type fields is restricted and managed by the *DynamicData* instance. Users must request access calling `loan_value` before using them, and release them with `return_loaned_value` once they finished. The loan operation will fail if the fields is already loaned and has not been released yet.

```
// Create a union DynamicTypeBuilder
DynamicType_ptr discriminator = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_union_
↪builder(discriminator);
```

(continues on next page)

(continued from previous page)

```
// Add a int32 to the union
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type(), "", { 0 },
        true);

// Create a struct type and add it to the union
DynamicTypeBuilder_ptr struct_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_struct_builder();
struct_builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type());
struct_builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->create_
↪uint64_type());
builder->add_member(1, "first", struct_builder.get(), "", { 1 }, false);

// Create the union data instance
DynamicType_ptr union_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(union_type));

// Access the struct member using the loan operations
DynamicData* child_data = data->loan_value(1);
child_data->set_int32_value(9, 0);
child_data->set_int64_value(13, 1);
data->return_loaned_value(child_data);
```

6.28.4 Annotations

DynamicTypeBuilder allows applying an annotation to both current type and inner members with the functions:

- `apply_annotation()`
- `apply_annotation_to_member()`

Both functions take the name, the key and the value of the annotation. `apply_annotation_to_member()` additionally receives the `MemberId` of the inner member.

For example, if we define an annotation like:

```
@annotation MyAnnotation
{
    long value;
    string name;
};
```

And then we apply it through IDL to a struct:

```
@MyAnnotation(5, "length")
struct MyStruct
{
    // ...
}
```

The equivalent code using *DynamicType* will be:

```
// Apply the annotation
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
//...
builder->apply_annotation("MyAnnotation", "value", "5");
builder->apply_annotation("MyAnnotation", "name", "length");
```

Builtin annotations

The following annotations modifies the behavior of DynamicTypes:

- `@position`: When applied to *Bitmask*, sets the position of the flag,
  as expected in the IDL annotation.
  If applied to *Bitset*, sets the base position of the bitfield,
  useful to identify unassigned bits.
- `@bit_bound`: Applies to *Bitset*. Sets the size in bits of the bitfield.
- `@key`: Alias for `@Key`. See *Data types with a key* section for more details.
- `@default`: Sets a default value for the member.
- `@non_serialized`: Excludes a member from being serialized.

6.28.5 Dynamic Types Discovery and Endpoint Matching

When using *DynamicType* support, *Fast DDS* checks the optional *TypeObject* and *TypeIdentifier* values during endpoint matching. Currently, the matching only verifies that both endpoints are using the same topic data type, but will not negotiate about it.

The process of checking the types is as follows:

- It checks `CompleteTypeObject` on TypeObject first.
- If one or both endpoints do not define the `CompleteTypeObject`, it tries with `MinimalTypeObject`.
- If one or both endpoints do not define `MinimalTypeObject` either, it compares the TypeIdentifier.
- If none is defined, then just the type name is checked.

If one of the endpoints transmits a `CompleteTypeObject`, *Discovery-Time Data Typing* can be performed.

TypeObject

TypeObject fully describes a data type, the same way as the IDL representation does. There are two kinds of TypeObjects: `CompleteTypeObject` and `MinimalTypeObject` .

- `CompleteTypeObject` fully describes the type, the same way as the IDL representation does.
- `MinimalTypeObject` is a compact representation of the data type, that contains only the information relevant for the remote Endpoint to be able to interpret the data.

TypeObject is an IDL union with both *Minimal* and *Complete* representation. Both are described in the annexes of DDS-XTypes V1.2 document, please refer to this document for details.

TypeInformation

TypeInformation is an extension of *XTypes 1.2* that allow Endpoints to share information about data types without sending the TypeObject. Endpoints instead share a TypeInformation containing the TypeIdentifier of the data type. Then each Endpoint can request the complete TypeObject for the data types it is interested in. This avoids sending the complete data type to Endpoints that may not be interested.

TypeInformation is described in the annexes of DDS-XTypes V1.2 document, please refer to this document for details.

TypeIdentifier

TypeIdentifier provides a unique way to identify each type. For basic types, the information contained in the TypeIdentifier completely describes the type, while for complex ones, it serves as a search key to retrieve the complete TypeObject.

TypeIdentifier is described in the annexes of DDS-XTypes V1.2 document, please refer to this document for details.

TypeObjectFactory

Singleton class that manages the creation and access for every registered TypeObject and TypeIdentifier. It can generate a full *DynamicType* from a basic TypeIdentifier (i.e., one whose discriminator is not `EK_MINIMAL` or `EK_COMPLETE`).

Fast DDS-Gen

Fast DDS-Gen supports the generation of *XXXTypeObject.h* and *XXXTypeObject.cxx* files, taking `XXX` as our IDL type. These files provide a small Type Factory for the type `XXX`. Generally, these files are not used directly, as now the type `XXX` will register itself through its factory to TypeObjectFactory in its constructor, making it very easy to use static types with dynamic types.

Discovery-Time Data Typing

Using the Fast DDS API, when a participant discovers a remote endpoint that sends a complete TypeObject or a simple TypeIdentifier describing a type that the participant does not know, the participant listener's function *on_type_discovery* is called with the received TypeObject or TypeIdentifier, and, when possible, a pointer to a *DynamicType* ready to be used.

Discovery-Time Data Typing allows the discovering of simple DynamicTypes. A TypeObject that depends on other TypeObjects, cannot be built locally using Discovery-Time Data Typing and should use *TypeLookup Service* instead.

To ease the sharing of the TypeObject and TypeIdentifier used by Discovery-Time Data Typing, *TopicDataType* contains a function member named *auto_fill_type_object*. If set to true, the local participant will send the TypeObject and TypeIdentifier to the remote endpoint during discovery.

TypeLookup Service

Using the Fast DDS API, when a participant discovers an endpoint that sends a type information describing a type that the participant doesn't know, the participant listener's function *on_type_information_received()* is called with the received TypeInformation. The user can then try to retrieve the full TypeObject hierarchy to build the remote type locally, using the TypeLookup Service.

To enable this builtin TypeLookup Service, the user must enable it in the *QoS* of the *DomainParticipant*:

C++

```
DomainParticipantQos qos;
qos.wire_protocol().builtin.typelookup_config.use_client = true;
qos.wire_protocol().builtin.typelookup_config.use_server = true;
```

XML

```
<participant profile_name="participant_typelookup_service_enabled">
    <rtps>
        <builtin>
            <typelookup_config>
                <use_client>true</use_client>
                <use_server>true</use_server>
            </typelookup_config>
        </builtin>
    </rtps>
</participant>
```

A participant can be enabled to act as a TypeLookup server, client, or both.

The process of retrieving the remote type from its TypeInformation, and then registering it, can be simplified using the *register_remote_type* function on the *DomainParticipant*. This function takes the name of the type, the type information, and a callback function. Internally it uses the TypeLookup Service to retrieve the full TypeObject, and, if successful, it will call the callback.

This callback has the following signature:

```
void(std::string& type_name, const DynamicType_ptr type)
```

- **type_name**: Is the name given to the type when calling *register_remote_type*, to allow the same callback to be used across different calls.
- **type**: If the *register_remote_type* was able to build and register a *DynamicType*, this parameter contains a pointer to the type. Otherwise it contains `nullptr`. In the latter case, the user can still try to build the type manually using the factories, but it is very likely that the build process will fail.

TopicDataType contains a data member named *auto_fill_type_information*. If set to true, the local participant will send the type information to the remote endpoint during discovery.

6.28.6 Serialization

Dynamic Types have their own pubsub type like any class generated with an IDL, and their management is pretty similar to them.

```
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_type();
DynamicPubSubType pubsubType(pType);

// SERIALIZATION EXAMPLE
DynamicData* pData = DynamicDataFactory::get_instance()->create_data(pType);
uint32_t payloadSize = static_cast<uint32_t>(pubsubType.
↪getSerializedSizeProvider(pData)());
SerializedPayload_t payload(payloadSize);
pubsubType.serialize(pData, &payload);

// DESERIALIZATION EXAMPLE
types::DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pType);
pubsubType.deserialize(&payload, data2);
```

A member can be marked to be ignored by serialization with the annotation `@non_serialized`.

6.28.7 XML profiles

Dynamic Types profiles allows *eProsima Fast DDS* to create DynamicTypes directly defining them through XML. This allows any application to change *TopicDataTypes* without the need to change its source code.

Please, refer to *Dynamic Types profiles* for further information about how to use this feature.

6.28.8 Memory management

Memory management is critical for dynamic types since every dynamic type and dynamic data is managed with pointers. Every object stored inside of a dynamic object is managed by its owner, and users must delete every object they create using the factories.

```
DynamicTypeBuilder* pBuilder = DynamicTypeBuilderFactory::get_instance()->create_uint32_
↪builder();
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_type();
DynamicData* pData = DynamicDataFactory::get_instance()->create_data(pType);

DynamicTypeBuilderFactory::get_instance()->delete_builder(pBuilder);
DynamicDataFactory::get_instance()->delete_data(pData);
```

To ease this management, the library defines smart pointers (`DynamicTypeBuilder_ptr`, `DynamicType` and `DynamicData_ptr`) that will delete the objects automatically when they are not needed anymore. `DynamicType` will always be returned as `DynamicType_ptr` because there is no internal management of its memory.

```
DynamicTypeBuilder_ptr pBuilder = DynamicTypeBuilderFactory::get_instance()->create_
↪uint32_builder();
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_type();
DynamicData_ptr pData(DynamicDataFactory::get_instance()->create_data(pType));
```

The only case where these smart pointers cannot be used is with functions `loan_value` and `return_loaned_value`. Raw pointers should be used with these functions, because the returned value should not be deleted, and using a smart pointer with them will cause a crash.

6.28.9 Dynamic HelloWorld Examples

These are complete working examples that make use of dynamic types. You can explore them to find how this feature connects to the rest of *Fast DDS*, and learn how to integrate it in your own application.

DynamicHelloWorldExample

This example is in folder examples/cpp/dds/DynamicHelloWorldExample of the Fast DDS GitHub repository. It shows the use of DynamicType generation to provide the *TopicDataType*. This example is compatible with the classic HelloWorldExample.

As a quick reference, the following piece of code shows how the HelloWorld type is created using DynamicTypes:

```
// In HelloWorldPublisher.h
// Dynamic Types
eprosima::fastrtps::types::DynamicData* m_DynHello;
eprosima::fastrtps::types::DynamicPubSubType m_DynType;

// In HelloWorldPublisher.cpp
// Create basic builders
DynamicTypeBuilder_ptr struct_type_builder(DynamicTypeBuilderFactory::get_instance()->
↪create_struct_builder());

// Add members to the struct.
struct_type_builder->add_member(0, "index", DynamicTypeBuilderFactory::get_instance()->
↪create_uint32_type());
struct_type_builder->add_member(1, "message", DynamicTypeBuilderFactory::get_instance()->
↪create_string_type());
struct_type_builder->set_name("HelloWorld");

DynamicType_ptr dynType = struct_type_builder->build();
m_DynType.SetDynamicType(dynType);
m_DynHello = DynamicDataFactory::get_instance()->create_data(dynType);
m_DynHello->set_uint32_value(0, 0);
m_DynHello->set_string_value("HelloWorld", 1);
```

DDSDynamicHelloWorldExample

This example uses the DDS API, and can be retrieve from folder examples/cpp/dds/DynamicHelloWorldExample of the Fast DDS GitHub repository. It shows a publisher that loads a type from an XML file, and shares it during discovery. The subscriber discovers the type using *Discovery-Time Data Typing*, and registers the discovered type on the `on_type_discovery()` listener function.

TypeLookupService

This example uses the DDS API, and it is located in folder examples/cpp/dds/TypeLookupService of the Fast DDS GitHub repository. It is very similar to DDSDynamicHelloWorldExample, but the shared type is complex enough to require the TypeLookup Service due to the dependency of inner struct types. Specifically, it uses the `register_remote_type` approach with a callback.

6.29 Typical Use-Cases

Fast DDS is highly configurable, which allows for its use in a large number of scenarios. This section provides configuration examples for the following typical use cases when dealing with distributed systems:

- *Large Data mode and Fast DDS over TCP*. Describes how to configure *Fast DDS* to use the `LARGE_DATA` builtin transports mode. This mode enables efficient utilization of TCP transport without the need for constant reconfiguration during deployment changes. It optimizes communication performance for large data samples over lossy networks by employing a combination of UDP and TCP/SHM transports.
- *Fast DDS over WIFI*. Presents a case where *Discovery* through multicast communication is a challenge. This example shows how to:
  - Configure an initial list of peers with the address-port pairs of the remote participants (see *Configuring Initial Peers*).
  - Disable the multicast discovery mechanism (see *Disabling multicast discovery*).
  - Configure a SERVER discovery mechanism (see *Discovery Server*).
- *Well Known Network Deployments*. Describes a situation where the entire entity network topology (Participants, Publishers, Subscribers, and their addresses and ports) are known beforehand. In these kind of environments, *Fast DDS* allows to completely avoid the discovery phase configuring a STATIC discovery mechanism.
- *Topics with many subscribers*. In cases where there are many *DataReaders* subscribed to the same *Topic*, using multicast delivery can help reducing the overhead in the network and CPU.
- *Large Data Rates*. Presents configuration options that can improve the performance in scenarios where the amount of data exchanged between a *Publisher* and a *Subscriber* is large, either because of the data size or because the message rate. The examples describe how to:
  - Use TCP based communications (see *Large Data mode and Fast DDS over TCP*).
  - Configure the socket buffer size (see *increase the buffers size*).
  - Limit the publication rate (see *Flow Controllers*).
  - Tune the size of the socket buffers (see *Increasing socket buffers size*).
  - Tune the Heartbeat period (see *Tuning Heartbeat Period*).
  - Configure a non-strict reliable mode (see *Using Non-strict Reliability*).
- *Real-time behavior*. Describes the configuration options that allows using *Fast DDS* on a real-time scenario. The examples describe how to:

  - Configure memory management to avoid dynamic memory allocation (see *Tuning allocations*).
  - Limit the blocking time of API functions to have a predictable response time (see *Non-blocking calls*).

- *Reduce memory usage*. For use cases with memory consumption constraints, *Fast DDS* can be configured to reduce memory footprint to a minimum by adjusting different QoS policies.
- *Zero-Copy communication*. Under certain constraints, *Fast DDS* can provide application level communication between publishing and subscribing nodes avoiding any data copy during the process.
- *Unique network flows*. This use case illustrates the APIs that allow for the request of unique network flows, and for the identification of those in use.
- *Dynamic network interfaces*. If the network interfaces are expected to change while the application is running, *Fast DDS* provides an easy way of re-scanning the available interfaces and including them.
- *Statistics module*. This use case explains how to enable the Statistics module within the monitored application, and how to create a statistics monitoring application.
- *ROS 2 using Fast DDS middleware*. Since *Fast DDS* is the default middleware implementation in every OSRF Robot Operation System 2 (ROS 2) long term (LTS) releases and most of the non-LTS releases, this documentation includes a whole independent section to show the use of the library in ROS 2, and how to take full advantage of *Fast DDS* wide set of capabilities in a ROS 2 project.
- *How to use eProsima DDS Record and Replay (rosbag2 and DDS)*. Instructions on how to tune your application to be able to record and replay your DDS messages using ROS 2 rosbag2 package.
- *Request-Reply communication*. Although *Fast DDS* provides a *Publisher-Subscriber* communication protocol, it offers a mechanism to use *Request-Reply* communications. This use case explains how to use the APIs to communicate two application using *Request-Reply*.

6.29.1 Large Data mode and Fast DDS over TCP

As explained in *TCP Transport*, Fast DDS offers the possibility to communicate nodes within distributed applications with DDS over a TCP transport layer. This has the advantage of leveraging the builtin flow control and reliability of the TCP protocol, which in the context of DDS applications is best suited for the transmission of large payloads, i.e. large data samples, over lossy networks. Examples of such cases would be transmitting video or large point clouds resulting from laser scanning over WiFi links.

The configuration of the TCP transport typically involves an *a priori* knowledge of the deployment in order to set *Initial peers* for *Discovery*, which may not always be possible and creates difficulties when reallocating nodes of the distributed applications, as the entire discovery configuration needs to be changed. To overcome this problem, Fast DDS presents the `LARGE_DATA` builtin transports configuration as an approach for leveraging the Fast DDS' TCP transport capabilities while at the same time not requiring configuration modifications when the deployment changes over time.

`LARGE_DATA` has been specifically designed to improve communication performance of large data samples over lossy networks. When configured, UDP transport will exclusively be used during the *PDP discovery* phase, taking advantage of the more reliable TCP/SHM for the remainder of the communication process. Fast DDS offers an extremely straightforward implementation for this mode through an environment variable, XML profiles or via code.

For a video demonstration showcasing a practical example of this configuration, please refer to: Large Data communication with ROS 2.

Also, it is possible to enable TCP communication while using *Discovery Server* to manage *Discovery*.

Large Data Mode

The following snippets show how to configure *Fast DDS DomainParticipants* to run the *PDP discovery* phase over UDP multicast and communicate application data over a *TCP Transport* or *Shared Memory Transport*, which is called the `LARGE_DATA` configuration (See *Managing the Builtin Transports*). With this approach, applications managing large samples can benefit from transmitting their data over TCP or SHM, while at the same time have the flexibility of automatic discovery. The `LARGE_DATA` mode can be set using the `FASTDDS_BUILTIN_TRANSPORTS` environment variable (see *FASTDDS_BUILTIN_TRANSPORTS*), XML profiles or via code.

Environment Variable

```
export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <!--
            UDP transport for PDP and SHM/TCPv4 transport for both EDP and application␣
↪data
        -->
        <participant profile_name="large_data_builtin_transports" is_default_profile=
↪"true">
            <rtps>
                <builtinTransports>LARGE_DATA</builtinTransports>
            </rtps>
        </participant>
    </profiles>
</dds>
```

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos = PARTICIPANT_QOS_DEFAULT;

/* Transports configuration */
// UDPv4 transport for PDP over multicast and SHM / TCPv4 transport for EDP and␣
↪application data
pqos.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_DATA);

/* Create participant as usual */
eprosima::fastdds::dds::DomainParticipant* participant =
        eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);
```

Note: `LARGE_DATA` configuration of the builtin transports will also create a SHM transport along the UDP and TCP transports. Shared Memory will be used whenever it is possible. Manual configuration will be required if a TCP communication is required when SHM is feasible.

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos = PARTICIPANT_QOS_DEFAULT;

/* Transports configuration */
```

(continues on next page)

(continued from previous page)

```
// UDPv4 transport for PDP over multicast
auto pdp_transport = std::make_shared<eprosima::fastdds::rtps::UDPv4TransportDescriptor>
↪();
pqos.transport().user_transports.push_back(pdp_transport);

// TCPv4 transport for EDP and application data (The listening port must to be unique for
// each participant in the same host)
constexpr uint16_t tcp_listening_port = 0;
auto data_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↪();
data_transport->add_listener_port(tcp_listening_port);
pqos.transport().user_transports.push_back(data_transport);

pqos.transport().use_builtin_transports = false;

/* Locators */
// Define locator for PDP over multicast
eprosima::fastrtps::rtps::Locator_t pdp_locator;
pdp_locator.kind = LOCATOR_KIND_UDPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(pdp_locator, "239.255.0.1");
pqos.wire_protocol().builtin.metatrafficMulticastLocatorList.push_back(pdp_locator);

// Define locator for EDP and user data
eprosima::fastrtps::rtps::Locator_t tcp_locator;
tcp_locator.kind = LOCATOR_KIND_TCPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(tcp_locator, "0.0.0.0");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(tcp_locator, tcp_listening_port);
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(tcp_locator, tcp_listening_port);
pqos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(tcp_locator);
pqos.wire_protocol().default_unicast_locator_list.push_back(tcp_locator);

/* Create participant as usual */
eprosima::fastdds::dds::DomainParticipant* participant =
        eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <transport_descriptors>
            <!--
                UDP transport for PDP
            -->
            <transport_descriptor>
                    <transport_id>pdp_transport</transport_id>
                    <type>UDPv4</type>
            </transport_descriptor>
            <!--
                TCP transport for both EDP and application data
            -->
```

(continues on next page)

(continued from previous page)

```
            <transport_descriptor>
                <transport_id>data_transport</transport_id>
                <type>TCPv4</type>
                <!--
                    Set listening port for the transport.
                    This needs to be unique for each participant in the host
                -->
                <listening_ports>
                    <port>0</port>
                </listening_ports>
            </transport_descriptor>
        </transport_descriptors>

        <participant profile_name="udp_multicast_pdp_w_tcp_data" is_default_profile="true
↪">
            <rtps>
                <!--
                    Set participant transports
                -->
                <userTransports>
                    <transport_id>pdp_transport</transport_id>
                    <transport_id>data_transport</transport_id>
                </userTransports>
                <useBuiltinTransports>false</useBuiltinTransports>
                <!--
                    Discovery configuration
                -->
                <builtin>
                    <!--
                        PDP over UDP multicast
                    -->
                    <metatrafficMulticastLocatorList>
                        <locator>
                            <udpv4>
                                <address>239.255.0.1</address>
                            </udpv4>
                        </locator>
                    </metatrafficMulticastLocatorList>
                    <!--
                        EDP over TCP
                    -->
                    <metatrafficUnicastLocatorList>
                        <locator>
                            <tcpv4>
                                <address>0.0.0.0</address>
                                <physical_port>12345</physical_port>
                                <port>12345</port>
                            </tcpv4>
                        </locator>
                    </metatrafficUnicastLocatorList>
                </builtin>
                <!--
```

(continues on next page)

(continued from previous page)

```
                    Application data over TCP
                -->
                <defaultUnicastLocatorList>
                    <locator>
                        <tcpv4>
                            <address>0.0.0.0</address>
                            <physical_port>12345</physical_port>
                            <port>12345</port>
                        </tcpv4>
                    </locator>
                </defaultUnicastLocatorList>
            </rtps>
        </participant>
    </profiles>
</dds>
```

TCP Communication with Discovery Server

Fast DDS Discovery Server consists on a client-server discovery mechanism, in which a server *DomainParticipant* operates as the central point of communication. It collects and processes the metatraffic sent by the client DomainParticipants, and then distributes the appropriate information among the rest of the clients. An extended description of the feature can be found at *Discovery Server Settings*.

To use TCP communication along with Discovery Server, both the server participant and the client participant need to use custom user transports. There exists several ways of configuring the server participant, being *Fast DDS CLI* the fastest solution:

Fast DDS CLI

It can be configured to work over a TCP transport layer by using the arguments `-t` and `-q` to set up the IP address and the TCP port, respectively. After sourcing the environment, the following command can be used to instantiate a server listening on localhost and port 12345 (see *CLI*).

```
fastdds discovery -i 0 -t 127.0.0.1 -q 12345
```

C++

The following snippet can be used to instantiate a server on IP 192.168.10.57 listening on port 12345.

```
eprosima::fastdds::dds::DomainParticipantQos qos = PARTICIPANT_QOS_DEFAULT;

// Configure the current participant as SERVER
qos.wire_protocol().builtin.discovery_config.discoveryProtocol =␣
↪eprosima::fastrtps::rtps::DiscoveryProtocol_t::SERVER;

// Add custom user transport with TCP port 12345
auto data_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↪();
data_transport->add_listener_port(12345);
qos.transport().user_transports.push_back(data_transport);

// Define the listening locator to be on interface 192.168.10.57 and port 12345
```

(continues on next page)

(continued from previous page)

```
constexpr uint16_t tcp_listening_port = 12345;
eprosima::fastrtps::rtps::Locator_t listening_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(listening_locator, "192.168.10.57");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(listening_locator, tcp_listening_
↪port);
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(listening_locator, tcp_listening_
↪port);
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(listening_locator);

// Set the GUID prefix to identify this server
std::istringstream("44.53.00.5f.45.50.52.4f.53.49.4d.41") >> qos.wire_protocol().prefix;
```

XML

The following snippet can be used to instantiate a server on IP 192.168.10.57 listening on port 12345.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>server_tcp_transport</transport_id>
                <type>TCPv4</type>
                <!--
                    Set listening port for the transport.
                    This port is where the clients will connect.
                -->
                <listening_ports>
                    <port>12345</port>
                </listening_ports>
            </transport_descriptor>
        </transport_descriptors>
        <participant profile_name="TCP_SERVER" is_default_profile="true">
            <rtps>
                <builtin>
                    <discovery_config>
                        <discoveryProtocol>SERVER</discoveryProtocol>
                    </discovery_config>
                    <metatrafficUnicastLocatorList>
                        <locator>
                            <tcpv4>
                                <address>192.168.10.57</address>
                                <port>12345</port>
                                <physical_port>12345</physical_port>
                            </tcpv4>
                        </locator>
                    </metatrafficUnicastLocatorList>
                </builtin>
                <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
                <useBuiltinTransports>false</useBuiltinTransports>
                <userTransports>
                    <transport_id>server_tcp_transport</transport_id>
                </userTransports>
```

(continues on next page)

(continued from previous page)

```
            </rtps>
        </participant>
    </profiles>
</dds>
```

Fast DDS Discovery Server Example

It can be configured to work over a TCP transport layer by using the argument `--transport tcpv4`. The IP address and the TCP port can be set up with arguments `--listening-address` and `--listening-port`, respectively. From the *DiscoveryServerExample* folder, the following command can be used to instantiate a server listening on localhost and port 12345.

```
./DiscoveryServerExample server --transport tcpv4 --listening-address 127.0.0.1 --
↪listening-port 12345
```

The client participant can be configured by either using the `ROS_DISCOVERY_SERVER` environment variable (see *ROS_DISCOVERY_SERVER*) or by manually setting it.

Environment Variable

To configure a client participant to communicate over the TCP transport layer with the `ROS_DISCOVERY_SERVER` environment variable, the prefix *TCPv4* needs to be used. The following command can be used to configure the variable to set up a client using TCP communication and connecting to a server on localhost and port 12345.

```
export ROS_DISCOVERY_SERVER=TCPv4:[127.0.0.1]:12345
```

C++

The following snippet can be used to instantiate a client that will try to connect to a server on IP 192.168.10.57 and port 12345, that is, the server instantiated above.

```
eprosima::fastdds::dds::DomainParticipantQos qos = PARTICIPANT_QOS_DEFAULT;

// Configure the current participant as SERVER
qos.wire_protocol().builtin.discovery_config.discoveryProtocol =␣
↪eprosima::fastrtps::rtps::DiscoveryProtocol_t::CLIENT;

// Add custom user transport with TCP port 0 (automatic port assignation)
auto data_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↪();
data_transport->add_listener_port(0);
qos.transport().user_transports.push_back(data_transport);

// Define the server locator to be on interface 192.168.10.57 and port 12345
constexpr uint16_t server_port = 12345;
eprosima::fastrtps::rtps::Locator_t server_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(server_locator, "192.168.10.57");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(server_locator, server_port);
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(server_locator, server_port);

// Define the server attributes
eprosima::fastrtps::rtps::RemoteServerAttributes remote_server_att;
remote_server_att.metatrafficUnicastLocatorList.push_back(server_locator);

// Set the GUID prefix to identify this server
```

(continues on next page)

(continued from previous page)

```
std::istringstream("44.53.00.5f.45.50.52.4f.53.49.4d.41") >> remote_server_att.
↪guidPrefix;

// Add the server
qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(remote_server_
↪att);
```

XML

The following snippet can be used to instantiate a client that will try to connect to a server on IP 192.168.10.57 and port 12345, that is, the server instantiated above.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <profiles>
        <transport_descriptors>
            <transport_descriptor>
                    <transport_id>client_tcp_transport</transport_id>
                    <type>TCPv4</type>
                    <!--
                        Set listening port for the transport to 0.
                        This automatically assigns a port.
                    -->
                    <listening_ports>
                        <port>0</port>
                    </listening_ports>
            </transport_descriptor>
        </transport_descriptors>
        <participant profile_name="TCP_CLIENT" is_default_profile="true">
            <rtps>
                <builtin>
                    <discovery_config>
                        <discoveryProtocol>CLIENT</discoveryProtocol>
                        <discoveryServersList>
                            <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
                                <!--
                                    Server locator specifying where it is listening
                                -->
                                <metatrafficUnicastLocatorList>
                                    <locator>
                                        <tcpv4>
                                            <address>192.168.10.57</address>
                                            <port>12345</port>
                                            <physical_port>12345</physical_port>
                                        </tcpv4>
                                    </locator>
                                </metatrafficUnicastLocatorList>
                            </RemoteServer>
                        </discoveryServersList>
                    </discovery_config>
                </builtin>
                <useBuiltinTransports>false</useBuiltinTransports>
                <userTransports>
```

(continues on next page)

(continued from previous page)

```
                    <transport_id>client_tcp_transport</transport_id>
                </userTransports>
            </rtps>
        </participant>
    </profiles>
</dds>
```

6.29.2 Fast DDS over WIFI

The RTPS v2.2 standard defines the SIMPLE *Discovery* as the default mechanism for discovering participants in the network. One of the main features of this mechanism is the use of multicast communication in the Participant Discovery Phase (PDP). This can be a problem in cases where WiFi communication is used, since multicast is not as reliable over WiFi as it is over ethernet.

The recommended solution to this challenge is to configure an initial list of remote peers on the *DomainParticipant*, so that it can set unicast communication with them. This way, the use of multicast is not needed to discover these initial peers. Furthermore, if all the peers are known and configured beforehand, all multicast communication can be removed.

Alternatively, **Discovery Server** can be used to avoid multicast discovery. A DomainParticipant with a well-know address acts as a discovery server, providing the rest of the participants the information required to connect among them. If all the peers are known and configured beforehand, STATIC discovery can be used instead, completely avoiding the discovery phase. Use-case *Well Known Network Deployments* provides a detailed explanation on how to configure *Fast DDS* for STATIC discovery.

Configuring Initial Peers

A complete description of the initial peers list and its configuration can be found in *Initial peers*. For convenience, this example shows how to configure an initial peers list with one peer on host `192.168.10.13` with participant ID `1` in domain `0`.

Note: Note that the port number used here is not arbitrary, as discovery ports are defined by the RTPS v2.2 standard. Refer to *Well Known Ports* to learn about these standard port numbers.

If the participant ID is not known, setting *TransportDescriptorInterface* `maxInitialPeersRange` to at least the maximum expected number of DomainParticipants will ensure discovery and communication.

C++

```
DomainParticipantQos qos;

// configure an initial peer on host 192.168.10.13.
// The port number corresponds to the well-known port for metatraffic unicast
// on participant ID `1` and domain `0`.
Locator_t initial_peer;
IPLocator::setIPv4(initial_peer, "192.168.10.13");
initial_peer.port = 7412;
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="initial_peers_example_profile" is_default_profile="true
↪">
        <rtps>
            <builtin>
                <initialPeersList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.13</address>
                            <port>7412</port>
                        </udpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

Disabling multicast discovery

If all the peers are known and configured on the initial peer list beforehand, it is possible to disable the multicast meta traffic completely, as all *DomainParticipants* can communicate among them through unicast.

The complete description of the procedure to disable multicast discovery can be found at *Disabling all Multicast Traffic*. For convenience, however, this example shows how to disable all multicast traffic configuring one *metatraffic unicast* locator. Consideration should be given to the assignment of the ports in the `metatrafficUnicastLocatorList`, avoiding the assignment of ports that are not available or do not match the address-port listed in the *intial peers list* of the peer participant.

C++

```
DomainParticipantQos qos;

// configure one metatraffic unicast locator on interface 192.168.10.13.
// on participant ID `1` and domain `0`.
Locator_t meta_unicast_locator;
IPLocator::setIPv4(meta_unicast_locator, "192.168.10.13");
meta_unicast_locator.port = 7412;
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(meta_unicast_
↪locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="initial_peers_multicast_avoidance" is_default_profile=
↪"true" >
        <rtps>
            <builtin>
                <!-- Choosing a specific unicast address -->
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.13</address>
                            <port>7412</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

Discovery Server

During *Discovery*, the Participant Discovery Phase (PDP) relies on meta traffic announcements sent to multicast addresses so that all the *DomainParticipants* in the network can acknowledge each other. This phase is followed by a Endpoint Discovery Phase (EDP) where all the DomainParticipants use discovered unicast addresses to exchange information about their *Publisher* and *Subscriber* entities with the rest of the DomainParticipants, so that matching between entities of the same topic can occur.

Fast DDS provides a client-server discovery mechanism, in which a server DomainParticipant operates as the central point of communication. It collects and processes the metatraffic sent by the client DomainParticipants, and then distributes the appropriate information among the rest of the clients.

A complete description of the feature can be found at *Discovery Server Settings*. The following subsections present configurations for different discovery server use cases.

- *UDPv4 basic example setup*
- *UDPv4 redundancy example*

- *UDPv4 persistency example*
- *UDPv4 partitioning using servers*

UDPv4 basic example setup

To configure the Discovery Server scenario, two types of participants are created: the server participant and the client participant. Two parameters to be configured in this type of implementation are outlined:

- **Server GUID Prefix**: This is the unique identifier of the server.
- **Server Address-port pair**: Specifies the IP address and port of the machine that implements the server. Any free random port can be used. However, using *RTPS standard ports* is discouraged.

SERVER

C++

```
DomainParticipantQos qos;

// Configure the current participant as SERVER
qos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_
↪t::SERVER;

// Define the listening locator to be on interface 192.168.10.57 and port 56542
Locator_t server_locator;
IPLocator::setIPv4(server_locator, "192.168.10.57");
server_locator.port = 56542;
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(server_locator);

// Set the GUID prefix to identify this server
std::istringstream("72.61.73.70.66.61.72.6d.74.65.73.74") >> qos.wire_protocol().
↪prefix;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="UDP SERVER" is_default_profile="true">
        <rtps>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.57</address>
                            <port>56542</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
            <prefix>72.61.73.70.66.61.72.6d.74.65.73.74</prefix>
        </rtps>
    </participant>
</profiles>
```

CLIENT

C++

```
DomainParticipantQos qos;

// Configure the current participant as CLIENT
qos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_
↪t::CLIENT;

// Define a locator for the SERVER Participant on address 192.168.10.57 and port 56542
Locator_t remote_server_locator;
IPLocator::setIPv4(remote_server_locator, "192.168.10.57");
remote_server_locator.port = 56542;

RemoteServerAttributes remote_server_attr;
remote_server_attr.metatrafficUnicastLocatorList.push_back(remote_server_locator);

// Set the GUID prefix to identify the remote server
remote_server_attr.ReadguidPrefix("72.61.73.70.66.61.72.6d.74.65.73.74");

// Connect to the SERVER at the previous locator
qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(remote_
↪server_attr);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="UDP CLIENT" is_default_profile="true">
        <rtps>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>CLIENT</discoveryProtocol>
                    <discoveryServersList>
                        <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.74">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.57</address>
                                        <port>56542</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                            <metatrafficMulticastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.58</address>
                                        <port>24565</port>
                                    </udpv4>
                                </locator>
                            </metatrafficMulticastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

UDPv4 redundancy example

The *basic setup example* presents a *single point of failure*. That is, if the server fails the clients are not able to perform the discovery. To prevent this, several servers could be linked to each client. Then, a discovery failure only takes place if *all servers* fail, which is a more unlikely event.

In the example below, the values have been chosen to ensure each server has a unique *GUID Prefix* and *unicast address-port pair*. Note that several servers can share the same IP address but their port numbers should be different. Likewise, several servers can share the same port if their IP addresses are different.

Prefix	UDPv4 address-port
75.63.2D.73.76.72.[illegible]	[illegible]
75.63.2D.73.76.72.[illegible]	[illegible]

SERVER

C++

```
// Configure first server's locator on interface 192.168.10.57 and port 56542
Locator_t server_locator_1;
IPLocator::setIPv4(server_locator_1, "192.168.10.57");
server_locator_1.port = 56542;

// Configure participant_1 as SERVER listening on the previous locator
DomainParticipantQos server_1_qos;
server_1_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::SERVER;
std::istringstream("75.63.2D.73.76.72.63.6C.6E.74.2D.31") >> server_1_qos.wire_
↪protocol().prefix;
server_1_qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(server_
↪locator_1);

// Configure second server's locator on interface 192.168.10.60 and port 56543
Locator_t server_locator_2;
IPLocator::setIPv4(server_locator_2, "192.168.10.60");
server_locator_2.port = 56543;

// Configure participant_2 as SERVER listening on the previous locator
DomainParticipantQos server_2_qos;
server_2_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::SERVER;
std::istringstream("75.63.2D.73.76.72.63.6C.6E.74.2D.32") >> server_2_qos.wire_
↪protocol().prefix;
server_2_qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(server_
↪locator_2);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="UDP SERVER 1">
        <rtps>
            <prefix>75.63.2D.73.76.72.63.6C.6E.74.2D.31</prefix>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.57</address>
                            <port>56542</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>

    <participant profile_name="UDP SERVER 2">
        <rtps>
            <prefix>75.63.2D.73.76.72.63.6C.6E.74.2D.32</prefix>
            <builtin>
                <discovery_config>
```

CLIENT

C++

```
// Define a locator for the first SERVER Participant
Locator_t remote_server_locator_1;
IPLocator::setIPv4(remote_server_locator_1, "192.168.10.57");
remote_server_locator_1.port = 56542;

RemoteServerAttributes remote_server_attr_1;
remote_server_attr_1.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.31");
remote_server_attr_1.metatrafficUnicastLocatorList.push_back(remote_server_locator_1);

// Define a locator for the second SERVER Participant
Locator_t remote_server_locator_2;
IPLocator::setIPv4(remote_server_locator_2, "192.168.10.60");
remote_server_locator_2.port = 56543;

RemoteServerAttributes remote_server_attr_2;
remote_server_attr_2.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.32");
remote_server_attr_2.metatrafficUnicastLocatorList.push_back(remote_server_locator_2);

// Configure the current participant as CLIENT connecting to the SERVERS at the␣
↪previous locators
DomainParticipantQos client_qos;
client_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::CLIENT;
client_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_attr_1);
client_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_attr_2);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="UDP CLIENT REDUNDANCY">
        <rtps>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>CLIENT</discoveryProtocol>
                    <discoveryServersList>
                        <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.31">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.57</address>
                                        <port>56542</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                        <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.32">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.60</address>
                                        <port>56543</port>
                                    </udpv4>
                                </locator>
```

UDPv4 persistency example

On Discovery Server, servers gather and maintain the information of all connected endpoints, and distribute it to the clients. In case of a server failure, all this information is lost and the server needs to recover it on restart. In the *basic setup* this is done starting over the *Discovery* process. Given that servers usually have lots of clients associated, this is very time consuming.

Alternatively, *Fast DDS* allows to synchronize the server's discovery record to a file, so that the information can be loaded back into memory during the restart. This feature is enabled specifying the *Discovery Protocol* as **BACKUP**.

The record file is located on the server's process working directory, and named following the pattern *server-<GUIDPREFIX>.db* (for example: *server-73-65-72-76-65-72-63-6C-69-65-6E-74.db*). Once the server is created, it automatically looks for this file. If it already exists, its contents are loaded, avoiding the need of re-discovering the clients. To make a fresh restart, any such backup file must be removed or renamed before launching the server.

UDPv4 partitioning using servers

Server association can be seen as another isolation mechanism besides *Domains* and *Partitions*. Clients that do not share a server cannot see each other and belong to isolated server networks. For example, in the following figure, *client 1* and *client 2* cannot communicate even if they are on the same physical network and Domain.

Fig. 15: Clients cannot see each other due to server isolation

However, it is possible to connect server isolated networks very much as physical networks can be connected through routers:

- *Option 1*: Connecting the clients to several servers, so that the clients belong to several networks.
- *Option 2*: Connecting one server to another, so that the networks are linked together.
- *Option 3*: Create a new server linked to the servers to which the clients are connected.

Options 1 and 2 can only be implemented by modifying QoS values or XML configuration files beforehand. In this regard they match the domain and partition strategy. Option 3, however, can be implemented at runtime, when the isolated networks are already up and running.

Option 1

Connect each client to both servers. This case matches the *redundancy use case* already introduced.

Option 2

Connect one server to the other. This means configuring one of the servers to act as a client of the other.

Consider two servers, each one managing an isolated network:

NetPrefix work	UDPv4 address
A	75.63.2D.73.76.72.[illegible]
B	75.63.2D.73.76.72.[illegible]

In order to communicate both networks we can set server A to act as a client of server B:

C++

```
DomainParticipantQos qos;

// Configure current Participant as SERVER on address 192.168.10.60
Locator_t server_locator;
IPLocator::setIPv4(server_locator, "192.168.10.60");
server_locator.port = 56543;

qos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_
↪t::SERVER;
std::istringstream("75.63.2D.73.76.72.63.6C.6E.74.2D.31") >> qos.wire_protocol().
↪prefix;
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(server_locator);

// Add the connection attributes to the remote server.
Locator_t remote_server_locator;
IPLocator::setIPv4(remote_server_locator, "192.168.10.57");
remote_server_locator.port = 56542;

RemoteServerAttributes remote_server_attr;
remote_server_attr.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.32");
remote_server_attr.metatrafficUnicastLocatorList.push_back(remote_server_locator);

qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(remote_
↪server_attr);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="UDP SERVER A">
        <rtps>
            <prefix>75.63.2D.73.76.72.63.6C.6E.74.2D.31</prefix>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                    <discoveryServersList>
                        <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.32">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.57</address>
                                        <port>56542</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.60</address>
                            <port>56543</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
```

Option 3

Create a new server linked to the servers to which the clients are connected.

Consider two servers (A and B), each one managing an isolated network, and a third server (C) that will be used to connect the first two:

Server	Prefix	UDPv4 address
A	75.63.2D.73.76.72.2D.63.6C.69.65.6E.74.2D.63.33	192.168.1.45:65143
B	75.63.2D.73.76.72.2D.63.6C.69.65.6E.74.2D.63.32	192.168.1.45:65142
C	75.63.2D.73.76.72.2D.63.6C.69.65.6E.74.2D.63.31	192.168.1.45:65141

In order to communicate both networks we can setup server C to act as client of servers A and B as follows:

C++

```
DomainParticipantQos qos;

// Configure current Participant as SERVER on address 192.168.10.60
Locator_t server_locator;
IPLocator::setIPv4(server_locator, "192.168.10.54");
server_locator.port = 56541;

qos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_
↪t::SERVER;
std::istringstream("75.63.2D.73.76.72.63.6C.6E.74.2D.33") >> qos.wire_protocol().
↪prefix;
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(server_locator);

// Add the connection attributes to the remote server A.
Locator_t remote_server_locator_A;
IPLocator::setIPv4(remote_server_locator_A, "192.168.10.60");
remote_server_locator_A.port = 56543;

RemoteServerAttributes remote_server_attr_A;
remote_server_attr_A.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.31");
remote_server_attr_A.metatrafficUnicastLocatorList.push_back(remote_server_locator_A);

qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(remote_
↪server_attr_A);

// Add the connection attributes to the remote server B.
Locator_t remote_server_locator_B;
IPLocator::setIPv4(remote_server_locator_B, "192.168.10.57");
remote_server_locator_B.port = 56542;

RemoteServerAttributes remote_server_attr_B;
remote_server_attr_B.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.32");
remote_server_attr_B.metatrafficUnicastLocatorList.push_back(remote_server_locator_B);

qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(remote_
↪server_attr_B);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="UDP SERVER C">
        <rtps>
            <prefix>75.63.2D.73.76.72.63.6C.6E.74.2D.33</prefix>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                    <discoveryServersList>
                        <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.32">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>192.168.10.57</address>
                                        <port>56542</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
```

6.29.3 Well Known Network Deployments

It is often the case in industrial deployments, such as productions lines, that the entire network topology (hosts, IP addresses, etc.) is known beforehand. Such scenarios are perfect candidates for *Fast DDS* STATIC *Discovery* mechanism, which drastically reduces the middleware setup time (time until all the entities are ready for information exchange), while at the same time limits the connections to those strictly necessary.

Knowing the complete network topology allows to:

- Minimize the PDP meta-traffic and avoid multicast communication with *Peer-to-Peer Participant Discovery Phase*.
- Completely avoid the EDP with *STATIC Endpoint Discovery Phase*.

Peer-to-Peer Participant Discovery Phase

The SIMPLE PDP discovery phase entails the *DomainParticipants* sending periodic PDP announcements over multicast, and answering to the announcements received from remote DomainParticipants. As a result, the number of PDP connections grows quadratically with the number of DomainParticipants, resulting in a large amount of meta traffic on the network.

However, if all DomainParticipants are known beforehand, they can be configured to send their announcements only to the unicast addresses of their peers. This is done by specifying a list of peer addresses, and by disabling the participant multicast announcements. As an additional advantage, with this method only the peers configured on the list are known to the DomainParticipant, allowing to arrange which participant will communicate with which. This reduces the amount of meta traffic if not all the DomainParticipants need to be aware of all the rest of the remote participants present in the network.

Use-case *Fast DDS over WIFI* provides a detailed explanation on how to configure *Fast DDS* for such cases.

STATIC Endpoint Discovery Phase

Users can manually configure which *Publisher* and *Subscriber* match with each other, so they can start sharing user data right away, avoiding the EDP phase.

A complete description of the feature can be found at *STATIC Discovery Settings*. There is also a fully functional helloworld example implementing STATIC EDP in the examples/cpp/dds/StaticHelloWorldExample folder.

The following subsections present an example configuration where a Publisher in Topic `HelloWorldTopic` from DomainParticipant `HelloWorldPublisher` is matched with a Subscriber from DomainParticipant `HelloWorldSubscriber`.

Create STATIC discovery XML files

HelloWorldPublisher.xml

```
<staticdiscovery>
    <participant>
        <name>HelloWorldPublisher</name>
        <writer>
            <userId>1</userId>
            <entityID>2</entityID>
            <topicName>HelloWorldTopic</topicName>
            <topicDataType>HelloWorld</topicDataType>
        </writer>
    </participant>
</staticdiscovery>
```

HelloWorldSubscriber.xml

```
<staticdiscovery>
    <participant>
        <name>HelloWorldSubscriber</name>
        <reader>
            <userId>3</userId>
            <entityID>4</entityID>
            <topicName>HelloWorldTopic</topicName>
            <topicDataType>HelloWorld</topicDataType>
        </reader>
    </participant>
</staticdiscovery>
```

Create entities and load STATIC discovery XML files

When creating the entities, the local writer/reader attributes must match those defined in the STATIC discovery XML file loaded by the remote entity.

PUBLISHER

C++

```
// Participant configuration
DomainParticipantQos participant_qos;
participant_qos.name("HelloWorldPublisher");
participant_qos.wire_protocol().builtin.discovery_config.use_SIMPLE_
↪EndpointDiscoveryProtocol = false;
participant_qos.wire_protocol().builtin.discovery_config.use_STATIC_
↪EndpointDiscoveryProtocol = true;
participant_qos.wire_protocol().builtin.discovery_config.static_edp_xml_config(
↪"HelloWorldSubscriber.xml");

// DataWriter configuration
DataWriterQos writer_qos;
writer_qos.endpoint().user_defined_id = 1;
writer_qos.endpoint().entity_id = 2;

// Create the DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, participant_
↪qos);
if (nullptr == participant)
{
    // Error
    return;
}

// Create the Publisher
Publisher* publisher =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher)
{
    // Error
    return;
}

// Create the Topic with the appropriate name and data type
std::string topic_name = "HelloWorldTopic";
std::string data_type = "HelloWorld";
Topic* topic =
        participant->create_topic(topic_name, data_type, TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create the DataWriter
DataWriter* writer =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == writer)
{
    // Error
    return;
}
```

XML

SUBSCRIBER

C++

```
// Participant configuration
DomainParticipantQos participant_qos;
participant_qos.name("HelloWorldSubscriber");
participant_qos.wire_protocol().builtin.discovery_config.use_SIMPLE_
↪EndpointDiscoveryProtocol = false;
participant_qos.wire_protocol().builtin.discovery_config.use_STATIC_
↪EndpointDiscoveryProtocol = true;
participant_qos.wire_protocol().builtin.discovery_config.static_edp_xml_config(
↪"HelloWorldPublisher.xml");

// DataWriter configuration
DataWriterQos writer_qos;
writer_qos.endpoint().user_defined_id = 3;
writer_qos.endpoint().entity_id = 4;

// Create the DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, participant_
↪qos);
if (nullptr == participant)
{
    // Error
    return;
}

// Create the Subscriber
Subscriber* subscriber =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}

// Create the Topic with the appropriate name and data type
std::string topic_name = "HelloWorldTopic";
std::string data_type = "HelloWorld";
Topic* topic =
        participant->create_topic(topic_name, data_type, TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create the DataReader
DataReader* reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == reader)
{
    // Error
    return;
}
```

XML

6.29.4 Large Data Rates

When the amount of data exchanged between a *Publisher* and a *Subscriber* is large, some extra configuration may be required to compensate for side effects on the network and CPU load. This large amount of data can be a result of the data types being large, a high message rate, or a combination of both.

In this scenario, several approaches can be considered depending on the problem:

- For the cases in which the data samples are large (in the order of MB) such as transmitting raw video frames, point clouds, images, etc. between different hosts, TCP based communications may yield better reception rates with lower message loss, specially in the cases where a best effort transport layer is more susceptible to data loss, such as WiFi. To tackle these cases, *Large Data mode and Fast DDS over TCP* documents several ways to configure Fast DDS to communicate over TCP.
- Network packages could be dropped because the transmitted amount of data fills the socket buffer before it can be processed. The solution is to *increase the buffers size*.
- It is also possible to limit the rate at which the Publisher sends data using *Flow Controllers*, in order to limit the effect of message bursts, and avoid to flood the Subscribers faster than they can process the messages.
- On `RELIABLE_RELIABILITY_QOS` mode, the overall message rate can be affected due to the retransmission of lost packets. Selecting the Heartbeat period allows to tune between increased meta traffic or faster response to lost packets. See *Tuning Heartbeat Period*.
- Also on `RELIABLE_RELIABILITY_QOS` mode, with high message rates, the history of the *DataWriter* can be filled up, blocking the publication of new messages. A *non-strict reliable mode* can be configured to avoid this blocking, at the cost of potentially losing some messages on some of the Subscribers.

Warning: *eProsima Fast DDS* defines a conservative default message size of 64kB, which roughly corresponds to TCP and UDP payload sizes. If the topic data is bigger, it will automatically be fragmented into several transport packets.

Warning: The loss of a fragment means the loss of the entire message. This has the most impact on `BEST_EFFORT_RELIABILITY_QOS` mode, where the message loss probability increases with the number of fragments

Increasing socket buffers size

In high rate scenarios or large data scenarios, network packages can be dropped because the transmitted amount of data fills the socket buffer before it can be processed. Using `RELIABLE_RELIABILITY_QOS` mode, *Fast DDS* will try to recover lost samples, but with the penalty of retransmission. With `BEST_EFFORT_RELIABILITY_QOS` mode, samples will be definitely lost.

By default *eProsima Fast DDS* creates socket buffers with the system default size. However, these sizes can be modified using the *DomainParticipantQos*, as shown in the example below.

C++

```
DomainParticipantQos participant_qos;

// Increase the sending buffer size
participant_qos.transport().send_socket_buffer_size = 1048576;

// Increase the receiving buffer size
participant_qos.transport().listen_socket_buffer_size = 4194304;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_xml_profile_qos_socketbuffers">
        <rtps>
            <sendSocketBufferSize>1048576</sendSocketBufferSize>
            <listenSocketBufferSize>4194304</listenSocketBufferSize>
        </rtps>
    </participant>
</profiles>
```

Finding out system maximum values

Operating systems set a maximum value for socket buffer sizes. If the buffer sizes are tuned with DomainParticipantQos, the values set cannot exceed the maximum value of the system.

Linux

The maximum buffer size values can be retrieved with the command `sysctl`. For socket buffers used to send data, use the following command:

```
$> sudo sysctl -a | grep net.core.wmem_max
net.core.wmem_max = 1048576
```

For socket buffers used to receive data the command is:

```
$> sudo sysctl -a | grep net.core.rmem_max
net.core.rmem_max = 4194304
```

However, these maximum values are also configurable and can be increased if needed. The following command increases the maximum buffer size of sending sockets:

```
$> sudo sysctl -w net.core.wmem_max=12582912
```

For receiving sockets, the command is:

```
$> sudo sysctl -w net.core.rmem_max=12582912
```

Windows

The following command changes the maximum buffer size of sending sockets:

```
C:\> reg add HKLM\SYSTEM\CurrentControlSet\services\AFD\Parameters /v DefaultSendWindow /
↪t REG_DWORD /d 12582912
```

For receiving sockets, the command is:

```
C:\> reg add HKLM\SYSTEM\CurrentControlSet\services\AFD\Parameters /v␣
↪DefaultReceiveWindow /t REG_DWORD /d 12582912
```

Increasing the Transmit Queue Length of an interface (Linux only)

The Transmit Queue Length (`txqueuelen`) is a TCP/UDP/IP stack network interface value. This value sets the number of packets allowed per kernel transmit queue of a network interface device. By default, the `txqueuelen` value for Ethernet interfaces is set to `1000` in Linux. This value is adequate for most Gigabit network devices. However, in some specific cases, the `txqueuelen` setting should be increased to avoid overflows that drop packets. Similarly, choosing a value that is too large can cause added overhead resulting in higher network latencies.

Note that this information only applies to the *sending* side, and not the *receiving* side. Also increasing the `txqueuelen` should go together with increasing the buffer sizes of the UDP and/or TCP buffers. (this must be applied for both the *sending* and *receiving* sides).

The settings for a specific network adapter can be viewed using the one of the following commands:

`ip`

```
ip link show ${interface}
```

`ifconfig`

```
ifconfig ${interface}
```

This will display the configuration of the adapter, and among the parameters the `txqueuelen`. This parameter can be a value between 1000 and 20000.

Important: If the `ip` command is used, the Transmit Queue Length parameter is called `qlen`.

The `txqueuelen` can be modified for the current session using either the `ifconfig` or `ip` commands. However, take into account that after rebooting the default values will be configured again.

`ip`

```
ip link set txqueuelen ${value} dev ${interface}
```

`ifconfig`

```
ifconfig ${interface} txqueuelen ${size}
```

Flow Controllers

eProsima Fast DDS provides a mechanism to limit the rate at which the data is sent by a DataWriter. These controllers should be registered on the creation of the DomainParticipant using *FlowControllersQos*, and then referenced on the creation of the DataWriter using *PublishModeQosPolicy*.

A new thread is spawned the first time a flow controller is referenced by an asynchronous DataWriter. This thread will be responsible for arbitrating the network output of the samples being transmitted by all the DataWriters referencing the same flow controller.

Flow controllers should be given a name so they can later on be referenced by the DataWriters. A default, unlimited, *FIFO* flow controller is always available with name *FASTDDS_FLOW_CONTROLLER_DEFAULT*.

Scheduling policy

There are different kinds of flow controllers, depending on the scheduling policy used. All of them will limit the number of bytes sent to the network to no more than *max_bytes_per_period* bytes during *period_ms* milliseconds. They only differ in the way they decide the order in which the samples are sent.

- *FIFO* will output samples on a first come, first served order.
- *ROUND_ROBIN* will output one sample from each DataWriter in circular order.
- *HIGH_PRIORITY* will output samples from DataWriters with the highest priority first. The priority of a DataWriter is configured using property `fastdds.sfc.priority`. Allowed values are from -10 (highest priority) to 10 (lowest priority). If the property is not present, it will be set to the lowest priority. Samples for DataWriters with the same priority are handled with FIFO order.
- *PRIORITY_WITH_RESERVATION* works as the previous one, but allows the DataWriters to reserve part of the output bandwidth. This is done with the property `fastdds.sfc.bandwidth_reservation`. Allowed values are from 0 to 100, and express a percentage of the total flow controller limit. If the property is not present, it will be set to 0 (no bandwidth is reserved for the DataWriter). After the reserved bandwidth has been consumed, the rest of the samples will be handled with the rules of *HIGH_PRIORITY*.

Example configuration

C++

```
// Limit to 300kb per second.
static const char* flow_controller_name = "example_flow_controller";
auto flow_control_300k_per_sec = std::make_shared
↪<eprosima::fastdds::rtps::FlowControllerDescriptor>();
flow_control_300k_per_sec->name = flow_controller_name;
flow_control_300k_per_sec->scheduler =␣
↪eprosima::fastdds::rtps::FlowControllerSchedulerPolicy::FIFO;
flow_control_300k_per_sec->max_bytes_per_period = 300 * 1000;
flow_control_300k_per_sec->period_ms = 1000;

// Register flow controller on participant
DomainParticipantQos participant_qos;
participant_qos.flow_controllers().push_back(flow_control_300k_per_sec);

// .... create participant and publisher

// Link writer to the registered flow controller.
// Note that ASYNCHRONOUS_PUBLISH_MODE must be used
DataWriterQos qos;
qos.publish_mode().kind = ASYNCHRONOUS_PUBLISH_MODE;
qos.publish_mode().flow_controller_name = flow_controller_name;
```

XML

There is currently no way of configuring flow controllers with XML. This will be added in future releases of the product

Warning: Specifying a flow controller with a size smaller than the transport buffer size can cause the messages to never be sent.

Tuning Heartbeat Period

On *RELIABLE_RELIABILITY_QOS* (*ReliabilityQosPolicy*), RTPS protocol can detect which messages have been lost and retransmit them. This mechanism is based on meta-traffic information exchanged between DataWriters and DataReaders, namely, Heartbeat and Ack/Nack messages.

A smaller Heartbeat period increases the CPU and network overhead, but speeds up the system response when a piece of data is lost. Therefore, users can customize the Heartbeat period to match their needs. This can be done with the DataWriterQos.

```
DataWriterQos qos;
qos.reliable_writer_qos().times.heartbeatPeriod.seconds = 0;
qos.reliable_writer_qos().times.heartbeatPeriod.nanosec = 500000000; //500 ms
```

Using Non-strict Reliability

When *HistoryQosPolicyKind* is set as *KEEP_ALL_HISTORY_QOS*, all samples have to be received (and acknowledged) by all subscribers before they can be overridden by the DataWriter. If the message rate is high and the network is not reliable (i.e., lots of packets get lost), the history of the DataWriter can be filled up, blocking the publication of new messages until any of the old messages is acknowledged by all subscribers.

If this strictness is not needed, *HistoryQosPolicyKind* can be set as *KEEP_LAST_HISTORY_QOS*. In this case, when the history of the DataWriter is full, the oldest message that has not been fully acknowledged yet is overridden with the new one. If any subscriber did not receive the discarded message, the publisher will send a GAP message to inform the subscriber that the message is lost forever.

Practical Examples

Example: Sending a large file

Consider the following scenario:

- A Publisher needs to send a file with a size of 9.9 MB.
- The Publisher and Subscriber are connected through a network with a bandwidth of 100 MB/s

With a fragment size of 64 kB, the Publisher has to send about 1100 fragments to send the whole file. A possible configuration for this scenario could be:

- Using *RELIABLE_RELIABILITY_QOS*, since a losing a single fragment would mean the loss of the complete file.
- Decreasing the heartbeat period, in order to increase the reactivity of the Publisher.
- Limiting the data rate using a *Flow Controller*, to avoid this transmission cannibalizing the whole bandwidth. A reasonable rate for this application could be 5 MB/s, which represents only 5% of the total bandwidth.

Note: Using *Shared Memory Transport* the only limit to the fragment size is the available memory. Therefore, all fragmentation can be avoided in SHM by increasing the size of the shared buffers.

Example: Video streaming

In this scenario, the application transmits a video stream between a Publisher and a Subscriber, at 50 fps. In real-time audio or video transmissions, it is usually preferred to have a high stable datarate feed, even at the cost of losing some samples. Losing one or two samples per second at 50 fps is more acceptable than freezing the video waiting for the retransmission of lost samples. Therefore, in this case *BEST_EFFORT_RELIABILITY_QOS* can be appropriate.

6.29.5 Topics with many subscribers

By default, every time a *DataWriter* publishes a data change on a *Topic*, it sends a unicast message for every *DataReader* that is subscribed to the Topic. If there are several DataReaders subscribed, it is recommendable to use multicast instead of unicast. By doing so, only one network package will be sent for each sample. This will improve both CPU and network usage.

This solution can be implemented with *UDP Transport* or *Shared Memory Transport* (SHM). SHM transport is multicast by default, but is only available between DataWriters and DataReaders on the same machine. UDP transport needs some extra configuration. The example below shows how to set a *DataReaderQos* to configure a DataReader to use a multicast transport on UDP. More information about configuring local and remote locators on endpoints can be found in *RTPSEndpointQos*.

Note: Multicast over UDP can be problematic on some scenarios, mainly WiFi and complex networks with multiple network links.

C++

```
DataReaderQos qos;

// Add new multicast locator with IP 239.255.0.4 and port 7900
eprosima::fastrtps::rtps::Locator_t new_multicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(new_multicast_locator, "239.255.0.4");
new_multicast_locator.port = 7900;
qos.endpoint().multicast_locator_list.push_back(new_multicast_locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_reader profile_name="reader_xml_conf_multicast_locators_profile">
        <multicastLocatorList>
            <locator>
                <udpv4>
                    <address>239.255.0.4</address>
                    <port>7900</port>
                </udpv4>
            </locator>
        </multicastLocatorList>
    </data_reader>
</profiles>
```

6.29.6 Real-time behavior

Real-time applications have very tight constraints on data processing times. In order to comply with these constraints, *Fast DDS* can be configured to guarantee responses within a specified time. This is achieved with the following restraints:

- Allocating all the required memory during entity initialization, so that all the data processing tasks are heap allocation free (see *Tuning allocations*).
- Returning from blocking functions if the provided timeout is reached (see *Non-blocking calls*).

This section explains how to configure *Fast DDS* to achieve this behavior.

Tuning allocations

Allocating and deallocating memory implies some non-deterministic time consuming operations. Therefore, most real-time systems need to operate in a way that all dynamic memory is allocated during the application initialization, avoiding memory management operations in the main loop.

If users provide maximum sizes for the data and collections that *Fast DDS* keeps internally, memory for these data and collections can be preallocated during entity initialization. In order to choose the correct size values, users must be aware of the topology of the whole domain. Specifically, the number of *DomainParticipants*, *DataWriters*, and *DataReaders* must be known when setting their configuration.

The following sections describe how to configure allocations to be done during the initialization of the entities. Although some examples are provided on each section as reference, there is also a *complete example use case*.

Parameters on the participant

Every DomainParticipant holds an internal collection with information about every local and remote peer DomainParticipants that has been discovered. This information includes, among other things:

- A nested collection with information of every DataWriter announced on the peer DomainParticipant.
- A nested collection with information of every DataReader announced on the peer DomainParticipant.
- Custom data configured by the user on the peer DomainParticipant, namely, *UserDataQosPolicy*, *PartitionQosPolicy*, and *PropertyPolicyQos*.

By default, these collections are fully dynamic, meaning that new memory is allocated when a new DomainParticipant, DataWriter, or DataReader is discovered. Likewise, the mentioned custom configuration data parameters have an arbitrary size. By default, the memory for these parameters is allocated when the peer DomainParticipant announces their value.

However, *DomainParticipantQos* has a member function `allocation()`, of type *ParticipantResourceLimitsQos*, that allows configuring maximum sizes for these collections and parameters, so that all the required memory can be preallocated during the initialization of the DomainParticipant.

Limiting the number of discovered entities

ParticipantResourceLimitsQos provides three data members to configure the allocation behavior of discovered entities:

- `participants` configures the allocation of the collection of discovered DomainParticipants.
- `readers` configures the allocation of the collection of DataWriters within each discovered DomainParticipant.
- `writers` configures the allocation of the collection of DataReaders within each discovered DomainParticipant.

By default, a full dynamic behavior is used. Using these members, however, it is easy to configure the collections to be preallocated during initialization, setting them to a static maximum expected value, as shown in the example below. Please, refer to *ResourceLimitedContainerConfig* for a complete description of additional configuration alternatives given by these data members.

C++

```
DomainParticipantQos qos;

// Fix the size of discovered participants to 3
// This will effectively preallocate the memory during initialization
qos.allocation().participants =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);

// Fix the size of discovered DataWriters to 1 per DomainParticipant
// Fix the size of discovered DataReaders to 3 per DomainParticipant
// This will effectively preallocate the memory during initialization
qos.allocation().writers =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(1u);
qos.allocation().readers =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_qos_entity_resource_limit">
        <rtps>
            <allocation>
                <!-- Limit to 3 participants -->
                <total_participants>
                    <initial>3</initial>
                    <maximum>3</maximum>
                    <increment>0</increment>
                </total_participants>

                <!-- Limit to 3 readers per participant -->
                <total_readers>
                    <initial>3</initial>
                    <maximum>3</maximum>
                    <increment>0</increment>
                </total_readers>

                <!-- Limit to 1 writer per participant -->
                <total_writers>
                    <initial>1</initial>
                    <maximum>1</maximum>
                    <increment>0</increment>
                </total_writers>
            </allocation>
        </rtps>
    </participant>
</profiles>
```

Warning: Configuring a collection as fixed in size effectively limits the number of peer entities that can be discovered. Once the configured limit is reached, any new entity will be ignored. In the given example, if a fourth peer DomainParticipant appears, it will not be discovered, as the collection of discovered DomainParticipants is already full.

Limiting the size of custom parameters

`data_limits` inside *ParticipantResourceLimitsQos* provides three data members to configure the allocation behavior of custom parameters:

- `max_user_data` limits the size of *UserDataQosPolicy* to the given number of octets.
- `max_properties` limits the size of *PartitionQosPolicy* to the given number of octets.
- `max_partitions` limits the size of *PropertyPolicyQos* to the given number of octets.

If these sizes are configured to something different than zero, enough memory will be allocated for them for each participant and endpoint. A value of zero implies no size limitation, and memory will be dynamically allocated as needed. By default, a full dynamic behavior is used.

`content_filter` inside *ParticipantResourceLimitsQos* provides members to configure the allocation behavior of content filter discovery information:

- `expression_initial_size` sets the preallocated size of the filter expression.
- `expression_parameters` controls the allocation behavior for the list of expression parameters. Refer to *ResourceLimitedContainerConfig* for a complete description of the alternatives. Receiving information about a content filter with more parameters than the maximum configured here, will make the filtering happen on the reader side.

C++

```
DomainParticipantQos qos;

// Fix the size of the complete user data field to 256 octets
qos.allocation().data_limits.max_user_data = 256u;
// Fix the size of the complete partitions field to 256 octets
qos.allocation().data_limits.max_partitions = 256u;
// Fix the size of the complete properties field to 512 octets
qos.allocation().data_limits.max_properties = 512u;
// Set the preallocated filter expression size to 512 characters
qos.allocation().content_filter.expression_initial_size = 512u;
// Set the maximum number of expression parameters to 4 and its allocation␣
↪configuration to fixed size
qos.allocation().content_filter.expression_parameters =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(4u);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_qos_parameter_resource_limit">
        <rtps>
            <allocation>
                <max_partitions>256</max_partitions>
                <max_user_data>256</max_user_data>
                <max_properties>512</max_properties>

                <!-- content_filter cannot be configured using XML (yet) -->
            </allocation>
        </rtps>
    </participant>
</profiles>
```

Warning: If the data fields announced by the remote peer do not fit on the preallocated memory, an error will be triggered during the processing of the announcement message. This usually means that the discovery messages of a remote peer with too large data fields will be discarded, i.e., peers with too large data fields will not be discovered.

Parameters on the DataWriter

Every DataWriter holds internal collections with information about every DataReader to which it matches. By default, these collections are fully dynamic, meaning that new memory is allocated when a new DataReader is matched. However, *DataWriterQos* has a data member *writer_resource_limits()*, of type *WriterResourceLimitsQos*, that allows configuring the memory allocation behavior on the DataWriter.

WriterResourceLimitsQos provides data members *matched_subscriber_allocation* and *reader_filters_allocation* of type *ResourceLimitedContainerConfig* that allow configuring the maximum expected size of the collection of matched DataReader, and the collection of writer side content filters, so they can be preallocated during the initialization of the DataWriter, as shown in the example below. Please, refer to

ResourceLimitedContainerConfig for a complete description of additional configuration alternatives given by these data members.

C++

```
DataWriterQos qos;

// Fix the size of matched DataReaders to 3
// This will effectively preallocate the memory during initialization
qos.writer_resource_limits().matched_subscriber_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);
// Fix the size of writer side content filters to 1
// This will effectively preallocate the memory during initialization
qos.writer_resource_limits().reader_filters_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(1u);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_writer profile_name="writer_profile_qos_resource_limit">
        <!-- Limit to 3 matching readers -->
        <matchedSubscribersAllocation>
            <initial>3</initial>
            <maximum>3</maximum>
            <increment>0</increment>
        </matchedSubscribersAllocation>

        <!-- reader_filters_allocation cannot be configured using XML (yet) -->
    </data_writer>
</profiles>
```

Warning: Configuring the collection of matched DataReaders as fixed in size effectively limits the number of DataReaders to be matched. Once the configured limit is reached, any new DataReader will be ignored. In the given example, if a fourth (potentially matching) DataReader appears, it will not be matched, as the collection is already full.

Parameters on the DataReader

Every DataReader holds an internal collection with information about every *ReaderResourceLimitsQos* to which it matches. By default, this collection is fully dynamic, meaning that new memory is allocated when a new DataWriter is matched. However, *DataReaderQos* has a data member `reader_resource_limits()`, of type *ReaderResourceLimitsQos*, that allows configuring the memory allocation behavior on the DataReader.

ReaderResourceLimitsQos provides a data member `matched_publisher_allocation` of type *ResourceLimitedContainerConfig* that allows configuring the maximum expected size of the collection of matched DataWriters, so that it can be preallocated during the initialization of the DataReader, as shown in the example below. Please, refer to *ResourceLimitedContainerConfig* for a complete description of additional configuration alternatives given by this data

member.

C++

```
DataReaderQos qos;

// Fix the size of matched DataWriters to 1
// This will effectively preallocate the memory during initialization
qos.reader_resource_limits().matched_publisher_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(1u);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_reader profile_name="reader_profile_qos_resource_limit">
        <!-- Limit to 1 matching writer -->
        <matchedPublishersAllocation>
            <initial>1</initial>
            <maximum>1</maximum>
            <increment>0</increment>
        </matchedPublishersAllocation>
    </data_reader>
</profiles>
```

Warning: Configuring the collection of matched DataWriters as fixed in size effectively limits the number of DataWriters to be matched. Once the configured limit is reached, any new DataWriter will be ignored. In the given example, if a fourth (potentially matching) DataWriter appears, it will not be matched, as the collection is already full.

Full example

Given a system with the following topology:

Table 13: **Allocation tuning example topology**

Participant P1	Participant P2	Participant P3
Topic 1 publisher	Topic 1 subscriber	Topic 2 subscriber
Topic 1 subscriber		Topic 2 publisher
Topic 1 subscriber		Topic 2 subscriber

- The total number of DomainParticipants is 3.
- The maximum number of DataWriters per DomainParticipant is 1
- The maximum number of DataReaders per DomainParticipant is 2.
- The DataWriter for topic 1 matches with 3 DataReaders.
- The DataWriter for topic 2 matches with 2 DataReaders.
- All the DataReaders match exactly with 1 DataWriter.

We will assume that content filtering is not being used, and will also limit the size of the parameters:

- Maximum *PartitionQosPolicy* size: 256
- Maximum *UserDataQosPolicy* size: 256
- Maximum *PropertyPolicyQos* size: 512

The following piece of code shows the set of parameters needed for the use case depicted in this example.

C++

```
// DomainParticipant configuration
///////////////////////////////////
DomainParticipantQos participant_qos;

// We know we have 3 participants on the domain
participant_qos.allocation().participants =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);
// We know we have at most 2 readers on each participant
participant_qos.allocation().readers =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(2u);
// We know we have at most 1 writer on each participant
participant_qos.allocation().writers =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(1u);

// We know the maximum size of partition data
participant_qos.allocation().data_limits.max_partitions = 256u;
// We know the maximum size of user data
participant_qos.allocation().data_limits.max_user_data = 256u;
// We know the maximum size of properties data
participant_qos.allocation().data_limits.max_properties = 512u;

// Content filtering is not being used
participant_qos.allocation().content_filter.expression_initial_size = 0u;
participant_qos.allocation().content_filter.expression_parameters =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(0u);

// DataWriter configuration for Topic 1
////////////////////////////////////////
DataWriterQos writer1_qos;

// We know we will only have three matching subscribers, and no content filters
writer1_qos.writer_resource_limits().matched_subscriber_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);
writer1_qos.writer_resource_limits().reader_filters_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(0u);

// DataWriter configuration for Topic 2
////////////////////////////////////////
DataWriterQos writer2_qos;

// We know we will only have two matching subscribers
writer2_qos.writer_resource_limits().matched_subscriber_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(2u);
writer2_qos.writer_resource_limits().reader_filters_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(0u);

// DataReader configuration for both Topics
///////////////////////////////////////////
```

Non-blocking calls

Note: As OSX does not support necessary POSIX Real-time features, this feature is not fully supported on OSX. In that case, the feature is limited by the implementation of *std::timed_mutex* and *std::condition_variable_any*.

Several functions on the *Fast DDS API* can be blocked for an undefined period of time when operations compete for the control of a resource. The blocked function cannot continue until the operation that gained the control finishes, thus blocking the calling thread.

Real-time applications need a predictable behavior, including a predictable maximum time since a function is called until it returns control. In order to comply with this restriction, *Fast DDS* can be configured to limit the maximum blocking time of these functions. If the blocking time limit is exceeded, the requested operation is aborted and function terminated, returning the control to the caller.

This configuration needs two steps:

- Set the CMake option `-DSTRICT_REALTIME=ON` during the compilation of the application.
- Configure the maximum blocking times for the functions.

Table 14: **Fast RTPS non-blocking API**

Method	Configuration attribute	Default value
DataWriter::write()	*reliability().max_blocking_time* on *DataWriterQos*.	100 milliseconds.
DataReader::take_next_sample()	*reliability().max_blocking_time* on *DataReaderQos*.	100 milliseconds.
DataReader::read_next_sample()	*reliability().max_blocking_time* on *DataReaderQos*.	100 milliseconds.
`DataReader::wait_for_unread_message()`	The method accepts an argument with the maximum blocking time.	

6.29.7 Reduce memory usage

A great number of modern systems have tight constraints on available memory, making the reduction of memory usage to a minimum critical. Reducing memory consumption of a *Fast DDS* application can be achieved through various approaches, mainly through architectural restructuring of the application, but also by limiting the resources the middleware utilizes, and by avoiding static allocations.

Limiting Resources

The *ResourceLimitsQosPolicy* controls the resources that the service can use in order to meet the requirements imposed. It limits the amount of allocated memory per *DataWriter* or *DataReader*, as per the following parameters:

- *max_samples*: Configures the maximum number of samples that the DataWriter or DataReader can manage across all the instances associated with it, i.e. it represents the maximum samples that the middleware can store for a DataReader or DataWriter.
- *max_instances*: Configures the maximum number of instances that the DataWriter or DataReader can manage.
- *max_samples_per_instance*: Controls the maximum number of samples within an instance that the DataWriter or DataReader can manage.
- *allocated_samples*: States the number of samples that will be allocated on initialization.

All these parameters may be lowered as much as needed to reduce memory consumption, limit the resources to the application's needs. Below is an example of a configuration for the minimum resource limits possible.

Warning:

- The value of *max_samples* must be higher or equal to the value of *max_samples_per_instance*.
- The value established for the *HistoryQosPolicy depth* must be lower or equal to the value stated for *max_samples_per_instance*.

C++

```
ResourceLimitsQosPolicy resource_limits;

// The ResourceLimitsQosPolicy is default constructed with max_samples = 5000
// Change max_samples to the minimum
resource_limits.max_samples = 1;

// The ResourceLimitsQosPolicy is default constructed with max_instances = 10
// Change max_instances to the minimum
resource_limits.max_instances = 1;

// The ResourceLimitsQosPolicy is default constructed with max_samples_per_instance =␣
↪400
// Change max_samples_per_instance to the minimum
resource_limits.max_samples_per_instance = 1;

// The ResourceLimitsQosPolicy is default constructed with allocated_samples = 100
// No allocated samples
resource_limits.allocated_samples = 0;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_writer profile_name="data_writer_min_samples">
        <topic>
            <historyQos>
                <kind>KEEP_LAST</kind>
                <depth>1</depth>
            </historyQos>
            <resourceLimitsQos>
                <max_samples>1</max_samples>
                <max_instances>1</max_instances>
                <max_samples_per_instance>1</max_samples_per_instance>
                <allocated_samples>0</allocated_samples>
            </resourceLimitsQos>
        </topic>
    </data_writer>

    <data_reader profile_name="data_reader_min_samples">
        <topic>
            <historyQos>
                <kind>KEEP_LAST</kind>
                <depth>1</depth>
            </historyQos>
            <resourceLimitsQos>
                <max_samples>1</max_samples>
                <max_instances>1</max_instances>
                <max_samples_per_instance>1</max_samples_per_instance>
                <allocated_samples>0</allocated_samples>
            </resourceLimitsQos>
        </topic>
    </data_reader>
</profiles>
```

Set Dynamic Allocation

By default *MemoryManagementPolicy* is set to *PREALLOCATED_WITH_REALLOC_MEMORY_MODE*, meaning that the amount of memory required by the configured *ResourceLimitsQosPolicy* will be allocated at initialization. If some more memory has to be allocated at run time, it is reallocated.

Using the dynamic settings of the *RTPSEndpointQos* will prevent unnecessary allocations. Lowest footprint is achieved with *DYNAMIC_RESERVE_MEMORY_MODE* at the cost of higher allocation counts, in this mode memory is allocated when needed and freed as soon as it stops being used. For higher determinism at a small memory cost the *DYNAMIC_REUSABLE_MEMORY_MODE* option is available, this option is similar but once more memory is allocated it is not freed and is reused for future messages.

C++

```
RTPSEndpointQos endpoint;
endpoint.history_memory_policy = eprosima::fastrtps::rtps::DYNAMIC_REUSABLE_MEMORY_
↪MODE;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <data_writer profile_name="data_writer_low_memory">
        <!-- ...  -->
        <historyMemoryPolicy>DYNAMIC_REUSABLE</historyMemoryPolicy>
    </data_writer>

    <data_reader profile_name="data_reader_low_memory">
        <!-- ...  -->
        <historyMemoryPolicy>DYNAMIC_REUSABLE</historyMemoryPolicy>
    </data_reader>
</profiles>
```

6.29.8 Zero-Copy communication

This section explains how to configure a Zero-Copy communication in *Fast DDS*. The Zero-Copy communication allows the transmission of data between applications without copying data in memory, saving time and resources. In order to achieve this, it uses Data-sharing delivery between the *DataWriter* and the *DataReader*, and data buffer loans between the application and *Fast DDS*.

- *Overview*
- *Getting started*
- *Writing and reading in Zero-Copy transfers*
- *Caveats*
- *Constraints*
- *Next steps*

Overview

Data-sharing delivery provides a communication channel between a DataWriter and a DataReader using shared memory. Therefore, it does not require copying the sample data to transmit it.

DataWriter sample loaning is a *Fast DDS* extension that allows the application to borrow a buffer for a sample in the publishing DataWriter. The sample can be constructed directly on this buffer, eliminating the need to copy it to the DataWriter afterwards. This prevents the copying of the data between the publishing application and the DataWriter. If Data-sharing delivery is used, the loaned data buffer will be in the shared memory itself.

Reading the data on the subscriber side can also be done with *loans from the DataReader*. The application gets the received samples as a reference to the receive queue itself. This prevents the copying of the data from the DataReader to the receiving application. Again, if Data-sharing delivery is used, the loaned data will be in the shared memory, and will indeed be the same memory buffer used in the DataWriter history.

Combining these three features, we can achieve Zero-Copy communication between the publishing application and the subscribing application.

Getting started

To enable Zero-Copy perform the following steps:

1. Define a plain and bounded type in an IDL file and generate the corresponding source code for further processing with the *Fast DDS-Gen* tool.

```
struct LoanableHelloWorld
{
    unsigned long index;
    char message[256];
};
```

2. On the DataWriter side:
    a) Create a DataWriter for the previous type. Make sure that the DataWriter does not have DataSharing disabled.
    b) Get a loan on a sample using *loan_sample()*.
    c) Write the sample using *write()*.
3. On the DataReader side:
    a) Create a DataReader for the previous type. Make sure that the DataReader does not have DataSharing disabled.
    b) Take/read samples using the available functions in the DataReader. Please refer to section *Loaning and Returning Data and SampleInfo Sequences* for further detail on how to access to loans of the received data.
    c) Return the loaned samples using *DataReader::return_loan()*.

Writing and reading in Zero-Copy transfers

The following is an example of how to publish and receive samples with DataWriters and DataReaders respectively that implement Zero-Copy.

DataWriter

When the DataWriter is created, *Fast DDS* will pre-allocate a pool of *max_samples* + *extra_samples* samples that reside in a shared memory mapped file. This pool will be used to loan samples when the *loan_sample()* function is called.

An application example of a DataWriter that supports Zero-Copy using the *Fast DDS* library is presented below. There are several points to note in the following code:

- Not disabling the *DataSharingQosPolicy*. *AUTO* kind automatically enables Zero-Copy when possible.
- The use of the *loan_sample()* function to access and modify data samples.
- The writing of data samples.

```
// CREATE THE PARTICIPANT
DomainParticipantQos pqos;
pqos.name("Participant_pub");
DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);

// REGISTER THE TYPE
TypeSupport type(new LoanableHelloWorldPubSubType());
type.register_type(participant);

// CREATE THE PUBLISHER
Publisher* publisher = participant->create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);

// CREATE THE TOPIC
Topic* topic = participant->create_topic(
    "LoanableHelloWorldTopic",
    type.get_type_name(),
    TOPIC_QOS_DEFAULT);

// CREATE THE WRITER
DataWriterQos wqos = publisher->get_default_datawriter_qos();
wqos.history().depth = 10;
wqos.durability().kind = TRANSIENT_LOCAL_DURABILITY_QOS;
// DataSharingQosPolicy has to be set to AUTO (the default) or ON to enable Zero-Copy
wqos.data_sharing().on("shared_directory");

DataWriter* writer = publisher->create_datawriter(topic, wqos);

std::cout << "LoanableHelloWorld DataWriter created." << std::endl;

int msgsent = 0;
void* sample = nullptr;
// Always call loan_sample() before writing a new sample.
// This function will provide the user with a pointer to an internal buffer where the␣
↪data type can be
```

(continues on next page)

(continued from previous page)

```
// prepared for sending.
if (ReturnCode_t::RETCODE_OK == writer->loan_sample(sample))
{
    // Modify the sample data
    LoanableHelloWorld* data = static_cast<LoanableHelloWorld*>(sample);
    data->index() = msgsent + 1;
    memcpy(data->message().data(), "LoanableHelloWorld ", 20);

    std::cout << "Sending sample (count=" << msgsent
              << ") at address " << &data << std::endl
              << "  index=" << data->index() << std::endl
              << "  message=" << data->message().data() << std::endl;

    // Write the sample.
    // After this function returns, the middleware owns the sample.
    writer->write(sample);
}
```

DataReader

The following is an application example of a DataReader that supports Zero-Copy using the *Fast DDS* library. As shown in this code snippet, the configuration in the DataReader is similar to the DataWriter. Be sure not to disable the *DataSharingQosPolicy*. *AUTO* kind automatically enables Zero-Copy when possible.

```
// CREATE THE PARTICIPANT
DomainParticipantQos pqos;
pqos.name("Participant_sub");
DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);

// REGISTER THE TYPE
TypeSupport type(new LoanableHelloWorldPubSubType());
type.register_type(participant);

// CREATE THE SUBSCRIBER
Subscriber* subscriber = participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT, nullptr);

// CREATE THE TOPIC
Topic* topic = participant->create_topic(
    "LoanableHelloWorldTopic",
    type.get_type_name(),
    TOPIC_QOS_DEFAULT);

// CREATE THE READER
DataReaderQos rqos = subscriber->get_default_datareader_qos();
rqos.history().depth = 10;
rqos.reliability().kind = RELIABLE_RELIABILITY_QOS;
rqos.durability().kind = TRANSIENT_LOCAL_DURABILITY_QOS;
// DataSharingQosPolicy has to be set to AUTO (the default) or ON to enable Zero-Copy
rqos.data_sharing().automatic();
```

(continues on next page)

(continued from previous page)

```
DataReader* reader = subscriber->create_datareader(topic, rqos, &datareader_listener);
```

Finally, the code snippet below implements the *on_data_available() DataReaderListener* callback. The key points to be noted in this function are:

- The declaration and handling of *LoanableSequence*.
- Checking *DataReader::is_sample_valid()* for verifying that the sample was not replaced. Refer to *DataReader and DataWriter history coupling* for further information.
- The use of the *DataReader::return_loan()* function to indicate to the DataReader that the application has finished accessing the sequence.

```
void on_data_available(
        eprosima::fastdds::dds::DataReader* reader) override
{
    // Declare a LoanableSequence for a data type
    FASTDDS_SEQUENCE(DataSeq, LoanableHelloWorld);

    DataSeq data;
    SampleInfoSeq infos;
    // Access to the collection of data-samples and its corresponding collection of␣
↪SampleInfo structures
    while (ReturnCode_t::RETCODE_OK == reader->take(data, infos))
    {
        // Iterate over each LoanableCollection in the SampleInfo sequence
        for (LoanableCollection::size_type i = 0; i < infos.length(); ++i)
        {
            // Check whether the DataSample contains data or is only used to communicate␣
↪of a
            // change in the instance
            if (infos[i].valid_data)
            {
                // Print the data.
                const LoanableHelloWorld& sample = data[i];

                ++samples;
                std::cout << "Sample received (count=" << samples
                          << ") at address " << &sample
                          << (reader->is_sample_valid(&sample, &infos[i]) ? " is valid"␣
↪: " was replaced" ) << std::endl
                          << "  index=" << sample.index() << std::endl
                          << "  message=" << sample.message().data() << std::endl;
            }
        }
        // Indicate to the DataReader that the application is done accessing the␣
↪collection of
        // data values and SampleInfo, obtained by some earlier invocation of read or␣
↪take on the
        // DataReader.
        reader->return_loan(data, infos);
    }
}
```

Caveats

- After calling *write()*, *Fast DDS* takes ownership of the sample and therefore it is no longer safe to make changes to that sample.
- If function *loan_sample()* is called first and the sample is never written, it is necessary to use function *discard_loan()* to return the sample to the DataWriter. If this is not done, the subsequent calls to *loan_sample()* may fail if DataWriter has no more *extra_samples* to loan.
- The current maximum supported sample size is the maximum value of an `uint32_t`.

Constraints

Although Zero-Copy can be used for one or several *Fast DDS* application processes running on the same machine, it has some constraints:

- Only plain types are supported.

  A plain type is a type whose CDR representation matches its in-memory representation. This requirement avoids the copy between the CDR buffer and the user buffer because the data representation is the same. Consequently, only *primitive types* (except `string`), arrays of these primitive types, and structures with FINAL *extensibility* and members of these primitive types, are considered to be plain (Fast DDS also provides an API to check if a defined type is plain: *TypeSupport::is_plain()*).
- Constraints for *datasharing delivery* also apply.

Note: Zero-Copy transfer support for non-plain types may be implemented in future releases of *Fast DDS*.

Next steps

The *eProsima Fast DDS* Github repository contains the complete example discussed in this section, as well as multiple other examples for different use cases. The example implementing Zero-Copy transfers can be found here.

6.29.9 Unique network flows

This section explains which APIs should be used on Fast DDS in order to have unique network flows on specific topics.

- *Background*
- *Identifying a flow*
- *Requesting unique flows*
- *Example*

Background

IP networking is the pre-dominant inter-networking technology used nowadays. Ethernet, WiFi, 4G/5G telecommunication, all of them rely on IP networking.

Streams of IP packets from a given source to destination are called *packet flows* or simply *flows*. The network QoS of a flow can be configured when using certain networking equipment (routers, switches). Such pieces of equipment typically support 3GPP/5QI protocols to assign certain Network QoS parameters to specific flows. Requesting a specific Network QoS is usually done on the endpoint sending the data, as it is the one that usually haves complete information about the network flow.

Applications may need to use specific Network QoS parameters on different topics.

This means an application should be able to:

a) Identify the flows being used in the communications, so they can correctly configure the networking equipment.

b) Use specific flows on selected topics.

Identifying a flow

The *5-tuple* is a traditional unique identifier for flows on 3GPP enabled equipment. The 5-tuple consists of five parameters: source IP address, source port, destination IP address, destination port, and the transport protocol (example, TCP/UDP).

Definitions

Network flow: A tuple of networking resources selected by the middleware for transmission of messages from a DataWriter to a DataReader, namely:

- Transport protocol: UDP or TCP
- Transport port
- Internet protocol: IPv4 or IPv6
- IP address

Network Flow Endpoint (NFE): The portion of a network flow specific to the DataWriter or the DataReader. In other words, each network flow has two NFEs; one for the DataWriter, and the other for the DataReader.

APIs

Fast DDS provides the APIs needed to get the list of NFEs used by a given DataWriter or a DataReader.

- On the DataWriter, `get_sending_locators()` allows the application to obtain the list of locators from which the writer may send data.
- On the DataReader, `get_listening_locators()` allows the application to obtain the list of locators on which the reader is listening.

Requesting unique flows

A unique flow can be created by ensuring that at least one of the two NFEs are unique. On Fast DDS, there are two ways to select unique listening locators on the DataReader:

- The application can specify on which locators the DataReader should be listening. This is done using *RTPSEndpointQos* on the *DataReaderQos*. In this case it is the responsibility of the application to ensure the uniqueness of the locators used.
- The application can request the reader to be created with unique listening locators. This is done using a *PropertyPolicyQos* including the property `"fastdds.unique_network_flows"`. In this case, the reader will listen on a unique port outside the range of ports typically used by RTPS.

Example

The following snippet demonstrates all the APIs described on this page:

```
// Create the DataWriter
DataWriter* writer = publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == writer)
{
    // Error
    return;
}

// Create DataReader with unique flows
DataReaderQos drqos = DATAREADER_QOS_DEFAULT;
drqos.properties().properties().emplace_back("fastdds.unique_network_flows", "");
DataReader* reader = subscriber->create_datareader(topic, drqos);

// Print locators information
eprosima::fastdds::rtps::LocatorList locators;
writer->get_sending_locators(locators);
std::cout << "Writer is sending from the following locators:" << std::endl;
for (const auto& locator : locators)
{
    std::cout << "  " << locator << std::endl;
}

reader->get_listening_locators(locators);
std::cout << "Reader is listening on the following locators:" << std::endl;
for (const Locator_t& locator : locators)
{
    std::cout << "  " << locator << std::endl;
}
```

6.29.10 Statistics module

eProsima Fast DDS Statistics Module allows the user to monitor the data being exchanged by its application. In order to use this module, the user must enable it in the monitored application, and create another application that receives the data being published by the statistics DataWriters. The user can also use for the latter the eProsima Fast DDS Statistics Backend which already implements the collection and aggregation of the data coming from the statistics topics.

- *Enable Statistics module*
- *Create monitoring application*

Enable Statistics module

The Statistics module has to be enabled both at build and runtime. On the one hand, *CMake option* `FASTDDS_STATISTICS` must be enabled when building the library (since Fast DDS v2.9.0 this CMake option is enabled by default). On the other hand, the desired statistics DataWriters should be enabled using the *Statistics Module DDS Layer*.

The statistics DataWriters can be enabled automatically using the *PropertyPolicyQos* `fastdds.statistics` and the *FASTDDS_STATISTICS* environment variable. They can also be enabled manually following the next example:

```
// Create a DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Obtain pointer to child class
eprosima::fastdds::statistics::dds::DomainParticipant* statistics_participant =
        eprosima::fastdds::statistics::dds::DomainParticipant::narrow(participant);

// Enable statistics DataWriter
if (statistics_participant->enable_statistics_
↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC,
        eprosima::fastdds::statistics::dds::STATISTICS_DATAWRITER_QOS) != ReturnCode_
↪t::RETCODE_OK)
{
    // Error
    return;
}

// Use the DomainParticipant to communicate
// (...)

// Disable statistics DataWriter
if (statistics_participant->disable_statistics_
↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC) !=
        ReturnCode_t::RETCODE_OK)
```

(continues on next page)

(continued from previous page)

```
{
    // Error
    return;
}

// Delete DomainParticipant
if (DomainParticipantFactory::get_instance()->delete_participant(participant) !=␣
↪ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Create monitoring application

Once the monitored application is publishing the collected data within the statistics topics enabled by the user, another application should be configured to subscribe to those topics. This application is a DDS standard application where the statistics DataReaders should be created. In order to create these statistics DataReaders, the user should follow the next steps:

- Using the statistics IDL provided in the public API, generate the *TopicDataTypes* with *Fast DDS-Gen*.
- Create the *DomainParticipant* and register the *TopicDataTypes* and the corresponding statistics *Topics*.
- Create the statistics DataReaders using the corresponding statistics topic.

6.29.11 Dynamic network interfaces

DDS *Simple Discovery* relies on well-known multicast addresses and ports to relay the Participant announcement messages (see *Discovery phases*). Such Participant announcement includes information about the unicast address-port pairs (a.k.a locators) where the Participant is expecting to receive incoming metatraffic data. The list with these unicast locators is automatically initialized taking into account the network interfaces that are available when the *Fast DDS* DomainParticipant is enabled. Consequently, any network interface that is added after enabling the DomainParticipant should be notified to *Fast DDS* in order to initialize an unicast locator in said network, so communication can be established over that new interface.

Dynamic network interface addition at run-time

In case that the user wants to include new network interfaces at run-time, some prerequisites have to be fulfilled. Then, once the interfaces are available, the user may notify Fast DDS so these interfaces are also used in the communication.

Prerequisites

This feature is intended to be used when *Fast DDS* automatically sets the listening unicast locators. Consequently, both *metatrafficUnicastLocatorList* and *metatrafficMulticastLocatorList* lists must be empty. These attributes are set within the *builtin* member of *wire_protocol()* contained in the *DomainParticipantQos* (please refer to *DomainParticipantQos*).

Note: Be aware of the remote locators' collections limits set within the *DomainParticipantQos* (please refer to *RemoteLocatorsAllocationAttributes*). It is recommended to use the highest number of local addresses found on all the systems belonging to the same domain.

Notify *Fast DDS*

Once a new network interface has been enabled, *Fast DDS* has to be manually notified. This is done calling *DomainParticipant::set_qos()*. The DomainParticipantQoS that is passed to the method can either change one of the mutable DomainParticipant QoS or it can simply be the current DomainParticipant QoS (obtained with *DomainParticipant::get_qos()*).

Using *DomainParticipant::set_qos()* is the reason for the previous prerequisites: once the DomainParticipant is enabled, there are several QoS policies that are immutable and cannot be changed at run-time. *WireProtocolConfigQos* where the aforementioned lists are defined is among these immutable policies.

Find below a brief snippet of how to use this feature:

```
// Create the DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// User application

// Notify Fast DDS a new network interface is available
participant->set_qos(PARTICIPANT_QOS_DEFAULT);
```

Important: This feature is still under development and only officially supported for UDPv4 Transport without whitelisting.

6.29.12 How to use eProsima DDS Record and Replay (rosbag2 and DDS)

eProsima DDS Record and Replay allows the user to continuously monitor the ROS 2 traffic in real time, and to play it back at any given time. This highly contributes to facilitating simulation of real life conditions, application testing, optimizing data analysis and general troubleshooting. rosbag2 is a ROS 2 application that can be used to capture DDS messages and store them on an SQLite database which allows inspecting and replaying said messages at a later time.

rosbag2 interactions with a native Fast DDS application

Using rosbag2 to capture traffic between ROS 2 talkers and listeners is straightforward. However, recording and replaying messages sent by Fast DDS participants outside ROS 2 ecosystem requires some modifications.

Prerequisites

A Fast DDS installation, either binary or from sources is required. Fast DDS-Gen is also required for generating the examples and Fast DDS TypeSupport from the IDL file. A ROS 2 installation with the rosbag2 package is needed as well.

DDS IDL interoperability with ROS 2 messages

DDS uses IDLs to define the data model being exchanged by the applications. While ROS 2 can use IDL files to define the messages, there are some rules that these IDL files must follow so compatibility between ROS 2 and Fast DDS native applications can be achieved. Specifically, the type definition must be nested inside the type module name and then the generator to be used. For ROS 2 messages, the generator would be `msg`, whereas in this case, the `idl` generator must be used. Assuming that the type module name selected is `fastdds_record_typesupport` the following `HelloWorld.idl` file could be defined. This IDL file will be the one used in the following steps.

```
module fastdds_record_typesupport
{
    module idl
    {
        struct HelloWorld
        {
            unsigned long index;
            string message;
        };
    };
};
```

By default, rosbag2 can only recognize those Topics which types ROS 2 has already defined in its different TypeSupport libraries. Therefore, a new ROS 2 TypeSupport module library generated with the previously defined types must be created, so rosbag2 would be able to parse the message contents coming from the Fast DDS application. First, the new ROS 2 TypeSupport package should be created. Follow the instructions below, after having sourced your ROS 2 installation:

```
ros2 pkg create --build-type ament_cmake fastdds_record_typesupport
```

This command will create a new ROS 2 package named `fastdds_record_typesupport` with the following folder structure:

```
.
└── fastdds_record_typesupport
    ├── include
    │   └── fastdds_record_typesupport
    ├── src
    ├── CMakeLists.txt
    └── package.xml
```

ROS 2 TypeSupport code generators expect IDL files inside their own idl folder, so the final folder structure would be like this:

```
.
└── fastdds_record_typesupport
    ├── idl
    │   └── HelloWorld.idl
    ├── include
    │   └── fastdds_record_typesupport
    ├── src
    ├── CMakeLists.txt
    └── package.xml
```

In order to generate the TypeSupport interfaces required, the CMakeLists.txt file should be modified accordingly so the ROS 2 TypeSupport generator is called. Please add the following lines to the CMakeLists.txt file before calling `ament_package()`:

```
find_package(rosidl_default_generators REQUIRED)

set(idl_files
  "idl/HelloWorld.idl"
)

rosidl_generate_interfaces(${PROJECT_NAME}
  ${idl_files}
)
```

Similarly, the `package.xml` file should be modified adding the ROS 2 TypeSupport generator dependency. Add the following lines to the `package.xml` file after the `buildtool_depend` tags:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

The last step would be to build the package. Run the following command within the `fastdds_record_typesupport` folder:

```
RMW_IMPLEMENTATION=rmw_fastrtps_cpp colcon build
```

The build process will create inside the install folder a new ROS 2 overlay with all the required libraries and scripts for the ROS 2 applications to use te type defined in the IDL file.

Fast DDS Application tuning

ROS 2 adds special tokens to the topic names depending on the ROS 2 subsystem the topic belongs to. More information on this topic can be found on ROS 2 design documentation .

Using the same IDL file defined earlier, Fast DDS-Gen can generate the required code to handle the new type in Fast DDS. The changes required in the Fast DDS application so rosbag2 can communicate with it are going to be illustrated via the Publisher/Subscriber example generated automatically from an IDL using Fast DDS-Gen. An in-depth guide to Fast DDS-Gen can be found here.

In the case of plain topics, the namespace "rt/" is added by ROS 2 to the DDS topic name. DataType names for ROS 2 generated types are structured concatenating the modules names. For the IDL being used in this example the data type name would be "fastdds_record_typesupport::idl::HelloWorld".

Create a new workspace different from the ROS 2 one used previously. Copy inside the same IDL file and run Fast DDS-Gen to generate the TypeSupport and the example source files:

```
mkdir HelloWorldExample
cd HelloWorldExample
cp <PATH_TO_ROS2_WORKSPACE>/fastdds_record_typesupport/idl/HelloWorld.idl .
fastddsgen -example CMake -typeros2 HelloWorld.idl
```

This command will populate the current folder with the required header and source files to build the TypeSupport, and the Publisher and Subscriber applications.

```
└── HelloWorldExample
    ├── CMakeLists.txt
    ├── HelloWorld.cxx
    ├── HelloWorld.h
    ├── HelloWorld.idl
    ├── HelloWorldPublisher.cxx
    ├── HelloWorldPublisher.h
    ├── HelloWorldPubSubMain.cxx
    ├── HelloWorldPubSubTypes.cxx
    ├── HelloWorldPubSubTypes.h
    ├── HelloWorldSubscriber.cxx
    └── HelloWorldSubscriber.h
```

The Fast DDS-Gen example should be modified taking into account the topic and type name mangling applied by ROS 2 so communication can be established with rosbag2. Having used the `-typeros2` Fast DDS-Gen option when generating the TypeSupport, the generated type name would already include the ROS 2 naming rule mangling. However, the topic name must be modified manually both in the Publisher and Subscriber applications. Look for the `create_topic` command in both the `HelloWorldPublisher.cxx` and the `HelloWorldSubscriber.cxx` files and modify the topic name:

```
topic_ = participant_->create_topic(
    "rt/HelloWorldTopic",
    type_.get_type_name(),
    TOPIC_QOS_DEFAULT);
if (topic_ == nullptr)
{
    return false;
}
```

To build this example run the following commands:

```
mkdir build && cd build
cmake ..
make
```

This will create a HelloWorld binary file inside the build directory that can be used to launch both the Publisher and the Subscriber applications. Run each application in a terminal and confirm that the communication is established.

```
./HelloWorld publisher|subscriber
```

eProsima DDS Record and Replay

In order to use the generated ROS 2 TypeSupport package, the ROS 2 workspace should be sourced besides the ROS 2 installation. This allows rosbag2 to record the data types used in this example. To start recording the traffic being exchanged between the Publisher/Subscriber applications the corresponding ROS 2 Topic name has to be passed to rosbag2 (not to be mistaken with the DDS Topic name). Remember also to ensure that Fast DDS is the ROS 2 middleware being used by setting the environment variable `RMW_IMPLEMENTATION`.

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
source <PATH_TO_ROS2_WORKSPACE>/fastdds_record_typesupport/install/setup.bash
ros2 bag record /HelloWorldTopic
```

Having the Publisher application running already, the following rosbag2 log discovery info would be shown:

```
[INFO] [1644320308.422161532] [rosbag2_recorder]: Subscribed to topic '/HelloWorldTopic'
[INFO] [1644320308.422292205] [rosbag2_recorder]: All requested topics are subscribed.␣
↪Stopping discovery...
```

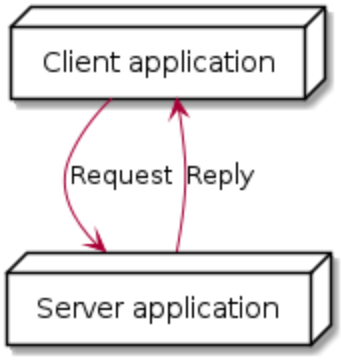
rosbag2 will proceed to create a folder named `rosbag2_<DATE>` with an SQLite database inside (`db3` extension) where the received messages will be recorded. Within the folder a `YAML` file provides metadata information about the record: type and topic name, number of messages recorded, record duration, etc. The path to this database file can be used to replay the recorded messages. Having the Subscriber application running, the previously recorded traffic will be replayed. After stopping the rosbag2 application, rerun it in replay mode running the following command. The recorded messages will be published by rosbag2 at their original publishing rate and the Subscriber application will receive them:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
source <PATH_TO_ROS2_WORKSPACE>/fastdds_record_typesupport/install/setup.bash
ros2 bag play <path-to-db-file>
```

6.29.13 Request-Reply communication

This section explains how to configure a Request-Reply communication in *Fast DDS* between two applications. A *Client application* will send a *Request* to the *Server application* and this one, after processing the request, will send a *Reply* to the *Client application*.

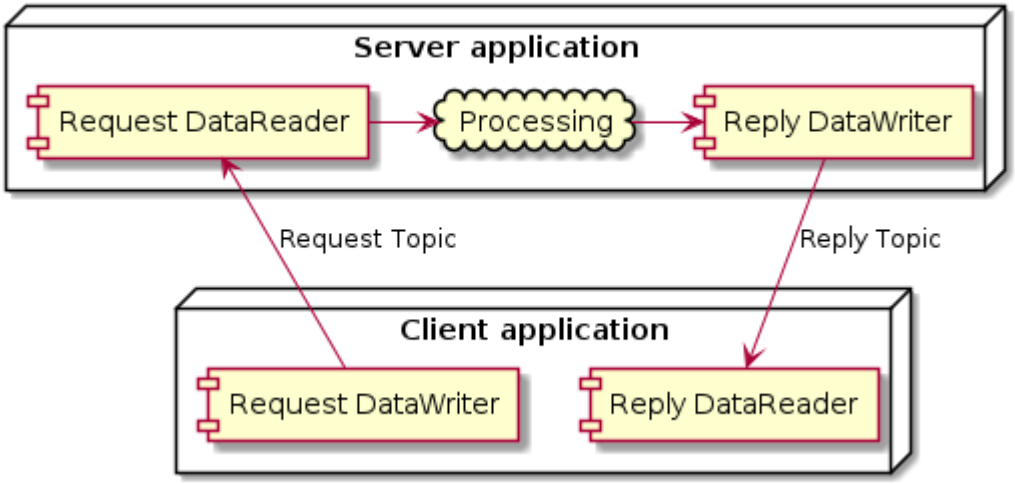
- *Overview*
- *Getting started*
- *Sending the request and storing the assigned identifier*
- *Receiving the request and sending a reply associated with it*



- *Identifying the reply for the client*

Overview

This kind of communication involves in the DDS paradigm the usage of two *Topics*: one for sending requests (Request Topic) and the other one for sending replies (Reply Topic). For managing these *Topics* four DDS entities are involved: a DataReader and a DataWriter for each *Topic*. The DDS communication schema will be:



The key for making *Request-Reply* work is relating the Request with the Reply in the client's side. *Fast DDS* API provides `SampleIdentity` to achieve this.

A full example can be found in Fast DDS repository.

Getting started

For *Request-Reply* communication perform the following steps:

1. Define two structures in an IDL file. One structure will be used as Request Topic's data type and the other one as Reply Topic's data type.

```
enum OperationType
{
    ADDITION,
    SUBSTRACTION,
    MULTIPLICATION,
    DIVISION
};

struct RequestType
{
    OperationType operation;
    long x;
    long y;
};

struct ReplyType
{
    long long z;
};
```

2. In the client application, create a DataWriter for the request and a DataReader for the reply.

```
participant->register_type(request_type);
participant->register_type(reply_type);

Topic* request_topic = participant->create_topic("CalculatorRequest",
                request_type.get_type_name(), TOPIC_QOS_DEFAULT);

Topic* reply_topic = participant->create_topic("CalculatorReply", reply_
↪type.get_type_name(), TOPIC_QOS_DEFAULT);

DataWriter* request_writer = publisher->create_datawriter(request_topic,␣
↪DATAWRITER_QOS_DEFAULT);

DataReader* reply_reader = subscriber->create_datareader(reply_topic,␣
↪DATAREADER_QOS_DEFAULT, &listener);
```

3. In the server application, create a DataWriter for the reply and a DataReader for the request.

```
participant->register_type(request_type);
participant->register_type(reply_type);

Topic* request_topic = participant->create_topic("CalculatorRequest",
                request_type.get_type_name(), TOPIC_QOS_DEFAULT);

Topic* reply_topic = participant->create_topic("CalculatorReply", reply_
↪type.get_type_name(), TOPIC_QOS_DEFAULT);
```

(continues on next page)

(continued from previous page)

```
DataWriter* reply_writer = publisher->create_datawriter(reply_topic,␣
↪DATAWRITER_QOS_DEFAULT);

DataReader* request_reader = subscriber->create_datareader(request_topic,␣
↪DATAREADER_QOS_DEFAULT, &listener);
```

Sending the request and storing the assigned identifier

For sending the request, the client application should retrieve and store the internal identifier assigned to the published sample. Therefore the sample should be published using the overloaded *write()* function which second argument is a reference to a *WriteParams* object. The assigned identifier will be stored in the *WriteParams*'s attribute *sample_identity()*.

```
eprosima::fastrtps::rtps::SampleIdentity my_request_sample_identity;
RequestType request;

// Fill the request

// Publish request
eprosima::fastrtps::rtps::WriteParams write_params;
request_writer->write(static_cast<void*>(&request), write_params);

// Store sample identity
my_request_sample_identity = write_params.sample_identity();
```

Receiving the request and sending a reply associated with it

When the server application receives the request (for example through *on_data_available()*), it has to retrieve the request's identifier using *sample_identity*.

```
    void on_data_available(
            eprosima::fastdds::dds::DataReader* reader) override
    {
        RequestType request;
        eprosima::fastdds::dds::SampleInfo sample_info;

        reader->take_next_sample(&request, &sample_info);

        if (eprosima::fastdds::dds::InstanceStateKind::ALIVE_INSTANCE_STATE == sample_
↪info.instance_state)
        {
            // Store the request identity.
            eprosima::fastrtps::rtps::SampleIdentity client_request_identity = sample_
↪info.sample_identity;
        }
    }
```

After processing the request, the server should send the reply to the client with the related request attached. This is done assigning the stored identifier in *related_sample_identity()*.

```
ReplyType reply;

// Fill reply

// Send reply associating it with the client request.
eprosima::fastrtps::rtps::WriteParams write_params;
write_params.related_sample_identity() = client_request_identity;
reply_writer->write(reinterpret_cast<void*>(&reply), write_params);
```

Identifying the reply for the client

When the client application receives a reply (for example through *on_data_available()*), the client application should identify if the received reply is the one expected for its request. For this the client application has to compare the stored *SampleIdentity* with the incoming *related_sample_identity*.

```
    void on_data_available(
            eprosima::fastdds::dds::DataReader* reader) override
    {
        ReplyType reply;
        eprosima::fastdds::dds::SampleInfo sample_info;

        reader->take_next_sample(&reply, &sample_info);

        if (eprosima::fastdds::dds::InstanceStateKind::ALIVE_INSTANCE_STATE == sample_
↪info.instance_state)
        {
            if (sample_info.related_sample_identity == my_request_sample_identity)
            {
                // Work to do
            }
        }
    }
```

6.30 ROS 2 using Fast DDS middleware

Fast DDS is the default middleware implementation in the Open Source Robotic Fundation (OSRF) Robot Operating System ROS 2 in every long term (LTS) releases and most of the non-LTS releases.

ROS 2 is a state-of-the-art software for robot engineering which consists of a set of free software libraries and tools for building robot applications. This section presents some use cases and shows how to take full advantage of Fast DDS wide set of capabilities in a ROS 2 project.

The interface between the ROS 2 stack and *Fast DDS* is provided by a ROS 2 package rmw_fastrtps. This package is available in all ROS 2 distributions, both from binaries and from sources. `rmw_fastrtps` actually provides not one but two different ROS 2 middleware implementations, both of them using *Fast DDS* as middleware layer: `rmw_fastrtps_cpp` and `rmw_fastrtps_dynamic_cpp`. The main difference between the two is that `rmw_fastrtps_dynamic_cpp` uses introspection type support at run time to decide on the serialization/deserialization mechanism, while `rmw_fastrtps_cpp` uses its own type support, which generates the mapping for each message type at build time. The default ROS 2 RMW implementation until *Foxy* is `rmw_fastrtps_cpp`. For *Galactic* the environment variable `RMW_IMPLEMENTATION` has to be set to select `rmw_fastrtps_cpp` in order to use *Fast DDS* as the

middleware layer. This environment variable can also be used to select the `rmw_fastrtps_dynamic_cpp` implementation:

1. Exporting `RMW_IMPLEMENTATION` environment variable:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
```

or

```
export RMW_IMPLEMENTATION=rmw_fastrtps_dynamic_cpp
```

2. When launching your ROS 2 application:

```
RMW_IMPLEMENTATION=rmw_fastrtps_cpp ros2 run <package> <application>
```

or

```
RMW_IMPLEMENTATION=rmw_fastrtps_dynamic_cpp ros2 run <package> <application>
```

Note: Since *Galactic* you may have to install the `rmw_fastrtps_cpp` package:

```
sudo apt install ros-galactic-rmw-fastrtps-cpp
```

6.30.1 Configuring *Fast DDS* in ROS 2

ROS 2 only allows for the configuration of certain middleware QoS (see ROS 2 QoS policies). However, *rmw_fastrtps* offers extended configuration capabilities to take full advantage of the features in *Fast DDS*. This section describes how to specify this extended configuration.

- *Changing publication mode*
- *XML configuration*
  - *XML configuration file location*
  - *Applying different profiles to different entities*
- *Example*

Changing publication mode

rmw_fastrtps in ROS 2 uses asynchronous publication by default. This can be easily changed setting the environment variable `RMW_FASTRTPS_PUBLICATION_MODE` to one of the following allowed values:

- **ASYNCHRONOUS**: asynchronous publication mode. Setting this mode implies that when the publisher invokes the write operation, the data is copied into a queue, a background thread (asynchronous thread) is notified about the addition to the queue, and control of the thread is returned to the user before the data is actually sent. The background thread is in charge of consuming the queue and sending the data to every matched reader.
- **SYNCHRONOUS**: synchronous publication mode. Setting this mode implies that the data is sent directly within the context of the user thread. This entails that any blocking call occurring during the write operation would block the user thread, thus preventing the application from continuing its operation. It is important to note that

this mode typically yields higher throughput rates at lower latencies, since there is no notification nor context switching between threads.

- **AUTO**: let Fast DDS select the publication mode. This implies using the publication mode set in the *XML file*, or otherwise, the default value set in Fast DDS (see *PublishModeQosPolicy*).

rmw_fastrtps defines two configurable parameters in addition to ROS 2 QoS policies. Said parameters, and their default values under ROS 2, are:

Parameter	Description	Default ROS 2 value
MemoryManagementPolicy	*Fast DDS* preallocates memory for the publisher and subscriber histories. When those histories fill up, a reallocation occurs to reserve more memory.	*PREALLOCATED_WITH_REALLOC_MEMOR*
PublishModeQosPolicy	User calls to publication method add the messages in a queue that is managed in a different thread, meaning that the user thread is available right after the call to send data.	*ASYNCHRONOUS_PUBLISH_MODE*

XML configuration

To use specific *Fast-DDS* features within a ROS 2 application, XML configuration files can be used to configure a wide set of *QoS*. Please refer to *XML profiles* to see the whole list of configuration options available in *Fast DDS*.

When configuring *rmw_fastrtps* using XML files, there are certain points that have to be taken into account:

- ROS 2 QoS contained in rmw_qos_profile_t are always honored, unless set to `*_SYSTEM_DEFAULT`. In that case, XML values, or Fast DDS default values in the absences of XML ones, are applied. This means that if any QoS in `rmw_qos_profile_t` is set to something other than `*_SYSTEM_DEFAULT`, the corresponding value in the XML is ignored.
- By default, *rmw_fastrtps* overrides the values for *MemoryManagementPolicy* and *PublishModeQosPolicy*. This means that the values configured in the XML for these two parameters will be ignored. Instead, *PREALLOCATED_WITH_REALLOC_MEMORY_MODE* and *ASYNCHRONOUS_PUBLISH_MODE* are used respectively.
- The override of MemoryManagementPolicy and PublishModeQosPolicy can be avoided by setting the environment variable `RMW_FASTRTPS_USE_QOS_FROM_XML` to `1` (its default value is `0`). This will make *rmw_fastrtps* use the values defined in the XML for MemoryManagementPolicy and PublishModeQosPolicy. Bear in mind that setting this environment variable but not setting these policies in the XML results in using the default values in *Fast DDS*. These are different from the aforementioned *rmw_fastrtps* default values (see *MemoryManagementPolicy* and *PublishModeQosPolicy*).
- Setting `RMW_FASTRTPS_USE_QOS_FROM_XML` effectively overrides whatever configuration was set with `RMW_FASTRTPS_PUBLICATION_MODE`, setting the publication mode to the value specified in the XML, or to the *Fast DDS* default publication mode if none is set in the XML.

The following table summarizes which values are used or ignored according to the configured variables:

RMW_FASTRTPS_USE_QOS_FROM_XML	rmw_qos_profile_t	Fast DDS XML QoS	Fast DDS XML history memory policy and publication mode
0 (default)	Default values	Overridden by `rmw_qos_profile_t`	Overridden by *rmw_fastrtps* default value
0 (default)	Non system default	overridden by `rmw_qos_profile_t`	Overridden by *rmw_fastrtps* default value
0 (default)	System default	Used	Overridden by *rmw_fastrtps* default value
1	Default values	Overridden by `rmw_qos_profile_t`	Used
1	Non system default	Overridden by *rmw_qos_profile_t*	Used
1	System default	Used	Used

XML configuration file location

There are two possibilities for providing *Fast DDS* with XML configuration files:

- **Recommended**: Setting the location with environment variable `FASTRTPS_DEFAULT_PROFILES_FILE` to contain the path to the XML configuration file (see *Environment variables*).

```
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
```

- **Alternative**: Placing the XML file in the running application directory under the name *DEFAULT_FASTRTPS_PROFILES.xml*.

For example:

```
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run <package> <application>
```

Applying different profiles to different entities

rmw_fastrtps allows for the configuration of different entities with different QoS using the same XML file. For doing so, *rmw_fastrtps* locates profiles in the XML based on topic names.

Creating publishers/subscribers with different profiles

- To configure a publisher, define a <data_writer> profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic prepended by the node namespace (which defaults to "" if not specified), i.e. the node's namespace followed by topic name used to create the publisher. Mind that topic names always start with `/` (it is added when creating the topic if not present), and that namespace and topic name are always separated by one `/`. If such profile is not defined, *rmw_fastrtps* attempts to load the <data_writer> profile with attribute `is_default_profile="true"`.
- To configure a subscriber, define a <data_reader> profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic prepended by the node namespace (which defaults to "" if not specified), i.e. the node's namespace followed by topic name used to create the subscriber. Mind that topic names always start with `/` (it is added when creating the topic if not present), and that namespace and topic name are always

separated by one `/`. If such profile is not defined, *rmw_fastrtps* attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.

The following table presents different combinations of node namespaces and user specified topic names, as well as the resulting topic names and the suitable profile names:

User specified topic name	Node namespace	Final topic name	Profile name
chatter	DEFAULT (“”)	/chatter	/chatter
chatter	test_namespace	/test_namespace/chatter	/test_namespace/chatter
chatter	/test_namespace	/test_namespace/chatter	/test_namespace/chatter
/chatter	test_namespace	/chatter	/chatter
/chatter	/test_namespace	/chatter	/chatter

Important: Node namespaces are NOT prepended to user specified topic names starting with /, a.k.a Fully Qualified Names (FQN). For a complete description of topic name remapping please refer to Remapping Names.

Creating services with different profiles

ROS 2 services contain a subscriber for receiving requests, and a publisher to reply to them. *rmw_fastrtps* allows for configuring each of these endpoints separately in the following manner:

- To configure the request subscriber, define a `<data_reader>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. For more information on name mangling, please refer to Topic and Service name mapping to DDS. If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_reader>` profile with attribute `profile_name="service"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.
- To configure the reply publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_writer>` profile with `attribute profile_name="service"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.

Creating clients with different profiles

ROS 2 clients contain a publisher to send requests, and a subscription to receive the service’s replies. *rmw_fastrtps* allows for configuring each of these endpoints separately in the following manner:

- To configure the requests publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_writer>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.
- To configure the reply subscription, define a `<data_reader>` profile with `attribute profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_reader>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.

Creating ROS contexts and nodes

ROS *context* and *node* entities are mapped to *Fast DDS* Participant entity, according to the following table:

ROS entity	*Fast DDS* entity since *Foxy*	*Fast DDS* entity in *Eloquent & below*
Context	Participant	*Not DDS direct mapping*
Node	*Not DDS direct mapping*	Participant

This means that on *Foxy* and later releases, contexts can be configured using a `<Participant>` profile with attribute `is_default_profile="true"`. The same profile will be used in *Eloquent* and below to configure nodes.

For example, a profile for a ROS 2 context on *Foxy* and later releases would be specified as:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_ros2" is_default_profile="true">
        <rtps>
            <name>profile_for_ros2_context</name>
        </rtps>
    </participant>
</profiles>
```

Example

The following example uses the ROS 2 talker/listener demo, configuring *Fast DDS* to publish synchronously, and to have dynamically allocated publisher and subscriber histories.

1. Create a XML file *ros_example.xml* and save it in *path/to/xml/*

XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_ros2" is_default_profile="true
↪">
        <rtps>
            <name>profile_for_ros2_context</name>
        </rtps>
    </participant>

    <!-- Default publisher profile -->
    <data_writer profile_name="default publisher profile" is_default_profile="true
↪">
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </data_writer>

    <!-- Publisher profile for topic helloworld -->
    <data_writer profile_name="helloworld">
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </data_writer>

    <!-- Request subscriber profile for services -->
    <data_reader profile_name="service">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </data_reader>

    <!-- Request publisher profile for clients -->
    <data_writer profile_name="client">
        <qos>
            <publishMode>
                <kind>ASYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </data_writer>

    <!-- Request subscriber profile for server of service "add_two_ints" -->
    <data_reader profile_name="rq/add_two_intsRequest">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </data_reader>

    <!-- Reply subscriber profile for client of service "add_two_ints" -->
    <data_reader profile_name="rr/add_two_intsReply">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </data_reader>
</profiles>
```

2. Open one terminal and run:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/ros_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp talker
```

3. Open one terminal and run:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/ros_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp listener
```

6.30.2 Use ROS 2 with Fast-DDS Discovery Server

This section explains how to run some ROS 2 examples using the Discovery Servers as discovery communication. In order to get more information about the specific use of this configuration, please check the *Discovery Server Documentation* or read the *common use cases* for this configuration.

The following tutorial gathers the steps to check this functionality and learn how to use it with ROS 2.

- *Discovery Server v2*
- *Prerequisites*
- *Run the demo*
  - *Setup Discovery Server*
  - *Launch node listener*
  - *Launch node talker*
  - *Demonstrate Discovery Server execution*
- *Advance user cases*
  - *Server Redundancy*
  - *Backup Server*
  - *Discovery partitions*
- *ROS 2 Introspection*
  - *Daemon's related commands*
  - *No Daemon commands*
- *Compare Discovery Server with Simple Discovery*

The *Simple Discovery Protocol* is the standard protocol defined in the DDS standard. However, it has certain known disadvantages in some scenarios, mainly:

- It does not **Scale** efficiently, as the number of exchanged packets highly increases as new nodes are added.
- It requires **Multicasting** capabilities that may not work reliably in some scenarios, e.g. WiFi.

The **Discovery Server** provides a Client-Server Architecture that allows the nodes to connect with each other using an intermediate server. Each node will work as a *Client*, sharing its info with the *Discovery Server* and receiving the

discovery information from it. This means that the network traffic is highly reduced in big systems, and it does not require *Multicasting*.

These **Discovery Servers** can be independent, duplicated or connected with each other in order to create redundancy over the network and avoid having a *Single-Point-Of-Failure*.

Discovery Server v2

The new version **v2** of Discovery Server, available from *Fast DDS* v2.0.2, implements a new filter feature that allows to further reduce the number of discovery messages sent. This version uses the *topic* of the different nodes to decide if two nodes must be connected, or they could be left unmatched. The following schema represents the decrease of the discovery packages:

This architecture reduces the number of packages sent between the server and the different clients dramatically. In the following graph, the reduction in traffic network over the discovery phase for a RMF Clinic demo use case, is shown:

In order to use this functionality, **Fast-DDS Discovery Server** can be set using the XML configuration for Participants. Furthermore, Fast DDS provides an easier way to set a **Discovery Server** communication using the `fastdds` *CLI tool* and an *environment variable*, which are going to be used along this tutorial. For a more detailed explanation about the configuration of the Discovery Server, visit *Discovery Server Settings*.

Prerequisites

This tutorial assumes you have at least a working Foxy ROS 2 installation. In case your installation is using a Fast DDS version lower than v2.0.2 you could not use the `fastdds` tool. You could update your repository to use a different Fast DDS version, or *set the discovery server by Fast-DDS XML QoS configuration*.

Note: This tutorial can also be run in *Galactic* exporting the environment variable that selects *Fast DDS* as the middleware layer:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
```

Run the demo

The `talker-listener` ROS 2 demo allows to create a *talker* node that publishes a *Hello World* message every second, and a *listener* node that listens to these messages.

By Sourcing ROS 2 you will get access to the CLI of *Fast DDS*: `fastdds`. This CLI gives access to the *discovery tool*, which allows to launch a server. This server will manage the discovery process for the nodes that connect to it.

Important: Do not forget to source ROS 2 in every new terminal opened.

Setup Discovery Server

Start by launching a server with id 0, with port 11811 and listening on all available interfaces.

Open a new terminal and run:

```
fastdds discovery -i 0
```

Launch node listener

Execute the listener demo, that will listen in `/chatter` topic.

In a new terminal, set the environment variable `ROS_DISCOVERY_SERVER` to use *Discovery Server*. (Do not forget to source ROS 2 in every new terminal)

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
```

Afterwards, launch the listener node. Use the argument `--remap __node:=listener_discovery_server` to change the node's name for future purpose.

```
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener_discovery_server
```

This process will create a ROS 2 node, that will automatically create a client for the *Discovery Server* and use the server created previously to run the discovery protocol.

Launch node talker

Open a new terminal and set the environment variable as before, so the node raises a client for the discovery protocol.

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker_discovery_server
```

Now, we should see the talker publishing *Hello World* messages, and the listener receiving these messages.

Demonstrate Discovery Server execution

So far, there is not proof that this example and the standard talker-listener example run differently. For this purpose, run another node that is not connected to our Discovery Server. Just run a new listener (listening in `/chatter` topic by default) in a new terminal and check that it is not connected to the talker already running.

```
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=simple_listener
```

In this case, we should not see the listener receiving the messages.

To finally verify that everything is running correctly, a new talker can be created using the *simple discovery protocol*.

```
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=simple_talker
```

Now we should see the listener *simple_listener* receiving the messages from *simple_talker* but not the other messages from *talker_discovery_server*.

Advance user cases

The following paragraphs are going to show different features of the Discovery Server that allows to hold a robust structure over the node's network.

Server Redundancy

By using the Fast DDS tool, several servers can be created, and the nodes can be connected to as many servers as desired. This allows to have a safe redundancy network that will work even if some servers or nodes shut down unexpectedly. Next schema shows a simple architecture that will work with server redundancy:

In different terminals, run the next code to establish a communication over redundant servers.

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

```
fastdds discovery -i 1 -l 127.0.0.1 -p 11888
```

`-i N` means server with id N. When referencing the servers with `ROS_DISCOVERY_SERVER`, server `0` must be in first place and server `1` in second place.

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

Now, if one of these servers fails, there would still be discovery communication between nodes.

Backup Server

Fast DDS Discovery Server allows to easily build a server with a **backup** functionality. This allows the server to retake the last state it saved in case of a shutdown.

In different terminals, run the next code to establish a communication over a backup server.

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811 -b
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

Several backup files are created in the path the server has run. Two `SQLite` files and two `json` files that contains the information required to raise a new server in case of failure, avoiding the whole discovery process to happen again and without losing information.

Discovery partitions

The **Discovery Server** communication could be used with different servers to split in virtual partitions the discovery info. This means that two endpoints only would know each other if there is a server or a server network between them. We are going to execute an example with two different independent servers. The following image shows a schema of the architecture desired:

With this schema *Listener 1* will be connected to *Talker 1* and *Talker 2*, as they share *Server 1*. *Listener 2* will connect with *Talker 1* as they share *Server 2*. But *Listener 2* will not hear the messages from *Talker 2* because they do not share any server or servers' network that connect them.

Run the first server listening in localhost in default port 11811.

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

In another terminal run the second server listening in localhost in port another port, in this case 11888.

```
fastdds discovery -i 1 -l 127.0.0.1 -p 11888
```

Now, run each node in a different terminal. Use the *environment variable* `ROS_DISCOVERY_SERVER` to decide which server they are connected to. Be aware that the ids must match (*Environment variables*).

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker_1
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener_1
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker_2
```

```
export ROS_DISCOVERY_SERVER=";127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener_2
```

We should see how *Listener 1* is receiving double messages, while *Listener 2* is in a different partition from *Talker 2* and so it does not listen to it.

Note: Once two endpoints know each other, they do not need the server network between them to listen to each other messages.

ROS 2 Introspection

ROS 2 Command Line Interface (CLI) implements several introspection features to analyze the behaviour of a ROS 2 execution. These features (i.e. *rosbag*, *topic list*, etc.) are very helpful to understand a ROS 2 working network.

Most of these features use the DDS capability to share any topic information with every exiting participant. However, the new *Discovery Server v2* implements a traffic network reduction that limits the discovery data between nodes that do not share a topic. This means that not every node will receive every topic data unless it has a reader in that topic. As most of ROS 2 CLI Introspection is executed by adding a node into the network (some of them use ROS 2 Daemon, and some create their own nodes), using Discovery Server v2 we will find that most of these functionalities are limited and do not have all the information.

The Discovery Server v2 functionality allows every node running as a *SUPER_CLIENT*, a kind of **Client** that connects to a *SERVER*, from which it receives all the available discovery information (instead of just what it needs). In this sense, ROS 2 introspection tools can be configured as **Super Client**, thus being able to discover every entity that is using the Discovery Server protocol within the network.

Daemon's related commands

The ROS 2 Daemon is used in several ROS 2 CLI introspection commands. It adds a ROS 2 Node to the network in order to receive all the data sent. In order for the ROS 2 CLI to work when using Discover Server discovery mechanism, the ROS 2 Daemon needs to be configured as **Super Client**. Therefore, this section is devoted to explain how to use ROS 2 CLI with ROS 2 Daemon running as a **Super Client**. This will allow the Daemon to discover the entire Node graph, and to receive every topic and endpoint information. To do so, a Fast DDS XML configuration file is used to configure the ROS 2 Daemon and CLI tools.

Warning: Although it is possible to run the ROS 2 Daemon as a **Server**, this is not recommended since the daemon will stop after two hours of inactivity, taking the **Server** down with it.

Below you can find a XML configuration file which will configure every new participant as a **Super Client**.

- `XML Super Client configuration file`

First of all, instantiate a Discovery Server using *Fast DDS CLI*

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

Run a talker and a listener that will discover each other through the Server (notice that `ROS_DISCOVERY_SERVER` configuration is the same as the one in *super_client_configuration_file.xml*).

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

Then, instantiate a ROS 2 Daemon using the **Super Client** configuration (remember to source ROS 2 installation in every new terminal).

```
export FASTRTPS_DEFAULT_PROFILES_FILE=super_client_configuration_file.xml
ros2 daemon stop
ros2 daemon start
ros2 topic list
ros2 node info /talker
ros2 topic info /chatter
ros2 topic echo /chatter
```

We can also see the Node's Graph using the ROS 2 tool *rqt_graph* as follows (you may need to press the refresh button):

```
export FASTRTPS_DEFAULT_PROFILES_FILE=super_client_configuration_file.xml
rqt_graph
```

No Daemon commands

Some ROS 2 CLI tools can be executed without the ROS 2 Daemon. In order for these tools to connect with a Discovery Server and receive all the topics information they need to be instantiated as a **Super Client** that connects to the **Server**.

Following the previous configuration, build a simple system with a talker and a listener. First, run a **Server**:

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

Then, run the talker and listener is separate terminals:

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

Continue using the ROS 2 CLI with `--no-daemon` option with the new configuration. New nodes will connect with the existing Server and will know every topic. Exporting `ROS_DISCOVERY_SERVER` is not needed as the remote server has been configured in the xml file.

```
export FASTRTPS_DEFAULT_PROFILES_FILE=super_client_configuration_file.xml
ros2 topic list --no-daemon
ros2 node info /talker --no-daemon --spin-time 2
```

Compare Discovery Server with Simple Discovery

In order to compare the ROS 2 execution using *Simple Discovery* or *Discovery Server*, two scripts that execute a talker and many listeners and analyze the network traffic during this time are provided. For this experiment, `tshark` is required to be installed on your system. The configuration file is mandatory in order to avoid using intra-process mode.

Note: These scripts require a Discovery Server closure feature that is only available from Fast DDS v2.1.0 and forward. In order to use this functionality, compile ROS 2 with Fast DDS v2.1.0 or higher.

These scripts' functionalities are references for advance purpose and their study is left to the user.

- `bash network traffic generator`
- `python3 graph generator`
- `XML configuration`

Run the bash script with the *setup* path to source ROS 2 as argument. This will generate the traffic trace for simple discovery. Executing the same script with second argument `SERVER`, it will generates the trace for service discovery.

Note: Depending on your configuration of `tcpdump`, this script may require `sudo` privileges to read traffic across your network device.

After both executions are done, run the python script to generates a graph similar to the one below:

```
$ export FASTRTPS_DEFAULT_PROFILES_FILE="no_intraprocess_configuration.xml"
$ sudo bash generate_discovery_packages.bash ~/ros2_foxy/install/local_setup.bash
```

(continues on next page)

(continued from previous page)

```
$ sudo bash generate_discovery_packages.bash ~/ros2_foxy/install/local_setup.bash SERVER
$ python3 discovery_packets.py
```

This graph is the result of a is a specific example, the user can execute the scripts and watch their own results. It can easily be seen how the network traffic is reduced when using *Discovery Service*.

The reduction in traffic is a result of avoiding every node announcing itself and waiting a response from every other node in the net. This creates a huge amount of traffic in large architectures. This reduction from this method increases with the number of Nodes, making this architecture more scalable than the simple one.

Since *Fast DDS* v2.0.2 the new Discovery Server v2 is available, substituting the old Discovery Server. In this new version, those nodes that do not share topics will not know each other, saving the whole discovery data required to connect them and their endpoints. Notice that this is not this example case, but even though the massive reduction could be appreciate due to the hidden architecture topics of ROS 2 nodes.

6.31 Troubleshooting

This section offers hints and pointers to help users with navigating through the documentation while troubleshooting issues.

- Although UDP/SHM default transports of Fast DDS are designed to work in most network environments, they may encounter certain limitations when operating over WiFi or within lossy network conditions. In these cases, it is advisable to set up the `LARGE_DATA` configuration, which has been specifically optimized for these scenarios. The `LARGE_DATA` profile limits the use of UDP solely to the *PDP discovery* phase, employing the more reliable TCP/SHM for the remainder of the communication process. Its implementation can be accomplished by simply configuring the `FASTDDS_BUILTIN_TRANSPORTS` environment variable, or alternatively through XML profiles or via code. For more information, please refer to *Large Data Mode*.

  Environment Variable

  ```
  export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA
  ```

  XML

  ```
  <?xml version="1.0" encoding="UTF-8" ?>
  <dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
      <profiles>
          <!--
              UDP transport for PDP and SHM/TCPv4 transport for both EDP and␣
  ↪application data
          -->
          <participant profile_name="large_data_builtin_transports" is_default_
  ↪profile="true">
              <rtps>
                  <builtinTransports>LARGE_DATA</builtinTransports>
              </rtps>
          </participant>
      </profiles>
  </dds>
  ```

  C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos = PARTICIPANT_QOS_DEFAULT;

/* Transports configuration */
// UDPv4 transport for PDP over multicast and SHM / TCPv4 transport for EDP and␣
↪application data
pqos.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_DATA);

/* Create participant as usual */
eprosima::fastdds::dds::DomainParticipant* participant =
        eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);
```

- Problems with transmitting **large samples such as video or point clouds**? Please refer to *Large Data Rates*.

6.32 C++ API Reference

Fast DDS, as a Data Distribution Service (DDS) standard implementation, exposes the DDS Data-Centric Publish-Subscribe (DCPS) Platform Independent Model (PIM) API, as specified in the DDS specification. Furthermore, is also gives the user the possibility to directly interact with the underlying Real-time Publish-Subscribe (RTPS) API that DDS implements for wired communications, as specified in the RTPS standard.

This section presents the most commonly used APIs provided by *Fast DDS*. For more information about the API reference, please refer to Fast DDS API reference.

6.32.1 DDS DCPS PIM

Data Distribution Service (DDS) Data-Centric Publish-Subscribe (DCPS) Platform Independent Model (PIM) API

Core

Entity

class **Entity**

The *Entity* class is the abstract base class for all the objects that support QoS policies, a listener and a status condition.

Subclassed by *eprosima::fastdds::dds::DomainEntity*, *eprosima::fastdds::dds::DomainParticipant*

Public Functions

inline **Entity**(const *StatusMask* &mask = *StatusMask*::*all*())

Constructor.

Parameters **mask** – *StatusMask* (default: all)

inline virtual fastrtps::types::ReturnCode_t **enable**()

This operation enables the *Entity*.

Returns RETCODE_OK

inline void **close**()

This operation disables the *Entity* before closing it.

inline const *StatusMask* &**get_status_mask**() const
Retrieves the set of relevant statuses for the *Entity*.

Returns Reference to the *StatusMask* with the relevant statuses set to 1

const *StatusMask* &**get_status_changes**() const
Retrieves the set of triggered statuses in the *Entity*.

Triggered statuses are the ones whose value has changed since the last time the application read the status. When the entity is first created or if the entity is not enabled, all communication statuses are in the non-triggered state, so the list returned by the get_status_changes operation will be empty. The list of statuses returned by the get_status_changes operation refers to the status that are triggered on the *Entity* itself and does not include statuses that apply to contained entities.

Returns const reference to the *StatusMask* with the triggered statuses set to 1

inline const InstanceHandle_t &**get_instance_handle**() const
Retrieves the instance handler that represents the *Entity*.

Returns Reference to the InstanceHandle

inline bool **is_enabled**() const
Checks if the *Entity* is enabled.

Returns true if enabled, false if not

inline *StatusCondition* &**get_statuscondition**()
Allows access to the *StatusCondition* associated with the *Entity*.

Returns Reference to *StatusCondition* object

DomainEntity

class **DomainEntity** : public eprosima::fastdds::dds::*Entity*
The *DomainEntity* class is a subclass of *Entity* created in order to differentiate between DomainParticipants and the rest of Entities.

Subclassed by *eprosima::fastdds::dds::DataReader*, *eprosima::fastdds::dds::DataWriter*, *eprosima::fastdds::dds::Publisher*, *eprosima::fastdds::dds::Subscriber*, *eprosima::fastdds::dds::Topic*

Public Functions

inline **DomainEntity**(const *StatusMask* &mask = *StatusMask*::*all*())
Constructor.

Parameters **mask** – *StatusMask* (default: all)

Policy

DataRepresentationId

enum eprosima::fastdds::dds::**DataRepresentationId**
Enum *DataRepresentationId*, different kinds of topic data representation

Values:

enumerator **XCDR_DATA_REPRESENTATION**
Extended CDR Encoding version 1.

enumerator **XML_DATA_REPRESENTATION**
XML Data Representation (Unsupported)

enumerator **XCDR2_DATA_REPRESENTATION**
Extended CDR Encoding version 2.

constexpr DataRepresentationId_t `eprosima::fastdds::dds::`**DEFAULT_DATA_REPRESENTATION** = {DataRepresentationId_t::XCDR_DATA_REPRESENTATION}
Default *DataRepresentationId* used in Fast DDS.

DataRepresentationQosPolicy

class **DataRepresentationQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
With multiple standard data Representations available, and vendor-specific extensions possible, DataWriters and DataReaders must be able to negotiate which data representation(s) to use. This negotiation shall occur based on *DataRepresentationQosPolicy*.

Note: Immutable Qos Policy

Warning: If a writer's offered representation is contained within a reader's sequence, the offer satisfies the request and the policies are compatible. Otherwise, they are incompatible.

Public Functions

inline **DataRepresentationQosPolicy**()
Constructor.

virtual **~DataRepresentationQosPolicy**() override = default
Destructor.

inline bool **operator==**(const *DataRepresentationQosPolicy* &b) const
Compares the given policy to check if it's equal.

Parameters **b** – QoS Policy.

Returns True if the policy is equal.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

std::vector<DataRepresentationId_t> **m_value**
List of *DataRepresentationId* .

By default, empty list.

DataSharingQosPolicy

class **DataSharingQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Qos Policy to configure the data sharing

Note: Immutable Qos Policy

Public Functions

inline **DataSharingQosPolicy**()
Constructor.

virtual **~DataSharingQosPolicy**() = default
Destructor.

inline **DataSharingQosPolicy**(const *DataSharingQosPolicy* &b)
Copy constructor.

Parameters **b** – Another *DataSharingQosPolicy* instance

inline virtual void **clear**() override
Clears the *QosPolicy* object.

inline const *DataSharingKind* &**kind**() const

Returns the current DataSharing configuration mode

inline const std::string &**shm_directory**() const

Returns the current DataSharing shared memory directory

inline const std::vector<uint64_t> &**domain_ids**() const
Gets the set of DataSharing domain IDs.

Each domain ID is 64 bit long. However, user-defined domain IDs are only 16 bit long, while the rest of the 48 bits are used for the automatically generated domain ID (if any).

- Automatic domain IDs use the 48 MSB and leave the 16 LSB as zero.
- User defined domain IDs use the 16 LSB and leave the 48 MSB as zero.

Returns the current DataSharing domain IDs

inline void **set_max_domains**(uint32_t size)

Parameters `size` – the new maximum number of domain IDs

inline const uint32_t &**max_domains**() const

Returns the current configured maximum number of domain IDs

inline void **automatic**()
Configures the DataSharing in automatic mode.

The default shared memory directory of the OS is used. A default domain ID is automatically computed.

inline void **automatic**(const std::vector<uint16_t> &domain_ids)
Configures the DataSharing in automatic mode.

The default shared memory directory of the OS is used.

Parameters `domain_ids` – the user configured DataSharing domain IDs (16 bits).

inline void **automatic**(const std::string &directory)
Configures the DataSharing in automatic mode.

A default domain ID is automatically computed.

Parameters `directory` – The shared memory directory to use.

inline void **automatic**(const std::string &directory, const std::vector<uint16_t> &domain_ids)
Configures the DataSharing in automatic mode.

Parameters

- `directory` – The shared memory directory to use.
- `domain_ids` – the user configured DataSharing domain IDs (16 bits).

inline void **on**(const std::string &directory)
Configures the DataSharing in active mode.

A default domain ID is automatically computed.

Parameters `directory` – The shared memory directory to use. It is mandatory to provide a non-empty name or the creation of endpoints will fail.

inline void **on**(const std::string &directory, const std::vector<uint16_t> &domain_ids)
Configures the DataSharing in active mode.

Parameters

- `directory` – The shared memory directory to use. It is mandatory to provide a non-empty name or the creation of endpoints will fail.
- `domain_ids` – the user configured DataSharing domain IDs (16 bits).

inline void **off**()
Configures the DataSharing in disabled mode.

inline void **add_domain_id**(uint16_t id)
Adds a user-specific DataSharing domain ID.

Parameters `id` – 16 bit identifier

inline rtps::ThreadSettings &**data_sharing_listener_thread**()
Getter for DataSharing listener thread ThreadSettings

Returns rtps::ThreadSettings reference

inline const rtps::ThreadSettings &**data_sharing_listener_thread**() const
Getter for DataSharing listener thread ThreadSettings

Returns rtps::ThreadSettings reference

inline void **data_sharing_listener_thread**(const rtps::ThreadSettings &value)
Setter for the DataSharing listener thread ThreadSettings

Parameters **value** – New ThreadSettings to be set

DataSharingKind

enum eprosima::fastdds::dds::**DataSharingKind**
Data sharing configuration kinds

Values:

enumerator **AUTO**
Automatic configuration. DataSharing will be used if requirements are met.

enumerator **ON**
Activate the use of DataSharing. *Entity* creation will fail if requirements for DataSharing are not met

enumerator **OFF**
Disable the use of DataSharing

DeadlineQosPolicy

class **DeadlineQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
DataReader expects a new sample updating the value of each instance at least once every deadline period. *DataWriter* indicates that the application commits to write a new value (using the *DataWriter*) for each instance managed by the *DataWriter* at least once every deadline period.

Note: Mutable Qos Policy

Public Functions

inline **DeadlineQosPolicy**()
Constructor.

virtual **~DeadlineQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

fastrtps::*Duration_t* **period**
Maximum time expected between samples. It is inconsistent for a *DataReader* to have a DEADLINE period less than its *TimeBasedFilterQosPolicy* minimum_separation.

By default, c_TimeInifinite.

DestinationOrderQosPolicy

class **DestinationOrderQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Controls the criteria used to determine the logical order among changes made by *Publisher* entities to the same instance of data (i.e., matching *Topic* and key).

Note: Immutable Qos Policy

Warning: This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Public Functions

inline **DestinationOrderQosPolicy**()
Constructor.

virtual **~DestinationOrderQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

DestinationOrderQosPolicyKind **kind**

DestinationOrderQosPolicyKind.

By default, BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS.

DestinationOrderQosPolicyKind

enum eprosima::fastdds::dds::**DestinationOrderQosPolicyKind**
Enum DestinationOrderQosPolicyKind, different kinds of destination order for *DestinationOrderQosPolicy*.

Values:

enumerator **BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS**
Indicates that data is ordered based on the reception time at each *Subscriber*. Since each subscriber may receive the data at different times there is no guaranteed that the changes will be seen in the same order. Consequently, it is possible for each subscriber to end up with a different final value for the data.

enumerator **BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS**
Indicates that data is ordered based on a timestamp placed at the source (by the Service or by the application). In any case this guarantees a consistent final value for the data in all subscribers.

DisablePositiveACKsQosPolicy

class **DisablePositiveACKsQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Class *DisablePositiveACKsQosPolicy* to disable sending of positive ACKs

Note: Immutable Qos Policy

Public Functions

inline **DisablePositiveACKsQosPolicy**()
Constructor.

virtual **~DisablePositiveACKsQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

bool **enabled**

True if this QoS is enabled.

By default, false.

fastrtps::*Duration_t* **duration**

The duration to keep samples for (not serialized as not needed by reader).

By default, c_TimeInfinite.

DurabilityQosPolicy

class **DurabilityQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
This policy expresses if the data should 'outlive' their writing time.

Note: Immutable Qos Policy

Public Functions

inline **DurabilityQosPolicy**()
Constructor.

virtual **~DurabilityQosPolicy**() = default
Destructor.

inline fastrtps::rtps::*DurabilityKind_t* **durabilityKind**() const
Translates kind to rtps layer equivalent

Returns fastrtps::rtps::DurabilityKind_t

inline void **durabilityKind**(const fastrtps::rtps::*DurabilityKind_t* new_kind)
Set kind passing the rtps layer equivalent kind

Parameters **new_kind** – fastrtps::rtps::DurabilityKind_t

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

DurabilityQosPolicyKind_t **kind**

DurabilityQosPolicyKind.

By default the value for DataReaders: VOLATILE_DURABILITY_QOS, for DataWriters TRANSIENT_LOCAL_DURABILITY_QOS.

DurabilityQosPolicyKind

enum eprosima::fastdds::dds::**DurabilityQosPolicyKind**
Enum DurabilityQosPolicyKind_t, different kinds of durability for *DurabilityQosPolicy*.

Values:

enumerator **VOLATILE_DURABILITY_QOS**
The Service does not need to keep any samples of data-instances on behalf of any *DataReader* that is not known by the *DataWriter* at the time the instance is written. In other words the Service will only attempt to provide the data to existing subscribers

enumerator **TRANSIENT_LOCAL_DURABILITY_QOS**
For TRANSIENT_LOCAL, the service is only required to keep the data in the memory of the *DataWriter* that wrote the data and the data is not required to survive the *DataWriter*.

enumerator **TRANSIENT_DURABILITY_QOS**
For TRANSIENT, the service is only required to keep the data in memory and not in permanent storage; but the data is not tied to the lifecycle of the *DataWriter* and will, in general, survive it.

enumerator **PERSISTENT_DURABILITY_QOS**
Data is kept on permanent storage, so that they can outlive a system session.

> **Warning:** Not Supported

DurabilityServiceQosPolicy

class **DurabilityServiceQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Specifies the configuration of the durability service. That is, the service that implements the *DurabilityQosPolicy* kind of TRANSIENT and PERSISTENT.

Note: Immutable Qos Policy

> **Warning:** This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Public Functions

inline **DurabilityServiceQosPolicy**()
Constructor.

virtual **~DurabilityServiceQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

fastrtps::*Duration_t* **service_cleanup_delay**

Control when the service is able to remove all information regarding a data-instance.

By default, c_TimeZero.

HistoryQosPolicyKind **history_kind**
Controls the *HistoryQosPolicy* of the fictitious *DataReader* that stores the data within the durability service.

By default, KEEP_LAST_HISTORY_QOS.

int32_t **history_depth**

Number of most recent values that should be maintained on the History. It only have effect if the history_kind is KEEP_LAST_HISTORY_QOS.

By default, 1.

int32_t **max_samples**
Control the ResourceLimitsQos of the implied *DataReader* that stores the data within the durability service. Specifies the maximum number of data-samples the *DataWriter* (or *DataReader*) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one *DataWriter* (or *DataReader*). It is inconsistent for this value to be less than max_samples_per_instance.

By default, LENGTH_UNLIMITED.

int32_t **max_instances**
Control the ResourceLimitsQos of the implied *DataReader* that stores the data within the durability service. Represents the maximum number of instances *DataWriter* (or *DataReader*) can manage.

By default, LENGTH_UNLIMITED.

int32_t **max_samples_per_instance**
Control the ResourceLimitsQos of the implied *DataReader* that stores the data within the durability service. Represents the maximum number of samples of any one instance a DataWriter(or DataReader) can manage. It is inconsistent for this value to be greater than max_samples.

By default, LENGTH_UNLIMITED.

EntityFactoryQosPolicy

class **EntityFactoryQosPolicy**
Controls the behavior of the entity when acting as a factory for other entities. In other words, configures the side-effects of the create_* and delete_* operations.

Note: Mutable Qos Policy

Public Functions

inline **EntityFactoryQosPolicy**()
Constructor without parameters.

inline **EntityFactoryQosPolicy**(bool autoenable)
Constructor.

Parameters **autoenable** – Value for the autoenable_created_entities boolean

inline virtual **~EntityFactoryQosPolicy**()
Destructor.

Public Members

bool **autoenable_created_entities**
Specifies whether the entity acting as a factory automatically enables the instances it creates. If True the factory will automatically enable each created *Entity* otherwise it will not.

By default, True.

GenericDataQosPolicy

class **GenericDataQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*, public fastrtps::ResourceLimitedVector<fastrtps::rtps::*octet*>
Class *GenericDataQosPolicy*, base class to transmit user data during the discovery phase.

Subclassed by *eprosima::fastdds::dds::GroupDataQosPolicy*, *eprosima::fastdds::dds::TopicDataQosPolicy*, *eprosima::fastdds::dds::UserDataQosPolicy*

Public Functions

inline **GenericDataQosPolicy**(const *GenericDataQosPolicy* &data)
Construct from another *GenericDataQosPolicy*.

The resulting *GenericDataQosPolicy* will have the same size limits as the input attribute

Parameters **data** – data to copy in the newly created object

inline **GenericDataQosPolicy**(ParameterId_t pid, const collection_type &data)
Construct from underlying collection type.

Useful to easy integration on old APIs where a traditional container was used. The resulting *GenericDataQosPolicy* will always be unlimited in size

Parameters

- **pid** – Id of the parameter
- **data** – data to copy in the newly created object

inline *GenericDataQosPolicy* &**operator=**(const collection_type &b)
Copies data from underlying collection type.

Useful to easy integration on old APIs where a traditional container was used. The resulting *GenericDataQosPolicy* will keep the current size limit. If the input data is larger than the current limit size, the elements exceeding that maximum will be silently discarded.

Parameters **b** – object to be copied

Returns reference to the current object.

inline *GenericDataQosPolicy* &**operator=**(const *GenericDataQosPolicy* &b)
Copies another *GenericDataQosPolicy*.

The resulting *GenericDataQosPolicy* will have the same size limit as the input parameter, so all data in the input will be copied.

Parameters **b** – object to be copied

Returns reference to the current object.

inline void **set_max_size**(size_t size)
Set the maximum size of the user data and reserves memory for that much.

Parameters **size** – new maximum size of the user data. Zero for unlimited size

inline const collection_type &**dataVec**() const

Returns const reference to the internal raw data.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

inline const collection_type &**data_vec**() const
Returns raw data vector.

Returns raw data as vector of octets.

inline collection_type &**data_vec**()
Returns raw data vector.

Returns raw data as vector of octets.

inline void **data_vec**(const collection_type &vec)
Sets raw data vector.

Parameters **vec** – raw data to set.

inline const collection_type &**getValue**() const
Returns raw data vector.

Returns raw data as vector of octets.

inline void **setValue**(const collection_type &vec)
Sets raw data vector.

Parameters **vec** – raw data to set.

GroupDataQosPolicy

class **GroupDataQosPolicy** : public eprosima::fastdds::dds::*GenericDataQosPolicy*
Class derived from *GenericDataQosPolicy*.

The purpose of this QoS is to allow the application to attach additional information to the created *Publisher* or *Subscriber*. The value of the GROUP_DATA is available to the application on the *DataReader* and *DataWriter* entities and is propagated by means of the built-in topics.

This QoS can be used by an application combination with the *DataReaderListener* and *DataWriterListener* to implement matching policies similar to those of the PARTITION QoS except the decision can be made based on an application-defined policy.

HistoryQosPolicy

class **HistoryQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Specifies the behavior of the Service in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers. This QoS policy controls whether the Service should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between. On the publishing side this policy controls the samples that should be maintained by the *DataWriter* on behalf of existing *DataReader* entities. The behavior with regards to a DataReaderentities discovered after a sample is written is controlled by the DURABILITY QoS policy. On the subscribing side it controls the samples that should be maintained until the application "takes" them from the Service.

Note: Immutable Qos Policy

Public Functions

inline **HistoryQosPolicy**()
Constructor.

virtual **~HistoryQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

HistoryQosPolicyKind **kind**

HistoryQosPolicyKind.

By default, KEEP_LAST_HISTORY_QOS.

int32_t **depth**
History depth.

By default, 1. If a value other than 1 is specified, it should be consistent with the settings of the *ResourceLimitsQosPolicy*.

Warning: Only takes effect if the kind is KEEP_LAST_HISTORY_QOS.

HistoryQosPolicyKind

enum eprosima::fastdds::dds::**HistoryQosPolicyKind**
Enum HistoryQosPolicyKind, different kinds of History Qos for *HistoryQosPolicy*.

Values:

enumerator **KEEP_LAST_HISTORY_QOS**
On the publishing side, the Service will only attempt to keep the most recent "depth" samples of each instance of data (identified by its key) managed by the *DataWriter*. On the subscribing side, the *DataReader* will only attempt to keep the most recent "depth" samples received for each instance (identified by its key) until the application "takes" them via the DataReader's take operation.

enumerator **KEEP_ALL_HISTORY_QOS**
On the publishing side, the Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its key) managed by the *DataWriter* until they can be delivered to all subscribers. On the subscribing side, the Service will attempt to keep all samples of each instance of data (identified by its key) managed by the *DataReader*. These samples are kept until the application "takes" them from the Service via the take operation.

LatencyBudgetQosPolicy

class **LatencyBudgetQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver's application-cache and the receiving application is notified of the fact.This policy is a hint to the Service, not something that must be monitored or enforced. The Service is not required to track or alert the user of any violation.

Note: Mutable Qos Policy

Warning: This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Public Functions

inline **LatencyBudgetQosPolicy**()
Constructor.

virtual **~LatencyBudgetQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

fastrtps::*Duration_t* **duration**

Maximum acceptable delay from the time data is written until it is received.

By default, c_TimeZero.

LifespanQosPolicy

class **LifespanQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Specifies the maximum duration of validity of the data written by the *DataWriter*.

Note: Mutable Qos Policy

Public Functions

inline **LifespanQosPolicy**()
Constructor.

virtual **~LifespanQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

fastrtps::*Duration_t* **duration**

Period of validity.

By default, c_TimeInfinite.

LivelinessQosPolicy

class **LivelinessQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Determines the mechanism and parameters used by the application to determine whether an *Entity* is "active" (alive). The "liveliness" status of an *Entity* is used to maintain instance ownership in combination with the setting of the *OwnershipQosPolicy*. The application is also informed via listener when an *Entity* is no longer alive.

The *DataReader* requests that liveliness of the writers is maintained by the requested means and loss of liveliness is detected with delay not to exceed the lease_duration.

The *DataWriter* commits to signaling its liveliness using the stated means at intervals not to exceed the lease_duration. Listeners are used to notify the DataReaderof loss of liveliness and *DataWriter* of violations to the liveliness contract.

Public Functions

inline **LivelinessQosPolicy**()
Constructor.

virtual **~LivelinessQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

LivelinessQosPolicyKind **kind**

Liveliness kind

By default, AUTOMATIC_LIVELINESS.

fastrtps::*Duration_t* **lease_duration**
Period within which liveliness should be asserted.

On a *DataWriter* it represents the period it commits to signal its liveliness. On a *DataReader* it represents the period without assertion after which a *DataWriter* is considered inactive. By default, c_TimeInfinite.

fastrtps::*Duration_t* **announcement_period**
The period for automatic assertion of liveliness.

Only used for DataWriters with AUTOMATIC liveliness. By default, c_TimeInfinite.

> **Warning:** When not infinite, must be < lease_duration, and it is advisable to be less than 0.7*lease_duration.

LivelinessQosPolicyKind

enum `eprosima::fastdds::dds::`**LivelinessQosPolicyKind**
Enum LivelinessQosPolicyKind, different kinds of liveliness for *LivelinessQosPolicy*

Values:

enumerator **AUTOMATIC_LIVELINESS_QOS**
The infrastructure will automatically signal liveliness for the DataWriters at least as often as required by the lease_duration.

enumerator **MANUAL_BY_PARTICIPANT_LIVELINESS_QOS**
The Service will assume that as long as at least one *Entity* within the *DomainParticipant* has asserted its liveliness the other Entities in that same *DomainParticipant* are also alive.

enumerator **MANUAL_BY_TOPIC_LIVELINESS_QOS**
The Service will only assume liveliness of the *DataWriter* if the application has asserted liveliness of that *DataWriter* itself.

OwnershipQosPolicy

class **OwnershipQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Specifies whether it is allowed for multiple DataWriters to write the same instance of the data and if so, how these modifications should be arbitrated

Note: Immutable Qos Policy

Public Functions

inline **OwnershipQosPolicy**()
Constructor.

virtual **~OwnershipQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

OwnershipQosPolicyKind **kind**
OwnershipQosPolicyKind.

OwnershipQosPolicyKind

enum eprosima::fastdds::dds::**OwnershipQosPolicyKind**
Enum OwnershipQosPolicyKind, different kinds of ownership for *OwnershipQosPolicy*.

Values:

enumerator **SHARED_OWNERSHIP_QOS**
Indicates shared ownership for each instance. Multiple writers are allowed to update the same instance and all the updates are made available to the readers. In other words there is no concept of an "owner" for the instances.

enumerator **EXCLUSIVE_OWNERSHIP_QOS**
Indicates each instance can only be owned by one *DataWriter*, but the owner of an instance can change dynamically. The selection of the owner is controlled by the setting of the *OwnershipStrengthQosPolicy*. The owner is always set to be the highest-strength *DataWriter* object among the ones currently "active" (as determined by the *LivelinessQosPolicy*).

OwnershipStrengthQosPolicy

class **OwnershipStrengthQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*

Specifies the value of the "strength" used to arbitrate among multiple *DataWriter* objects that attempt to modify the same instance of a data-object (identified by *Topic* + key).This policy only applies if the OWNERSHIP QoS policy is of kind EXCLUSIVE.

Note: Mutable Qos Policy

Public Functions

inline **OwnershipStrengthQosPolicy**()

Constructor.

virtual **~OwnershipStrengthQosPolicy**() = default

Destructor.

inline virtual void **clear**() override

Clears the *QosPolicy* object.

Public Members

uint32_t **value**

Strength

By default, 0.

ParticipantResourceLimitsQos

using eprosima::fastdds::dds::**ParticipantResourceLimitsQos** = fastrtps::rtps::*RTPSParticipantAllocationAttributes*

Holds allocation limits affecting collections managed by a participant.

Partition_t

class **Partition_t**

Public Functions

inline explicit **Partition_t**(const void *ptr)
Constructor using a pointer.

Parameters **ptr** – Pointer to be set

inline uint32_t **size**() const
Getter for the size.

Returns uint32_t with the size

inline const char ***name**() const
Getter for the partition name.

Returns name

PartitionQosPolicy

class **PartitionQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Set of strings that introduces a logical partition among the topics visible by the *Publisher* and *Subscriber*. A *DataWriter* within a *Publisher* only communicates with a *DataReader* in a *Subscriber* if (in addition to matching the *Topic* and having compatible QoS) the *Publisher* and *Subscriber* have a common partition name string.

The empty string ("") is considered a valid partition that is matched with other partition names using the same rules of string matching and regular-expression matching used for any other partition name.

Note: Mutable Qos Policy

Public Functions

inline **PartitionQosPolicy**()
Constructor without parameters.

inline **PartitionQosPolicy**(uint16_t in_length)
Constructor using Parameter length.

Parameters **in_length** – Length of the parameter

inline **PartitionQosPolicy**(const *PartitionQosPolicy* &b)
Copy constructor.

Parameters **b** – Another *PartitionQosPolicy* instance

virtual **~PartitionQosPolicy**() = default
Destructor.

inline *const_iterator* **begin**() const
Getter for the first position of the partition list.

Returns *const_iterator*

inline *const_iterator* **end**() const
Getter for the end of the partition list.

Returns *const_iterator*

inline uint32_t **size**() const
Getter for the number of partitions.

Returns uint32_t with the size

inline uint32_t **empty**() const
Check if the set is empty.

Returns true if it is empty, false otherwise

inline void **set_max_size**(uint32_t size)
Setter for the maximum size reserved for partitions (in bytes)

Parameters **size** – Size to be set

inline uint32_t **max_size**() const
Getter for the maximum size (in bytes)

Returns uint32_t with the maximum size

inline void **push_back**(const char *name)
Appends a name to the list of partition names.

Parameters **name** – Name to append.

inline virtual void **clear**() override
Clears list of partition names

inline const std::vector<std::string> **getNames**() const
Returns partition names.

Returns Vector of partition name strings.

inline void **setNames**(std::vector<std::string> &nam)
Overrides partition names

Parameters **nam** – Vector of partition name strings.

inline const std::vector<std::string> **names**() const
Returns partition names.

Returns Vector of partition name strings.

inline void **names**(std::vector<std::string> &nam)
Overrides partition names

Parameters **nam** – Vector of partition name strings.

class **const_iterator**

Public Functions

inline **const_iterator**(const fastrtps::rtps::*octet* *ptr)
Constructor using a pointer.
Parameters **ptr** – Pointer to be set

PresentationQosPolicy

class **PresentationQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*

Specifies how the samples representing changes to data instances are presented to the subscribing application. This policy affects the application's ability to specify and receive coherent changes and to see the relative order of changes.access_scope determines the largest scope spanning the entities for which the order and coherency of changes can be preserved. The two booleans control whether coherent access and ordered access are supported within the scope access_scope.

Note: Immutable Qos Policy

Warning: This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Public Functions

inline **PresentationQosPolicy**()

Constructor without parameters.

virtual **~PresentationQosPolicy**() = default

Destructor.

inline virtual void **clear**() override

Clears the *QosPolicy* object.

Public Members

PresentationQosPolicyAccessScopeKind **access_scope**

Access Scope Kind

By default, INSTANCE_PRESENTATION_QOS.

bool **coherent_access**

Specifies support coherent access. That is, the ability to group a set of changes as a unit on the publishing end such that they are received as a unit at the subscribing end. by default, false.

bool **ordered_access**

Specifies support for ordered access to the samples received at the subscription end. That is, the ability of the subscriber to see changes in the same order as they occurred on the publishing end. By default, false.

PresentationQosPolicyAccessScopeKind

enum eprosima::fastdds::dds::**PresentationQosPolicyAccessScopeKind**
Enum PresentationQosPolicyAccessScopeKind, different kinds of Presentation Policy order for *PresentationQosPolicy*.

Values:

enumerator **INSTANCE_PRESENTATION_QOS**
Scope spans only a single instance. Indicates that changes to one instance need not be coherent nor ordered with respect to changes to any other instance. In other words, order and coherent changes apply to each instance separately.

enumerator **TOPIC_PRESENTATION_QOS**
Scope spans to all instances within the same *DataWriter* (or *DataReader*), but not across instances in different *DataWriter* (or *DataReader*).

enumerator **GROUP_PRESENTATION_QOS**
Scope spans to all instances belonging to *DataWriter* (or *DataReader*) entities within the same *Publisher* (or *Subscriber*).

PropertyPolicyQos

using eprosima::fastdds::dds::**PropertyPolicyQos** = fastrtps::rtps::*PropertyPolicy*
Property policies.

PublishModeQosPolicy

class **PublishModeQosPolicy** : public eprosima::fastdds::dds::*QosPolicy*
Class *PublishModeQosPolicy*, defines the publication mode for a specific writer.

Public Functions

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

PublishModeQosPolicyKind **kind** = SYNCHRONOUS_PUBLISH_MODE

PublishModeQosPolicyKind

By default, SYNCHRONOUS_PUBLISH_MODE.

const char ***flow_controller_name** = fastdds::rtps::*FASTDDS_FLOW_CONTROLLER_DEFAULT*
Name of the flow controller used when publish mode kind is ASYNCHRONOUS_PUBLISH_MODE.

Since 2.4.0

PublishModeQosPolicyKind

enum eprosima::fastdds::dds::**PublishModeQosPolicyKind**
Enum PublishModeQosPolicyKind, different kinds of publication synchronism

Values:

enumerator **SYNCHRONOUS_PUBLISH_MODE**
Synchronous publication mode (default for writers).

enumerator **ASYNCHRONOUS_PUBLISH_MODE**
Asynchronous publication mode.

QosPolicy

class **QosPolicy**
Class *QosPolicy*, base for all QoS policies defined for Writers and Readers.

Subclassed by *eprosima::fastdds::dds::DataRepresentationQosPolicy*, *eprosima::fastdds::dds::DataSharingQosPolicy*, *eprosima::fastdds::dds::DeadlineQosPolicy*, *eprosima::fastdds::dds::DestinationOrderQosPolicy*, *eprosima::fastdds::dds::DisablePositiveACKsQosPolicy*, *eprosima::fastdds::dds::DurabilityQosPolicy*, *eprosima::fastdds::dds::DurabilityServiceQosPolicy*, *eprosima::fastdds::dds::GenericDataQosPolicy*, *eprosima::fastdds::dds::HistoryQosPolicy*, *eprosima::fastdds::dds::LatencyBudgetQosPolicy*, *eprosima::fastdds::dds::LifespanQosPolicy*, *eprosima::fastdds::dds::LivelinessQosPolicy*, *eprosima::fastdds::dds::OwnershipQosPolicy*, *eprosima::fastdds::dds::OwnershipStrengthQosPolicy*, *eprosima::fastdds::dds::PartitionQosPolicy*, *eprosima::fastdds::dds::PresentationQosPolicy*, *eprosima::fastdds::dds::PublishModeQosPolicy*, *eprosima::fastdds::dds::ReliabilityQosPolicy*, *eprosima::fastdds::dds::ResourceLimitsQosPolicy*, *eprosima::fastdds::dds::TimeBasedFilterQosPolicy*, *eprosima::fastdds::dds::TransportConfigQos*, *eprosima::fastdds::dds::TransportPriorityQosPolicy*, *eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy*, *eprosima::fastdds::dds::TypeConsistencyQos*, *eprosima::fastdds::dds::TypeIdV1*, *eprosima::fastdds::dds::TypeObjectV1*, *eprosima::fastdds::dds::WireProtocolConfigQos*, *eprosima::fastdds::dds::xtypes::TypeInformation*

Public Functions

inline **QosPolicy**()
Constructor without parameters.

inline explicit **QosPolicy**(bool send_always)
Constructor.

Parameters **send_always** – Boolean that set if the Qos need to be sent even if it is not changed

QosPolicy(const *QosPolicy* &b) = default
Copy Constructor.

Parameters **b** – Another instance of *QosPolicy*

virtual **~QosPolicy**() = default
Destructor.

inline virtual bool **send_always**() const
Whether it should always be sent.

Returns True if it should always be sent.

inline virtual void **clear**() = 0
Clears the *QosPolicy* object.

Public Members

bool **hasChanged**
Boolean that indicates if the Qos has been changed with respect to the default Qos.

QosPolicyId_t

enum eprosima::fastdds::dds::**QosPolicyId_t**
The identifier for each *QosPolicy*.

Each *QosPolicy* class has a different ID that is then used to refer to the incompatible policies on OfferedIncompatibleQosStatus and RequestedIncompatibleQosStatus.

Values:

enumerator **INVALID_QOS_POLICY_ID**

enumerator **USERDATA_QOS_POLICY_ID**

enumerator **DURABILITY_QOS_POLICY_ID**

enumerator **PRESENTATION_QOS_POLICY_ID**

enumerator **DEADLINE_QOS_POLICY_ID**

enumerator **LATENCYBUDGET_QOS_POLICY_ID**

enumerator **OWNERSHIP_QOS_POLICY_ID**

enumerator **OWNERSHIPSTRENGTH_QOS_POLICY_ID**

enumerator **LIVELINESS_QOS_POLICY_ID**

enumerator **TIMEBASEDFILTER_QOS_POLICY_ID**

enumerator **PARTITION_QOS_POLICY_ID**

enumerator **RELIABILITY_QOS_POLICY_ID**

enumerator **DESTINATIONORDER_QOS_POLICY_ID**

enumerator **HISTORY_QOS_POLICY_ID**

enumerator **RESOURCELIMITS_QOS_POLICY_ID**

enumerator **ENTITYFACTORY_QOS_POLICY_ID**

enumerator **WRITERDATALIFECYCLE_QOS_POLICY_ID**

enumerator **READERDATALIFECYCLE_QOS_POLICY_ID**

enumerator **TOPICDATA_QOS_POLICY_ID**

enumerator **GROUPDATA_QOS_POLICY_ID**

enumerator **TRANSPORTPRIORITY_QOS_POLICY_ID**

enumerator **LIFESPAN_QOS_POLICY_ID**

enumerator **DURABILITYSERVICE_QOS_POLICY_ID**

enumerator **DATAREPRESENTATION_QOS_POLICY_ID**

enumerator **TYPECONSISTENCYENFORCEMENT_QOS_POLICY_ID**

enumerator **DISABLEPOSITIVEACKS_QOS_POLICY_ID**

enumerator **PARTICIPANTRESOURCELIMITS_QOS_POLICY_ID**

enumerator **PROPERTYPOLICY_QOS_POLICY_ID**

enumerator **PUBLISHMODE_QOS_POLICY_ID**

enumerator **READERRESOURCELIMITS_QOS_POLICY_ID**

enumerator **RTPSENDPOINT_QOS_POLICY_ID**

enumerator **RTPSRELIABLEREADER_QOS_POLICY_ID**

enumerator **RTPSRELIABLEWRITER_QOS_POLICY_ID**

enumerator **TRANSPORTCONFIG_QOS_POLICY_ID**

enumerator **TYPECONSISTENCY_QOS_POLICY_ID**

enumerator **WIREPROTOCOLCONFIG_QOS_POLICY_ID**

enumerator **WRITERRESOURCELIMITS_QOS_POLICY_ID**

enumerator **NEXT_QOS_POLICY_ID**

ReaderDataLifecycleQosPolicy

class **ReaderDataLifecycleQosPolicy**

Specifies the behavior of the *DataReader* with regards to the lifecycle of the data-instances it manages.

Note: Mutable Qos Policy

Warning: This Qos Policy will be implemented in future releases.

Public Functions

inline **ReaderDataLifecycleQosPolicy**()

Constructor.

inline virtual **~ReaderDataLifecycleQosPolicy**()

Destructor.

Public Members

Duration_t **autopurge_no_writer_samples_delay**
Indicates the duration the *DataReader* must retain information regarding instances that have the instance_state NOT_ALIVE_NO_WRITERS.

By default, c_TimeInfinite.

Duration_t **autopurge_disposed_samples_delay**
Indicates the duration the *DataReader* must retain information regarding instances that have the instance_state NOT_ALIVE_DISPOSED.

By default, c_TimeInfinite.

ReliabilityQosPolicy

class **ReliabilityQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Indicates the reliability of the endpoint.

Note: Immutable Qos Policy

Public Functions

inline **ReliabilityQosPolicy**()
Constructor.

virtual **~ReliabilityQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

ReliabilityQosPolicyKind **kind**
Defines the reliability kind of the endpoint.

By default, BEST_EFFORT_RELIABILITY_QOS for DataReaders and RELIABLE_RELIABILITY_QOS for DataWriters.

fastrtps::*Duration_t* **max_blocking_time**
Defines the maximum period of time certain methods will be blocked.

Methods affected by this property are:

- *DataWriter::write*
- DataReader::takeNextData

- DataReader::readNextData

  By default, 100 ms.

ReliabilityQosPolicyKind

enum eprosima::fastdds::dds::**ReliabilityQosPolicyKind**
Enum ReliabilityQosPolicyKind, different kinds of reliability for *ReliabilityQosPolicy*.

Values:

enumerator **BEST_EFFORT_RELIABILITY_QOS**
Indicates that it is acceptable to not retry propagation of any samples. Presumably new values for the samples are generated often enough that it is not necessary to re-send or acknowledge any samples

enumerator **RELIABLE_RELIABILITY_QOS**
Specifies the Service will attempt to deliver all samples in its history. Missed samples may be retried. In steady-state (no modifications communicated via the *DataWriter*) the middleware guarantees that all samples in the *DataWriter* history will eventually be delivered to all the *DataReader* objects. Outside steady state the *HistoryQosPolicy* and *ResourceLimitsQosPolicy* will determine how samples become part of the history and whether samples can be discarded from it.

ResourceLimitsQosPolicy

class **ResourceLimitsQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Specifies the resources that the Service can consume in order to meet the requested QoS

Note: Immutable Qos Policy

Public Functions

inline **ResourceLimitsQosPolicy**()
Constructor.

virtual **~ResourceLimitsQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

int32_t **max_samples**
Specifies the maximum number of data-samples the *DataWriter* (or *DataReader*) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one *DataWriter* (or *DataReader*).

Value 0 means infinite resources. By default, 5000.

> **Warning:** It is inconsistent if `max_samples < (max_instances * max_samples_per_instance)`.

int32_t **max_instances**
Represents the maximum number of instances *DataWriter* (or *DataReader*) can manage.

Value 0 means infinite resources. By default, 10.

> **Warning:** It is inconsistent if `(max_instances * max_samples_per_instance) > max_samples`.

int32_t **max_samples_per_instance**

Represents the maximum number of samples of any one instance a DataWriter(or DataReader) can manage.

Value 0 means infinite resources. By default, 400.

> **Warning:** It is inconsistent if `(max_instances * max_samples_per_instance) > max_samples`.

int32_t **allocated_samples**

Number of samples currently allocated.

By default, 100.

int32_t **extra_samples**
Represents the extra number of samples available once the max_samples have been reached in the history. This makes it possible, for example, to loan samples even with a full history. By default, 1.

RTPSEndpointQos

class **RTPSEndpointQos**
Qos Policy to configure the endpoint.

Public Members

rtps::*LocatorList* **unicast_locator_list**
Unicast locator list.

rtps::*LocatorList* **multicast_locator_list**
Multicast locator list.

rtps::*LocatorList* **remote_locator_list**
Remote locator list.

fastdds::rtps::*ExternalLocators* **external_unicast_locators**
The collection of external locators to use for communication.

bool **ignore_non_matching_locators** = false
Whether locators that don't match with the announced locators should be kept.

int16_t **user_defined_id** = -1

User Defined ID, used for StaticEndpointDiscovery.

By default, -1.

int16_t **entity_id** = -1
Entity ID, if the user wants to specify the EntityID of the endpoint.

By default, -1.

fastrtps::rtps::MemoryManagementPolicy_t **history_memory_policy** =
fastrtps::rtps::PREALLOCATED_WITH_REALLOC_MEMORY_MODE

Underlying History memory policy.

By default, PREALLOCATED_WITH_REALLOC_MEMORY_MODE.

TimeBasedFilterQosPolicy

class **TimeBasedFilterQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Filter that allows a *DataReader* to specify that it is interested only in (potentially) a subset of the values of the data. The filter states that the *DataReader* does not want to receive more than one value each minimum_separation, regardless of how fast the changes occur. It is inconsistent for a *DataReader* to have a minimum_separation longer than its Deadline period.

Note: Mutable Qos Policy

Warning: This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Public Functions

inline **TimeBasedFilterQosPolicy**()
Constructor.

virtual **~TimeBasedFilterQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

fastrtps::*Duration_t* **minimum_separation**
Minimum interval between samples. By default, c_TimeZero (the *DataReader* is interested in all values)

TopicDataQosPolicy

class **TopicDataQosPolicy** : public eprosima::fastdds::dds::*GenericDataQosPolicy*
Class derived from *GenericDataQosPolicy*.

The purpose of this QoS is to allow the application to attach additional information to the created *Topic* such that when a remote application discovers their existence it can examine the information and use it in an application-defined way.

In combination with the listeners on the *DataReader* and *DataWriter* as well as by means of operations such as ignore_topic,these QoS can assist an application to extend the provided QoS.

TransportConfigQos

class **TransportConfigQos** : public eprosima::fastdds::dds::*QosPolicy*
Qos Policy to configure the transport layer.

Public Functions

inline **TransportConfigQos**()
Constructor.

virtual **~TransportConfigQos**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

std::vector<std::shared_ptr<fastdds::rtps::*TransportDescriptorInterface*>> **user_transports**
User defined transports to use alongside or in place of builtins.

bool **use_builtin_transports**

Set as false to disable the default UDPv4 implementation.

By default, true.

uint32_t **send_socket_buffer_size**
Send socket buffer size for the send resource.

Zero value indicates to use default system buffer size.

By default, 0.

uint32_t **listen_socket_buffer_size**
Listen socket buffer for all listen resources.

Zero value indicates to use default system buffer size.

By default, 0.

rtps::ThreadSettings **builtin_transports_reception_threads_**
Thread settings for the builtin transports reception threads.

uint32_t **max_msg_size_no_frag**
Maximum message size used to avoid fragmentation, set ONLY in LARGE_DATA.

If this value is not zero, the network factory will allow the initialization of UDP transports with maxMessageSize higher than 65500K.

fastdds::rtps::NetmaskFilterKind **netmask_filter**
Netmask filter configuration.

TransportPriorityQosPolicy

class **TransportPriorityQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
This policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data.

Note: Mutable Qos Policy

Warning: This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Public Functions

inline **TransportPriorityQosPolicy**()
Constructor.

virtual **~TransportPriorityQosPolicy**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

uint32_t **value**

Priority

By default, 0.

TypeConsistencyEnforcementQosPolicy

class **TypeConsistencyEnforcementQosPolicy** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
The *TypeConsistencyEnforcementQosPolicy* defines the rules for determining whether the type used to publish a given data stream is consistent with that used to subscribe to it. It applies to DataReaders.

Note: Immutable Qos Policy

Public Functions

inline **TypeConsistencyEnforcementQosPolicy**()
Constructor.

virtual **~TypeConsistencyEnforcementQosPolicy**() override = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

TypeConsistencyKind **m_kind**

TypeConsistencyKind.

By default, ALLOW_TYPE_COERCION.

bool **m_ignore_sequence_bounds**

This option controls whether sequence bounds are taken into consideration for type assignability. If the option is set to TRUE, sequence bounds (maximum lengths) are not considered as part of the type assignability. This means that a T2 sequence type with maximum length L2 would be assignable to a T1 sequence type

with maximum length L1, even if L2 is greater than L1. If the option is set to false, then sequence bounds are taken into consideration for type assignability and in order for T1 to be assignable from T2 it is required that L1>= L2.

By default, true.

bool **m_ignore_string_bounds**

This option controls whether string bounds are taken into consideration for type assignability. If the option is set to TRUE, string bounds (maximum lengths) are not considered as part of the type assignability. This means that a T2 string type with maximum length L2 would be assignable to a T1 string type with maximum length L1, even if L2 is greater than L1. If the option is set to false, then string bounds are taken into consideration for type assignability and in order for T1 to be assignable from T2 it is required that L1>= L2.

By default, true.

bool **m_ignore_member_names**

This option controls whether member names are taken into consideration for type assignability. If the option is set to TRUE, member names are considered as part of assignability in addition to member IDs (so that members with the same ID also have the same name). If the option is set to FALSE, then member names are not ignored.

By default, false.

bool **m_prevent_type_widening**

This option controls whether type widening is allowed. If the option is set to FALSE, type widening is permitted. If the option is set to TRUE,it shall cause a wider type to not be assignable to a narrower type.

By default, false.

bool **m_force_type_validation**

This option requires type information to be available in order to complete matching between a *DataWriter* and *DataReader* when set to TRUE, otherwise matching can occur without complete type information when set to FALSE.

By default, false.

TypeConsistencyKind

enum `eprosima::fastdds::dds::`**TypeConsistencyKind**

Values:

enumerator **DISALLOW_TYPE_COERCION**

The *DataWriter* and the *DataReader* must support the same data type in order for them to communicate.

enumerator **ALLOW_TYPE_COERCION**

The *DataWriter* and the *DataReader* need not support the same data type in order for them to communicate as long as the reader's type is assignable from the writer's type.

UserDataQosPolicy

class **UserDataQosPolicy** : public eprosima::fastdds::dds::*GenericDataQosPolicy*
Class derived from *GenericDataQosPolicy*.

The purpose of this QoS is to allow the application to attach additional information to the created *Entity* objects such that when a remote application discovers their existence it can access that information and use it for its own purposes.

One possible use of this QoS is to attach security credentials or some other information that can be used by the remote application to authenticate the source.

WireProtocolConfigQos

class **WireProtocolConfigQos** : public eprosima::fastdds::dds::*QosPolicy*
Qos Policy that configures the wire protocol.

Public Functions

inline **WireProtocolConfigQos**()
Constructor.

virtual **~WireProtocolConfigQos**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

fastrtps::rtps::*GuidPrefix_t* **prefix**
Optionally allows user to define the GuidPrefix_t.

int32_t **participant_id**

Participant ID

By default, -1.

fastrtps::rtps::*BuiltinAttributes* **builtin**
Builtin parameters.

fastrtps::rtps::*PortParameters* **port**
Port Parameters.

fastrtps::rtps::ThroughputControllerDescriptor **throughput_controller**
Throughput controller parameters. Leave default for uncontrolled flow.

Deprecated:
Use flow_controllers() on DomainParticipantQoS

rtps::*LocatorList* **default_unicast_locator_list**
Default list of Unicast Locators to be used for any Endpoint defined inside this RTPSParticipant in the case that it was defined with NO UnicastLocators. At least ONE locator should be included in this list.

rtps::*LocatorList* **default_multicast_locator_list**
Default list of Multicast Locators to be used for any Endpoint defined inside this RTPSParticipant in the case that it was defined with NO MulticastLocators. This is usually left empty.

rtps::*ExternalLocators* **default_external_unicast_locators**
The collection of external locators to use for communication on user created topics.

bool **ignore_non_matching_locators** = false
Whether locators that don't match with the announced locators should be kept.

WriterDataLifecycleQosPolicy

class **WriterDataLifecycleQosPolicy**
Specifies the behavior of the *DataWriter* with regards to the lifecycle of the data-instances it manages.

Note: Mutable Qos Policy

Warning: This Qos Policy will be implemented in future releases.

Public Functions

inline **WriterDataLifecycleQosPolicy**()
Constructor.

inline virtual **~WriterDataLifecycleQosPolicy**()
Destructor.

Public Members

bool **autodispose_unregistered_instances**
Controls whether a *DataWriter* will automatically dispose instances each time they are unregistered. The setting autodispose_unregistered_instances = TRUE indicates that unregistered instances will also be considered disposed.

By default, true.

WriterResourceLimitsQos

class **WriterResourceLimitsQos**
Qos Policy to configure the limit of the writer resources.

Public Functions

inline **WriterResourceLimitsQos**()
Constructor.

virtual **~WriterResourceLimitsQos**() = default
Destructor.

Public Members

fastrtps::ResourceLimitedContainerConfig **matched_subscriber_allocation**
Matched subscribers allocation limits.

fastrtps::ResourceLimitedContainerConfig **reader_filters_allocation**
Reader filters allocation limits.

Status

BaseStatus

struct **BaseStatus**
A struct storing the base status.

Subclassed by eprosima::fastdds::statistics::rtps::DDSEntityStatus

Public Members

int32_t **total_count** = 0
Total cumulative count.

int32_t **total_count_change** = 0
Increment since the last time the status was read.

DeadlineMissedStatus

struct **DeadlineMissedStatus**
A struct storing the deadline status.

Subclassed by eprosima::fastdds::statistics::rtps::DDSEntityStatus

Public Functions

inline **DeadlineMissedStatus**()
Constructor.

inline **~DeadlineMissedStatus**()
Destructor.

Public Members

uint32_t **total_count**
Total cumulative number of offered deadline periods elapsed during which a writer failed to provide data.

Missed deadlines accumulate, that is, each deadline period the total_count will be incremented by 1

uint32_t **total_count_change**
The change in total_count since the last time the listener was called or the status was read.

InstanceHandle_t **last_instance_handle**
Handle to the last instance missing the deadline.

IncompatibleQosStatus

struct **IncompatibleQosStatus**
A struct storing the requested incompatible QoS status.

Subclassed by eprosima::fastdds::statistics::rtps::DDSEntityStatus

Public Members

uint32_t **total_count** = 0
Total cumulative number of times the concerned writer discovered a reader for the same topic.

The requested QoS is incompatible with the one offered by the writer

uint32_t **total_count_change** = 0
The change in total_count since the last time the listener was called or the status was read.

QosPolicyId_t **last_policy_id** = INVALID_QOS_POLICY_ID
The id of the policy that was found to be incompatible the last time an incompatibility is detected.

QosPolicyCountSeq **policies**
A list of *QosPolicyCount*.

InconsistentTopicStatus

using eprosima::fastdds::dds::**InconsistentTopicStatus** = *BaseStatus*
Alias of *BaseStatus*.

LivelinessChangedStatus

struct **LivelinessChangedStatus**
A struct storing the liveliness changed status.

Subclassed by eprosima::fastdds::statistics::rtps::DDSEntityStatus

Public Members

int32_t **alive_count** = 0
The total number of currently active publishers that write the topic read by the subscriber.

This count increases when a newly matched publisher asserts its liveliness for the first time or when a publisher previously considered to be not alive reasserts its liveliness. The count decreases when a publisher considered alive fails to assert its liveliness and becomes not alive, whether because it was deleted normally or for some other reason

int32_t **not_alive_count** = 0
The total count of current publishers that write the topic read by the subscriber that are no longer asserting their liveliness.

This count increases when a publisher considered alive fails to assert its liveliness and becomes not alive for some reason other than the normal deletion of that publisher. It decreases when a previously not alive publisher either reasserts its liveliness or is deleted normally

int32_t **alive_count_change** = 0
The change in the alive_count since the last time the listener was called or the status was read.

int32_t **not_alive_count_change** = 0
The change in the not_alive_count since the last time the listener was called or the status was read.

InstanceHandle_t **last_publication_handle**
Handle to the last publisher whose change in liveliness caused this status to change.

MatchedStatus

struct **MatchedStatus**
A structure storing a matching status.

Subclassed by *eprosima::fastdds::dds::PublicationMatchedStatus*, *eprosima::fastdds::dds::SubscriptionMatchedStatus*

Public Functions

MatchedStatus() = default
Constructor.

~MatchedStatus() = default
Destructor.

Public Members

int32_t **total_count** = 0
Total cumulative count the concerned reader discovered a match with a writer.

It found a writer for the same topic with a requested QoS that is compatible with that offered by the reader

int32_t **total_count_change** = 0
The change in total_count since the last time the listener was called or the status was read.

int32_t **current_count** = 0
The number of writers currently matched to the concerned reader.

int32_t **current_count_change** = 0
The change in current_count since the last time the listener was called or the status was read.

OfferedDeadlineMissedStatus

typedef *DeadlineMissedStatus* eprosima::fastdds::dds::**OfferedDeadlineMissedStatus**
Typedef of *DeadlineMissedStatus*.

OfferedIncompatibleQosStatus

using eprosima::fastdds::dds::**OfferedIncompatibleQosStatus** = *IncompatibleQosStatus*
Alias of *IncompatibleQosStatus*.

PublicationMatchedStatus

struct **PublicationMatchedStatus** : public eprosima::fastdds::dds::*MatchedStatus*
A structure storing the publication status.

Public Members

InstanceHandle_t **last_subscription_handle**
Handle to the last reader that matched the writer causing the status to change.

QosPolicyCount

struct **QosPolicyCount**
A struct storing the id of the incompatible QoS Policy and the number of times it fails.

Public Functions

QosPolicyCount() = default
Constructor.

inline **QosPolicyCount**(*QosPolicyId_t* id, int32_t c)
Constructor.

Public Members

QosPolicyId_t **policy_id** = INVALID_QOS_POLICY_ID
The id of the policy.

uint32_t **count** = 0
Total number of times that the concerned writer discovered a reader for the same topic.

The requested QoS is incompatible with the one offered by the writer

QosPolicyCountSeq

using eprosima::fastdds::dds::**QosPolicyCountSeq** = std::vector<*QosPolicyCount*>
Alias of std::vector<QosPolicyCount>

RequestedDeadlineMissedStatus

typedef *DeadlineMissedStatus* eprosima::fastdds::dds::**RequestedDeadlineMissedStatus**
Typedef of *DeadlineMissedStatus*.

RequestedIncompatibleQosStatus

using eprosima::fastdds::dds::**RequestedIncompatibleQosStatus** = *IncompatibleQosStatus*
Alias of *IncompatibleQosStatus*.

LivelinessLostStatus

using eprosima::fastdds::dds::**LivelinessLostStatus** = *BaseStatus*
Alias of *BaseStatus*.

SampleLostStatus

using eprosima::fastdds::dds::**SampleLostStatus** = *BaseStatus*
Alias of *BaseStatus*.

SampleRejectedStatus

struct **SampleRejectedStatus**
A struct storing the sample rejected status.

Public Members

uint32_t **total_count** = 0
Total cumulative count of samples rejected by the *DataReader*.

uint32_t **total_count_change** = 0
The incremental number of samples rejected since the last time the listener was called or the status was read.

SampleRejectedStatusKind **last_reason** = NOT_REJECTED
Reason for rejecting the last sample rejected. If no samples have been rejected, the reason is the special value NOT_REJECTED.

InstanceHandle_t **last_instance_handle**
Handle to the instance being updated by the last sample that was rejected.

SampleRejectedStatusKind

enum eprosima::fastdds::dds::**SampleRejectedStatusKind**
An enum with the possible values for the sample rejected reason.

Values:

enumerator **NOT_REJECTED**
Default value.

enumerator **REJECTED_BY_INSTANCES_LIMIT**
Exceeds the max_instance limit.

enumerator **REJECTED_BY_SAMPLES_LIMIT**
Exceeds the max_samples limit.

enumerator **REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT**
Exceeds the max_samples_per_instance limit.

StatusMask

class **StatusMask** : public std::bitset<FASTDDS_STATUS_COUNT>
StatusMask is a bitmap or bitset field.

This bitset is used to:

- determine which listener functions to call
- set conditions in dds::core::cond::StatusCondition
- indicate status changes when calling dds::core::Entity::status_changes

Public Types

typedef std::bitset<FASTDDS_STATUS_COUNT> **MaskType**
Convenience typedef for std::bitset<FASTDDS_STATUS_COUNT>.

Public Functions

inline **StatusMask**()
Construct an *StatusMask* with no flags set.

inline explicit **StatusMask**(uint32_t mask)
Construct an *StatusMask* with an uint32_t bit mask.

Parameters **mask** – the bit array to initialize the bitset with

inline *StatusMask* &**operator<<**(const *StatusMask* &mask)
Add given *StatusMask* bits into this *StatusMask* bitset.

Returns *StatusMask* this

inline *StatusMask* &**operator>>**(const *StatusMask* &mask)
Remove given *StatusMask* bits into this *StatusMask* bitset.

Returns *StatusMask* this

inline bool **is_active**(*StatusMask* status) const
Checks if the status passed as parameter is 1 in the actual *StatusMask*.

Parameters **status** – Status that need to be checked

Returns true if the status is active and false if not

Public Static Functions

static inline *StatusMask* **all**()
Get all StatusMasks

Returns *StatusMask* all

static inline *StatusMask* **none**()
Get no StatusMasks

Returns *StatusMask* none

static inline *StatusMask* **inconsistent_topic**()
Get the *StatusMask* associated with dds::core::status::InconsistentTopicStatus

Returns *StatusMask* inconsistent_topic

static inline *StatusMask* **offered_deadline_missed**()
Get the *StatusMask* associated with dds::core::status::OfferedDeadlineMissedStatus

Returns *StatusMask* offered_deadline_missed

static inline *StatusMask* **requested_deadline_missed**()
Get the *StatusMask* associated with dds::core::status::RequestedDeadlineMissedStatus

Returns *StatusMask* requested_deadline_missed

static inline *StatusMask* **offered_incompatible_qos**()
Get the *StatusMask* associated with dds::core::status::OfferedIncompatibleQosStatus

Returns *StatusMask* offered_incompatible_qos

static inline *StatusMask* **requested_incompatible_qos**()
Get the *StatusMask* associated with dds::core::status::RequestedIncompatibleQosStatus

Returns *StatusMask* requested_incompatible_qos

static inline *StatusMask* **sample_lost**()
Get the *StatusMask* associated with dds::core::status::SampleLostStatus

Returns *StatusMask* sample_lost

static inline *StatusMask* **sample_rejected**()
Get the *StatusMask* associated with dds::core::status::SampleRejectedStatus

Returns *StatusMask* sample_rejected

static inline *StatusMask* **data_on_readers**()
Get the *StatusMask* associated with dds::core::status::data_on_readers

Returns *StatusMask* data_on_readers

static inline *StatusMask* **data_available**()
get the statusmask associated with dds::core::status::data_available

Returns statusmask data_available

static inline *StatusMask* **liveliness_lost**()
Get the *StatusMask* associated with dds::core::status::LivelinessLostStatus

Returns *StatusMask* liveliness_lost

static inline *StatusMask* **liveliness_changed**()
Get the *StatusMask* associated with dds::core::status::LivelinessChangedStatus

Returns *StatusMask* liveliness_changed

static inline *StatusMask* **publication_matched**()
Get the statusmask associated with dds::core::status::PublicationMatchedStatus

Returns *StatusMask* publication_matched

static inline *StatusMask* **subscription_matched**()
Get the statusmask associated with dds::core::status::SubscriptionMatchedStatus

Returns *StatusMask* subscription_matched

FASTDDS_STATUS_COUNT size_t(16)
Alias of size_t(16)

SubscriptionMatchedStatus

struct **SubscriptionMatchedStatus** : public eprosima::fastdds::dds::*MatchedStatus*
A structure storing the subscription status.

Public Members

InstanceHandle_t **last_publication_handle**
Handle to the last writer that matched the reader causing the status change.

Condition

Condition

class **Condition**
The *Condition* class is the root base class for all the conditions that may be attached to a *WaitSet*.

Subclassed by *eprosima::fastdds::dds::GuardCondition*, *eprosima::fastdds::dds::ReadCondition*, *eprosima::fastdds::dds::StatusCondition*

Public Functions

inline virtual bool **get_trigger_value**() const
Retrieves the trigger_value of the *Condition*.

Returns true if trigger_value is set to 'true', 'false' otherwise

ConditionSeq

using eprosima::fastdds::dds::**ConditionSeq** = std::vector<*Condition**>

GuardCondition

class **GuardCondition** : public eprosima::fastdds::dds::*Condition*
The *GuardCondition* class is a specific *Condition* whose trigger_value is completely under the control of the application.

The purpose of the *GuardCondition* is to provide the means for the application to manually wakeup a *WaitSet*. This is accomplished by attaching the *GuardCondition* to the *WaitSet* and then setting the trigger_value by means of the set_trigger_value operation.

Public Functions

virtual bool **get_trigger_value**() const override
Retrieves the trigger_value of the *Condition*.

Returns true if trigger_value is set to 'true', 'false' otherwise

ReturnCode_t **set_trigger_value**(bool value)
Set the trigger_value.

Parameters **value** – new value for trigger

Returns RETURN_OK

StatusCondition

class **StatusCondition** : public eprosima::fastdds::dds::*Condition*
The *StatusCondition* class is a specific *Condition* that is associated with each *Entity*.

Public Functions

virtual bool **get_trigger_value**() const override
Retrieves the trigger_value of the *Condition*.

Returns true if trigger_value is set to 'true', 'false' otherwise

ReturnCode_t **set_enabled_statuses**(const *StatusMask* &mask)
Defines the list of communication statuses that are taken into account to determine the trigger_value.

Parameters **mask** – defines the mask for the status

Returns RETCODE_OK with everything ok, error code otherwise

const *StatusMask* &**get_enabled_statuses**() const
Retrieves the list of communication statuses that are taken into account to determine the trigger_value.

Returns Status set or default status if it has not been set

Entity ***get_entity**() const
Returns the *Entity* associated.

Returns *Entity*

Wait-set

class **WaitSet**
The *WaitSet* class allows an application to wait until one or more of the attached *Condition* objects has a trigger_value of TRUE or until timeout expires.

Public Functions

ReturnCode_t **attach_condition**(const *Condition* &cond)
Attaches a *Condition* to the Wait Set.

Parameters **cond** – *Condition*

Returns RETCODE_OK if attached correctly, error code otherwise

ReturnCode_t **detach_condition**(const *Condition* &cond)
Detaches a *Condition* from the *WaitSet*.

Parameters **cond** – *Condition*

Returns RETCODE_OK if detached correctly, PRECONDITION_NOT_MET if condition was not attached

ReturnCode_t **wait**(*ConditionSeq* &active_conditions, const fastrtps::*Duration_t* timeout) const
Allows an application thread to wait for the occurrence of certain conditions. If none of the conditions attached to the *WaitSet* have a trigger_value of true, the wait operation will block suspending the calling thread.

Parameters

- **active_conditions** – Reference to the collection of conditions which trigger_value are true
- **timeout** – Maximum time of the wait

Returns RETCODE_OK if everything correct, PRECONDITION_NOT_MET if *WaitSet* already waiting, TIMEOUT if maximum time expired, error code otherwise

ReturnCode_t **get_conditions**(*ConditionSeq* &attached_conditions) const
Retrieves the list of attached conditions.

Parameters **attached_conditions** – Reference to the collection of attached conditions

Returns RETCODE_OK if everything correct, error code otherwise

LoanableArray

template<typename **T**, std::size_t **num_items**>

struct **LoanableArray** : public std::array<*T*, *num_items*>
A type-safe, ordered collection of elements allocated on the stack, which can be loaned to a *LoanableCollection*.

Public Functions

inline void ****buffer_for_loans**() const
Get a buffer pointer that could be used on *LoanableCollection::loan*.

Returns buffer pointer for loans.

LoanableCollection

class **LoanableCollection**
A collection of generic opaque pointers that can receive the buffer from outside (loan).

This is an abstract class. See *LoanableSequence* for details.

Subclassed by eprosima::fastdds::dds::LoanableTypedCollection< T, _NonConstEnabler >, eprosima::fastdds::dds::UserAllocatedSequence, eprosima::fastdds::dds::LoanableTypedCollection< T >, eprosima::fastdds::dds::LoanableTypedCollection< T, std::true_type >

Public Functions

inline const element_type ***buffer**() const
Get the pointer to the elements buffer.

The returned value may be nullptr if *maximum()* is 0. Otherwise it is guaranteed that up to *maximum()* elements can be accessed.

Returns the pointer to the elements buffer.

inline bool **has_ownership**() const
Get the ownership flag.

Returns whether the collection has ownership of the buffer.

inline size_type **maximum**() const
Get the maximum number of elements currently allocated.

Returns the maximum number of elements currently allocated.

inline size_type **length**() const
Get the number of elements currently accessible.

Returns the number of elements currently accessible.

inline bool **length**(size_type new_length)
Set the number of elements currently accessible.

This method tells the collection that a certain number of elements should be accessible. If the new length is greater than the current *maximum()* the collection should allocate space for the new elements. If this is the case and the collection does not own the buffer (i.e. *has_ownership()* is false) then no allocation will be performed, the length will remain unchanged, and false will be returned.

Parameters **new_length** – **[in]** New number of elements to be accessible.

Pre new_length >= 0

Returns true if the new length was correctly set.

Post *length()* == new_length

Post *maximum()* >= new_length

inline bool **loan**(element_type *buffer, size_type new_maximum, size_type new_length)
Loan a buffer to the collection.

Parameters

- **buffer** – **[in]** pointer to the buffer to be loaned.
- **new_maximum** – **[in]** number of allocated elements in buffer.
- **new_length** – **[in]** number of accessible elements in buffer.

Pre (*has_ownership()* == false) || (*maximum()* == 0)

Pre new_maximum > 0

Pre new_maximum >= new_length

Pre buffer != nullptr

Returns false if preconditions are not met.

Returns true if operation succeeds.

Post *buffer()* == buffer

Post *has_ownership()* == false

Post *maximum()* == new_maximum

Post *length()* == new_length

inline element_type *__unloan__(size_type &maximum, size_type &length)
Remove the loan from the collection.

Parameters

- **maximum** – **[out]** number of allocated elements on the returned buffer.
- **length** – **[out]** number of accessible elements on the returned buffer.

Pre *has_ownership()* == false

Returns nullptr if preconditions are not met.

Returns pointer to the previously loaned buffer of elements.

Post *buffer()* == nullptr

Post *has_ownership()* == true

Post *length()* == 0

Post *maximum()* == 0

inline element_type *__unloan__()
Remove the loan from the collection.

Pre *has_ownership()* == false

Returns nullptr if preconditions are not met.

Returns pointer to the previously loaned buffer of elements.

Post *buffer()* == nullptr

Post *has_ownership()* == true

Post *length()* == 0

Post *maximum()* == 0

LoanableSequence

template<typename **T**, typename **_NonConstEnabler** = std::true_type>

class **LoanableSequence** : public eprosima::fastdds::dds::LoanableTypedCollection<*T*, std::true_type>

A type-safe, ordered collection of elements that can receive the buffer from outside (loan).

For users who define data types in OMG IDL, this type corresponds to the IDL express sequence<T>.

For any user-data type Foo that an application defines for the purpose of data-distribution with Fast DDS, a '`using FooSeq = LoanableSequence<Foo>`' is generated. The sequence offers a subset of the methods defined by the standard OMG IDL to C++ mapping for sequences. We refer to an IDL '`sequence<Foo>`' as `FooSeq`.

The state of a sequence is described by the properties 'maximum', 'length' and 'has_ownership'.

- The 'maximum' represents the size of the underlying buffer; this is the maximum number of elements it can possibly hold. It is returned by the *maximum()* operation.
- The 'length' represents the actual number of elements it currently holds. It is returned by the *length()* operation.
- The 'has_ownership' flag represents whether the sequence owns the underlying buffer. It is returned by the *has_ownership()* operation. If the sequence does not own the underlying buffer, the underlying buffer is loaned from somewhere else. This flag influences the lifecycle of the sequence and what operations are allowed on it. The general guidelines are provided below and more details are described in detail as pre-conditions and post-conditions of each of the sequence's operations:

- If has_ownership == true, the sequence has ownership on the buffer. It is then responsible for destroying the buffer when the sequence is destroyed.
- If has_ownership == false, the sequence does not have ownership on the buffer. This implies that the sequence is loaning the buffer. The sequence should not be destroyed until the loan is returned.
- A sequence with a zero maximum always has has_ownership == true

Public Functions

LoanableSequence() = default

Default constructor.

Creates the sequence with no data.

Post *buffer()* == nullptr

Post *has_ownership()* == true

Post *length()* == 0

Post *maximum()* == 0

inline **LoanableSequence**(size_type max)
Pre-allocation constructor.

Creates the sequence with an initial number of allocated elements. When the input parameter is less than or equal to 0, the behavior is equivalent to the default constructor. Otherwise, the post-conditions below will apply.

Parameters **max** – **[in]** Number of elements to pre-allocate.

Post *buffer()* != nullptr

Post *has_ownership()* == true

Post *length()* == 0

Post *maximum()* == max

inline **~LoanableSequence**()
Deallocate this sequence's buffer.

Pre *has_ownership()* == true. If this precondition is not met, no memory will be released and a warning will be logged.

Post *maximum()* == 0 and the underlying buffer is released.

inline **LoanableSequence**(const *LoanableSequence* &other)
Construct a sequence with the contents of another sequence.

This method performs a deep copy of the sequence received into this one. Allocations will happen when other.length() > 0

Parameters **other** – **[in]** The sequence from where contents are to be copied.

Post *has_ownership()* == true

Post *maximum()* == other.length()

Post *length()* == other.length()

Post *buffer()* != nullptr when other.length() > 0

inline *LoanableSequence* &**operator=**(const *LoanableSequence* &other)
Copy the contents of another sequence into this one.

This method performs a deep copy of the sequence received into this one. If this sequence had a buffer loaned, it will behave as if *unloan* has been called. Allocations will happen when (a) *has_ownership()* == false and other.length() > 0 (b) *has_ownership()* == true and other.length() > *maximum()*

Parameters **other** – **[in]** The sequence from where contents are to be copied.

Post *has_ownership()* == true

Post *maximum()* >= other.length()

Post *length()* == other.length()

Post *buffer()* != nullptr when other.length() > 0

FASTDDS_SEQUENCE(FooSeq, Foo) using FooSeq = *eprosima::fastdds::dds::LoanableSequence*<Foo>

StackAllocatedSequence

template<typename **T**, *LoanableCollection*::size_type **num_items**>

struct **StackAllocatedSequence** : public eprosima::fastdds::dds::LoanableTypedCollection<*T*>
A type-safe, ordered collection of elements allocated on the stack.

Domain

DomainParticipant

class **DomainParticipant** : public eprosima::fastdds::dds::*Entity*
Class *DomainParticipant* used to group Publishers and Subscribers into a single working unit.

Subclassed by *eprosima::fastdds::statistics::dds::DomainParticipant*

Public Functions

virtual **~DomainParticipant**()
Destructor.

ReturnCode_t **get_qos**(*DomainParticipantQos* &qos) const
This operation returns the value of the *DomainParticipant* QoS policies

Parameters **qos** – *DomainParticipantQos* reference where the qos is going to be returned

Returns RETCODE_OK

const *DomainParticipantQos* &**get_qos**() const
This operation returns the value of the *DomainParticipant* QoS policies.

Returns A reference to the *DomainParticipantQos*

ReturnCode_t **set_qos**(const *DomainParticipantQos* &qos) const
This operation sets the value of the *DomainParticipant* QoS policies.

Parameters **qos** – *DomainParticipantQos* to be set

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *DomainParticipantListener* ***get_listener**() const
Allows accessing the *DomainParticipantListener*.

Returns *DomainParticipantListener* pointer

ReturnCode_t **set_listener**(*DomainParticipantListener* *listener)
Modifies the *DomainParticipantListener*, sets the mask to *StatusMask::all()*

> **Warning:** Do not call this method from a `DomainParticipantListener` callback.

Parameters **listener** – New value for the *DomainParticipantListener*

Returns RETCODE_OK if successful, RETCODE_ERROR otherwise.

ReturnCode_t **set_listener**(*DomainParticipantListener* *listener, const std::chrono::seconds timeout)
Modifies the *DomainParticipantListener*, sets the mask to *StatusMask::all()*

> **Warning:** Do not call this method from a `DomainParticipantListener` callback.

Parameters

- **listener** – New value for the *DomainParticipantListener*
- **timeout** – Maximum time to wait for executing callbacks to finish.

Returns RETCODE_OK if successful, RETCODE_ERROR if failed (timeout expired).

ReturnCode_t **set_listener**(*DomainParticipantListener* *listener, const *StatusMask* &mask)
Modifies the *DomainParticipantListener*.

> **Warning:** Do not call this method from a `DomainParticipantListener` callback.

Parameters

- **listener** – New value for the *DomainParticipantListener*
- **mask** – *StatusMask* that holds statuses the listener responds to

Returns RETCODE_OK if successful, RETCODE_ERROR otherwise.

ReturnCode_t **set_listener**(*DomainParticipantListener* *listener, const *StatusMask* &mask, const std::chrono::seconds timeout)
Modifies the *DomainParticipantListener*.

> **Warning:** Do not call this method from a `DomainParticipantListener` callback.

Parameters

- **listener** – New value for the *DomainParticipantListener*
- **mask** – *StatusMask* that holds statuses the listener responds to
- **timeout** – Maximum time to wait for executing callbacks to finish.

Returns RETCODE_OK if successful, RETCODE_ERROR if failed (timeout expired)

virtual ReturnCode_t **enable**() override
This operation enables the *DomainParticipant*.

Returns RETCODE_OK

Publisher ***create_publisher**(const *PublisherQos* &qos, *PublisherListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())
Create a *Publisher* in this Participant.

Parameters

- **qos** – QoS of the *Publisher*.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all)

Returns Pointer to the created *Publisher*.

Publisher ***create_publisher_with_profile**(const std::string &profile_name, *PublisherListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())

Create a *Publisher* in this Participant.

Parameters

- **profile_name** – *Publisher* profile name.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all)

Returns Pointer to the created *Publisher*.

ReturnCode_t **delete_publisher**(const *Publisher* *publisher)

Deletes an existing *Publisher*.

Parameters **publisher** – to be deleted.

Returns RETCODE_PRECONDITION_NOT_MET if the publisher does not belong to this participant or if it has active DataWriters, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Subscriber ***create_subscriber**(const *SubscriberQos* &qos, *SubscriberListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())

Create a *Subscriber* in this Participant.

Parameters

- **qos** – QoS of the *Subscriber*.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all)

Returns Pointer to the created *Subscriber*.

Subscriber ***create_subscriber_with_profile**(const std::string &profile_name, *SubscriberListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())

Create a *Subscriber* in this Participant.

Parameters

- **profile_name** – *Subscriber* profile name.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all)

Returns Pointer to the created *Subscriber*.

ReturnCode_t **delete_subscriber**(const *Subscriber* *subscriber)

Deletes an existing *Subscriber*.

Parameters **subscriber** – to be deleted.

Returns RETCODE_PRECONDITION_NOT_MET if the subscriber does not belong to this participant or if it has active DataReaders, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Topic ***create_topic**(const std::string &topic_name, const std::string &type_name, const *TopicQos* &qos, *TopicListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())

Create a *Topic* in this Participant.

Parameters

- **topic_name** – Name of the *Topic*.
- **type_name** – Data type of the *Topic*.
- **qos** – QoS of the *Topic*.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all)

Returns Pointer to the created *Topic*.

Topic ***create_topic_with_profile**(const std::string &topic_name, const std::string &type_name, const std::string &profile_name, *TopicListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())

Create a *Topic* in this Participant.

Parameters

- **topic_name** – Name of the *Topic*.
- **type_name** – Data type of the *Topic*.
- **profile_name** – *Topic* profile name.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all)

Returns Pointer to the created *Topic*.

ReturnCode_t **delete_topic**(const *Topic* *topic)

Deletes an existing *Topic*.

Parameters **topic** – to be deleted.

Returns RETCODE_BAD_PARAMETER if the topic passed is a nullptr, RETCODE_PRECONDITION_NOT_MET if the topic does not belong to this participant or if it is referenced by any entity and RETCODE_OK if the *Topic* was deleted.

ContentFilteredTopic ***create_contentfilteredtopic**(const std::string &name, *Topic* *related_topic, const std::string &filter_expression, const std::vector<std::string> &expression_parameters)

Create a *ContentFilteredTopic* in this Participant.

Parameters

- **name** – Name of the *ContentFilteredTopic*
- **related_topic** – Related *Topic* to being subscribed
- **filter_expression** – Logic expression to create filter
- **expression_parameters** – Parameters to filter content

Returns Pointer to the created *ContentFilteredTopic*.

Returns nullptr if `related_topic` does not belong to this participant.

Returns nullptr if a topic with the specified `name` has already been created.

Returns nullptr if a filter cannot be created with the specified `filter_expression` and `expression_parameters`.

ContentFilteredTopic *create_contentfilteredtopic(const std::string &name, *Topic* *related_topic, const std::string &filter_expression, const std::vector<std::string> &expression_parameters, const char *filter_class_name)

Create a *ContentFilteredTopic* in this Participant using a custom filter.

Parameters

- **name** – Name of the *ContentFilteredTopic*
- **related_topic** – Related *Topic* to being subscribed
- **filter_expression** – Logic expression to create filter
- **expression_parameters** – Parameters to filter content
- **filter_class_name** – Name of the filter class to use

Returns Pointer to the created *ContentFilteredTopic*.

Returns nullptr if `related_topic` does not belong to this participant.

Returns nullptr if a topic with the specified `name` has already been created.

Returns nullptr if a filter cannot be created with the specified `filter_expression` and `expression_parameters`.

Returns nullptr if the specified `filter_class_name` has not been registered.

ReturnCode_t **delete_contentfilteredtopic**(const *ContentFilteredTopic* *a_contentfilteredtopic)

Deletes an existing *ContentFilteredTopic*.

Parameters **a_contentfilteredtopic** – *ContentFilteredTopic* to be deleted

Returns RETCODE_BAD_PARAMETER if the topic passed is a nullptr, RETCODE_PRECONDITION_NOT_MET if the topic does not belong to this participant or if it is referenced by any entity and RETCODE_OK if the *ContentFilteredTopic* was deleted.

MultiTopic *create_multitopic(const std::string &name, const std::string &type_name, const std::string &subscription_expression, const std::vector<std::string> &expression_parameters)

Create a MultiTopic in this Participant.

Parameters

- **name** – Name of the MultiTopic
- **type_name** – Result type of the MultiTopic
- **subscription_expression** – Logic expression to combine filter
- **expression_parameters** – Parameters to subscription content

Returns Pointer to the created *ContentFilteredTopic*, nullptr in error case

ReturnCode_t **delete_multitopic**(const MultiTopic *a_multitopic)

Deletes an existing MultiTopic.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters **a_multitopic** – MultiTopic to be deleted

Returns RETCODE_BAD_PARAMETER if the topic passed is a nullptr, RETCODE_PRECONDITION_NOT_MET if the topic does not belong to this participant or if it is referenced by any entity and RETCODE_OK if the *Topic* was deleted.

Topic ***find_topic**(const std::string &topic_name, const fastrtps::*Duration_t* &timeout)

Gives access to an existing (or ready to exist) enabled *Topic*. It should be noted that the returned *Topic* is a local object that acts as a proxy to designate the global concept of topic. Topics obtained by means of find_topic, must also be deleted by means of delete_topic so that the local resources can be released. If a *Topic* is obtained multiple times by means of find_topic or create_topic, it must also be deleted that same number of times using delete_topic.

Parameters

- **topic_name** – *Topic* name
- **timeout** – Maximum time to wait for the *Topic*

Returns Pointer to the existing *Topic*, nullptr in case of error or timeout

TopicDescription ***lookup_topicdescription**(const std::string &topic_name) const

Looks up an existing, locally created *TopicDescription*, based on its name. May be called on a disabled participant.

Remark

UNSAFE. It is unsafe to lookup a topic description while another thread is creating a topic.

Parameters **topic_name** – Name of the *TopicDescription* to search for.

Returns Pointer to the topic description, if it has been created locally. Otherwise, nullptr is returned.

const *Subscriber* ***get_builtin_subscriber**() const

Allows access to the builtin *Subscriber*.

Returns Pointer to the builtin *Subscriber*, nullptr in error case

ReturnCode_t **ignore_participant**(const InstanceHandle_t &handle)

Locally ignore a remote domain participant.

Note: This action is not reversible.

Parameters **handle** – Identifier of the remote participant to ignore

Returns RETURN_OK code if everything correct, RETCODE_BAD_PARAMENTER otherwise

ReturnCode_t **ignore_topic**(const InstanceHandle_t &handle)

Locally ignore a topic.

Note: This action is not reversible.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters `handle` – Identifier of the topic to ignore

Returns RETURN_OK code if everything correct, error code otherwise

ReturnCode_t **ignore_publication**(const InstanceHandle_t &handle)
Locally ignore a remote datawriter.

Note: This action is not reversible.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters `handle` – Identifier of the datawriter to ignore

Returns RETURN_OK code if everything correct, error code otherwise

ReturnCode_t **ignore_subscription**(const InstanceHandle_t &handle)
Locally ignore a remote datareader.

Note: This action is not reversible.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters `handle` – Identifier of the datareader to ignore

Returns RETURN_OK code if everything correct, error code otherwise

DomainId_t **get_domain_id**() const
This operation retrieves the domain_id used to create the *DomainParticipant*. The domain_id identifies the DDS domain to which the *DomainParticipant* belongs.

Returns The Participant's domain_id

ReturnCode_t **delete_contained_entities**()
Deletes all the entities that were created by means of the "create" methods

Returns RETURN_OK code if everything correct, error code otherwise

ReturnCode_t **assert_liveliness**()
This operation manually asserts the liveliness of the *DomainParticipant*. This is used in combination with the LIVELINESS QoS policy to indicate to the Service that the entity remains active.

This operation needs to only be used if the *DomainParticipant* contains *DataWriter* entities with the LIVELINESS set to MANUAL_BY_PARTICIPANT and it only affects the liveliness of those *DataWriter* entities. Otherwise, it has no effect.

Note: Writing data via the write operation on a *DataWriter* asserts liveliness on the *DataWriter* itself and its *DomainParticipant*. Consequently the use of assert_liveliness is only needed if the application is not

writing data regularly.

Returns RETCODE_OK if the liveliness was asserted, RETCODE_ERROR otherwise.

ReturnCode_t **set_default_publisher_qos**(const *PublisherQos* &qos)

This operation sets a default value of the *Publisher* QoS policies which will be used for newly created *Publisher* entities in the case where the QoS policies are defaulted in the create_publisher operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value PUBLISHER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_publisher_qos operation had never been called.

Parameters **qos** – *PublisherQos* to be set

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *PublisherQos* &**get_default_publisher_qos**() const

This operation retrieves the default value of the *Publisher* QoS, that is, the QoS policies which will be used for newly created *Publisher* entities in the case where the QoS policies are defaulted in the create_publisher operation.

The values retrieved get_default_publisher_qos will match the set of values specified on the last successful call to set_default_publisher_qos, or else, if the call was never made, the default values.

Returns Current default publisher qos.

ReturnCode_t **get_default_publisher_qos**(*PublisherQos* &qos) const

This operation retrieves the default value of the *Publisher* QoS, that is, the QoS policies which will be used for newly created *Publisher* entities in the case where the QoS policies are defaulted in the create_publisher operation.

The values retrieved get_default_publisher_qos will match the set of values specified on the last successful call to set_default_publisher_qos, or else, if the call was never made, the default values.

Parameters **qos** – *PublisherQos* reference where the default_publisher_qos is returned

Returns RETCODE_OK

ReturnCode_t **get_publisher_qos_from_profile**(const std::string &profile_name, *PublisherQos* &qos) const

Fills the *PublisherQos* with the values of the XML profile.

Parameters

- **profile_name** – *Publisher* profile name.
- **qos** – *PublisherQos* object where the qos is returned.

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

ReturnCode_t **set_default_subscriber_qos**(const *SubscriberQos* &qos)

This operation sets a default value of the *Subscriber* QoS policies that will be used for newly created *Subscriber* entities in the case where the QoS policies are defaulted in the create_subscriber operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value SUBSCRIBER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_subscriber_qos operation had never been called.

Parameters **qos** – *SubscriberQos* to be set

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *SubscriberQos* &**get_default_subscriber_qos**() const

This operation retrieves the default value of the *Subscriber* QoS, that is, the QoS policies which will be used for newly created *Subscriber* entities in the case where the QoS policies are defaulted in the create_subscriber operation.

The values retrieved get_default_subscriber_qos will match the set of values specified on the last successful call to set_default_subscriber_qos, or else, if the call was never made, the default values.

Returns Current default subscriber qos.

ReturnCode_t **get_default_subscriber_qos**(*SubscriberQos* &qos) const

This operation retrieves the default value of the *Subscriber* QoS, that is, the QoS policies which will be used for newly created *Subscriber* entities in the case where the QoS policies are defaulted in the create_subscriber operation.

The values retrieved get_default_subscriber_qos will match the set of values specified on the last successful call to set_default_subscriber_qos, or else, if the call was never made, the default values.

Parameters **qos** – *SubscriberQos* reference where the default_subscriber_qos is returned

Returns RETCODE_OK

ReturnCode_t **get_subscriber_qos_from_profile**(const std::string &profile_name, *SubscriberQos* &qos) const

Fills the *SubscriberQos* with the values of the XML profile.

Parameters

- **profile_name** – *Subscriber* profile name.
- **qos** – *SubscriberQos* object where the qos is returned.

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

ReturnCode_t **set_default_topic_qos**(const *TopicQos* &qos)

This operation sets a default value of the *Topic* QoS policies which will be used for newly created *Topic* entities in the case where the QoS policies are defaulted in the create_topic operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value TOPIC_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_topic_qos operation had never been called.

Parameters **qos** – *TopicQos* to be set

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *TopicQos* &**get_default_topic_qos**() const

This operation retrieves the default value of the *Topic* QoS, that is, the QoS policies that will be used for newly created *Topic* entities in the case where the QoS policies are defaulted in the create_topic operation.

The values retrieved get_default_topic_qos will match the set of values specified on the last successful call to set_default_topic_qos, or else, TOPIC_QOS_DEFAULT if the call was never made.

Returns Current default topic qos.

ReturnCode_t **get_default_topic_qos**(*TopicQos* &qos) const
This operation retrieves the default value of the *Topic* QoS, that is, the QoS policies that will be used for newly created *Topic* entities in the case where the QoS policies are defaulted in the create_topic operation.

The values retrieved get_default_topic_qos will match the set of values specified on the last successful call to set_default_topic_qos, or else, TOPIC_QOS_DEFAULT if the call was never made.

Parameters **qos** – *TopicQos* reference where the default_topic_qos is returned

Returns RETCODE_OK

ReturnCode_t **get_topic_qos_from_profile**(const std::string &profile_name, *TopicQos* &qos) const
Fills the *TopicQos* with the values of the XML profile.

Parameters

- **profile_name** – *Topic* profile name.
- **qos** – *TopicQos* object where the qos is returned.

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

ReturnCode_t **get_discovered_participants**(std::vector<InstanceHandle_t> &participant_handles) const
Retrieves the list of DomainParticipants that have been discovered in the domain and are not "ignored".

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters **participant_handles** – **[out]** Reference to the vector where discovered participants will be returned

Returns RETCODE_OK if everything correct, error code otherwise

ReturnCode_t **get_discovered_participant_data**(builtin::ParticipantBuiltinTopicData &participant_data, const InstanceHandle_t &participant_handle) const
Retrieves the *DomainParticipant* data of a discovered not ignored participant.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- **participant_data** – **[out]** Reference to the ParticipantBuiltinTopicData object to return the data
- **participant_handle** – InstanceHandle of *DomainParticipant* to retrieve the data from

Returns RETCODE_OK if everything correct, PRECONDITION_NOT_MET if participant does not exist

ReturnCode_t **get_discovered_topics**(std::vector<InstanceHandle_t> &topic_handles) const
Retrieves the list of topics that have been discovered in the domain and are not "ignored".

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters **topic_handles** – **[out]** Reference to the vector where discovered topics will be returned

Returns RETCODE_OK if everything correct, error code otherwise

ReturnCode_t **get_discovered_topic_data**(builtin::TopicBuiltinTopicData &topic_data, const InstanceHandle_t &topic_handle) const

Retrieves the *Topic* data of a discovered not ignored topic.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- **topic_data** – **[out]** Reference to the TopicBuiltinTopicData object to return the data
- **topic_handle** – InstanceHandle of *Topic* to retrieve the data from

Returns RETCODE_OK if everything correct, PRECONDITION_NOT_MET if topic does not exist

bool **contains_entity**(const InstanceHandle_t &a_handle, bool recursive = true) const

This operation checks whether or not the given handle represents an *Entity* that was created from the *DomainParticipant*.

Parameters

- **a_handle** – InstanceHandle of the entity to look for.
- **recursive** – The containment applies recursively. That is, it applies both to entities (*TopicDescription*, *Publisher*, or *Subscriber*) created directly using the *DomainParticipant* as well as entities created using a contained *Publisher*, or *Subscriber* as the factory, and so forth. (default: true)

Returns True if entity is contained. False otherwise.

ReturnCode_t **get_current_time**(fastrtps::*Time_t* &current_time) const

This operation returns the current value of the time that the service uses to time-stamp data-writes and to set the reception-timestamp for the data-updates it receives.

Parameters **current_time** – Time_t reference where the current time is returned

Returns RETCODE_OK

ReturnCode_t **register_type**(*TypeSupport* type, const std::string &type_name)

Register a type in this participant.

Parameters

- **type** – *TypeSupport*.
- **type_name** – The name that will be used to identify the Type.

Returns RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there is another *TypeSupport* with the same name and RETCODE_OK if it is correctly registered.

ReturnCode_t **register_type**(*TypeSupport* type)
Register a type in this participant.

Parameters **type** – *TypeSupport*.

Returns RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there is another *TypeSupport* with the same name and RETCODE_OK if it is correctly registered.

ReturnCode_t **unregister_type**(const std::string &typeName)
Unregister a type in this participant.

Parameters **typeName** – Name of the type

Returns RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there are entities using that *TypeSupport* and RETCODE_OK if it is correctly unregistered.

TypeSupport **find_type**(const std::string &type_name) const
This method gives access to a registered type based on its name.

Parameters **type_name** – Name of the type

Returns *TypeSupport* corresponding to the type_name

const InstanceHandle_t &**get_instance_handle**() const
Returns the *DomainParticipant*'s handle.

Returns InstanceHandle of this *DomainParticipant*.

const fastrtps::rtps::*GUID_t* &**guid**() const
Getter for the Participant GUID.

Returns A reference to the GUID

std::vector<std::string> **get_participant_names**() const
Getter for the participant names.

Returns Vector with the names

bool **new_remote_endpoint_discovered**(const fastrtps::rtps::*GUID_t* &partguid, uint16_t userId, fastrtps::rtps::*EndpointKind_t* kind)
This method can be used when using a StaticEndpointDiscovery mechanism different that the one included in FastRTPS, for example when communicating with other implementations. It indicates the Participant that an Endpoint from the XML has been discovered and should be activated.

Parameters

- **partguid** – Participant GUID_t.
- **userId** – User defined ID as shown in the XML file.
- **kind** – EndpointKind (WRITER or READER)

Returns True if correctly found and activated.

fastrtps::rtps::ResourceEvent &**get_resource_event**() const
Getter for the resource event.

Pre The *DomainParticipant* is enabled.

Returns A reference to the resource event

fastrtps::rtps::*SampleIdentity* **get_type_dependencies**(const fastrtps::types::TypeIdentifierSeq &in) const
When a *DomainParticipant* receives an incomplete list of TypeIdentifiers in a PublicationBuiltinTopicData or SubscriptionBuiltinTopicData, it may request the additional type dependencies by invoking the getTypeDependencies operation.

Parameters **in** – TypeIdentifier sequence

Returns SampleIdentity

fastrtps::rtps::*SampleIdentity* **get_types**(const fastrtps::types::TypeIdentifierSeq &in) const
A *DomainParticipant* may invoke the operation getTypes to retrieve the TypeObjects associated with a list of TypeIdentifiers.

Parameters **in** – TypeIdentifier sequence

Returns SampleIdentity

ReturnCode_t **register_remote_type**(const fastrtps::types::TypeInformation &type_information, const std::string &type_name, std::function<void(const std::string &name, const fastrtps::types::DynamicType_ptr type)> &callback)
Helps the user to solve all dependencies calling internally to the type lookup service and registers the resulting dynamic type. The registration may be perform asynchronously, case in which the user will be notified through the given callback, which receives the type_name as unique argument.

Parameters

- **type_information** –
- **type_name** –
- **callback** –

Returns RETCODE_OK If the given type_information is enough to build the type without using the typelookup service (callback will not be called).

Returns RETCODE_OK if the given type is already available (callback will not be called).

Returns RETCODE_NO_DATA if type is not available yet (the callback will be called if negotiation is success, and ignored in other case).

Returns RETCODE_NOT_ENABLED if the *DomainParticipant* is not enabled.

Returns RETCODE_PRECONDITION_NOT_MET if the *DomainParticipant* type lookup service is disabled.

ReturnCode_t **register_content_filter_factory**(const char *filter_class_name, *IContentFilterFactory* *const filter_factory)
Register a custom content filter factory, which can be used to create a *ContentFilteredTopic*.

DDS specifies a SQL-like content filter to be used by content filtered topics. If this filter does not meet your filtering requirements, you can register a custom filter factory.

To use a custom filter, a factory for it must be registered in the following places:

- In any application that uses the custom filter factory to create a *ContentFilteredTopic* and the corresponding *DataReader*.
- In each application that writes the data to the applications mentioned above.

For example, suppose Application A on the subscription side creates a *Topic* named X and a *ContentFilteredTopic* named filteredX (and a corresponding *DataReader*), using a previously registered content filter factory, myFilterFactory. With only that, you will have filtering at the subscription side. If you also want to

perform filtering in any application that publishes *Topic* X, then you also need to register the same definition of the ContentFilterFactory myFilterFactory in that application.

Each `filter_class_name` can only be used to register a content filter factory once per *DomainParticipant*.

Parameters

- **filter_class_name** – Name of the filter class. Cannot be nullptr, must not exceed 255 characters, and must be unique within this *DomainParticipant*.
- **filter_factory** – Factory of content filters to be registered. Cannot be nullptr.

Returns RETCODE_BAD_PARAMETER if any parameter is nullptr, or the filter_class_name exceeds 255 characters.

Returns RETCODE_PRECONDITION_NOT_MET if the filter_class_name has been already registered.

Returns RETCODE_PRECONDITION_NOT_MET if filter_class_name is FASTDDS_SQLFILTER_NAME.

Returns RETCODE_OK if the filter is correctly registered.

IContentFilterFactory ***lookup_content_filter_factory**(const char *filter_class_name)

Lookup a custom content filter factory previously registered with register_content_filter_factory.

Parameters **filter_class_name** – Name of the filter class. Cannot be nullptr.

Returns nullptr if the given filter_class_name has not been previously registered on this *DomainParticipant*. Otherwise, the content filter factory previously registered with the given filter_class_name.

ReturnCode_t **unregister_content_filter_factory**(const char *filter_class_name)

Unregister a custom content filter factory previously registered with register_content_filter_factory.

A filter_class_name can be unregistered only if it has been previously registered to the *DomainParticipant* with register_content_filter_factory.

The unregistration of filter is not allowed if there are any existing *ContentFilteredTopic* objects that are using the filter.

If there is any existing discovered *DataReader* with the same filter_class_name, filtering on the writer side will be stopped, but this operation will not fail.

Parameters **filter_class_name** – Name of the filter class. Cannot be nullptr.

Returns RETCODE_BAD_PARAMETER if the filter_class_name is nullptr.

Returns RERCODE_PRECONDITION_NOT_MET if the filter_class_name has not been previously registered.

Returns RERCODE_PRECONDITION_NOT_MET if there is any *ContentFilteredTopic* referencing the filter.

Returns RETCODE_OK if the filter is correctly unregistered.

bool **has_active_entities**()

Check if the Participant has any *Publisher*, *Subscriber* or *Topic*.

Returns true if any, false otherwise.

DomainParticipantFactory

class **DomainParticipantFactory**
Class *DomainParticipantFactory*

Public Functions

DomainParticipant ***create_participant**(DomainId_t domain_id, const *DomainParticipantQos* &qos, *DomainParticipantListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())
Create a Participant.

Parameters

- **domain_id** – Domain Id.
- **qos** – *DomainParticipantQos* Reference.
- **listener** – *DomainParticipantListener* Pointer (default: nullptr)
- **mask** – *StatusMask* Reference (default: all)

Returns *DomainParticipant* pointer. (nullptr if not created.)

DomainParticipant ***create_participant_with_profile**(DomainId_t domain_id, const std::string &profile_name, *DomainParticipantListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())
Create a Participant.

Parameters

- **domain_id** – Domain Id.
- **profile_name** – Participant profile name.
- **listener** – *DomainParticipantListener* Pointer (default: nullptr)
- **mask** – *StatusMask* Reference (default: all)

Returns *DomainParticipant* pointer. (nullptr if not created.)

DomainParticipant ***create_participant_with_profile**(const std::string &profile_name, *DomainParticipantListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*())
Create a Participant.

Parameters

- **profile_name** – Participant profile name.
- **listener** – *DomainParticipantListener* Pointer (default: nullptr)
- **mask** – *StatusMask* Reference (default: all)

Returns *DomainParticipant* pointer. (nullptr if not created.)

DomainParticipant ***lookup_participant**(DomainId_t domain_id) const
This operation retrieves a previously created *DomainParticipant* belonging to specified domain_id. If no such *DomainParticipant* exists, the operation will return 'nullptr'. If multiple *DomainParticipant* entities belonging to that domain_id exist, then the operation will return one of them. It is not specified which one.

Parameters **domain_id** –

Returns previously created *DomainParticipant* within the specified domain

std::vector<*DomainParticipant**> **lookup_participants**(DomainId_t domain_id) const
Returns all participants that belongs to the specified domain_id.

Parameters **domain_id** –

Returns previously created DomainParticipants within the specified domain

ReturnCode_t **get_default_participant_qos**(*DomainParticipantQos* &qos) const
This operation retrieves the default value of the *DomainParticipant* QoS, that is, the QoS policies which will be used for newly created *DomainParticipant* entities in the case where the QoS policies are defaulted in the create_participant operation. The values retrieved get_default_participant_qos will match the set of values specified on the last successful call to set_default_participant_qos, or else, if the call was never made, the default values.

Parameters **qos** – *DomainParticipantQos* where the qos is returned

Returns RETCODE_OK

const *DomainParticipantQos* &**get_default_participant_qos**() const
This operation retrieves the default value of the *DomainParticipant* QoS, that is, the QoS policies which will be used for newly created *DomainParticipant* entities in the case where the QoS policies are defaulted in the create_participant operation. The values retrieved get_default_participant_qos will match the set of values specified on the last successful call to set_default_participant_qos, or else, if the call was never made, the default values.

Returns A reference to the default *DomainParticipantQos*

ReturnCode_t **set_default_participant_qos**(const *DomainParticipantQos* &qos)
This operation sets a default value of the *DomainParticipant* QoS policies which will be used for newly created *DomainParticipant* entities in the case where the QoS policies are defaulted in the create_participant operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value PARTICIPANT_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_participant_qos operation had never been called.

Parameters **qos** – *DomainParticipantQos* to be set

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

ReturnCode_t **get_participant_qos_from_profile**(const std::string &profile_name,
DomainParticipantQos &qos) const
Fills the *DomainParticipantQos* with the values of the XML profile.

Parameters

- **profile_name** – *DomainParticipant* profile name.
- **qos** – *DomainParticipantQos* object where the qos is returned.

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

ReturnCode_t **delete_participant**(*DomainParticipant* *part)
Remove a Participant and all associated publishers and subscribers.

Parameters **part** – Pointer to the participant.

Returns RETCODE_PRECONDITION_NOT_MET if the participant has active entities, RETCODE_OK if the participant is correctly deleted and RETCODE_ERROR otherwise.

ReturnCode_t **load_profiles**()
Load profiles from default XML file.

Returns RETCODE_OK

ReturnCode_t **load_XML_profiles_file**(const std::string &xml_profile_file)
Load profiles from XML file.

Parameters **xml_profile_file** – XML profile file.

Returns RETCODE_OK if it is correctly loaded, RETCODE_ERROR otherwise.

ReturnCode_t **load_XML_profiles_string**(const char *data, size_t length)
Load profiles from XML string.

Parameters

- **data** – buffer containing xml data.
- **length** – length of data

Returns RETCODE_OK if it is correctly loaded, RETCODE_ERROR otherwise.

ReturnCode_t **check_xml_static_discovery**(std::string &xml_file)
Check the validity of the provided static discovery XML file

Parameters **xml_file** – xml file path

Returns RETCODE_OK if the validation is successful, RETCODE_ERROR otherwise.

ReturnCode_t **get_qos**(*DomainParticipantFactoryQos* &qos) const
This operation returns the value of the *DomainParticipantFactory* QoS policies.

Parameters **qos** – DomaParticipantFactoryQos reference where the qos is returned

Returns RETCODE_OK

ReturnCode_t **set_qos**(const *DomainParticipantFactoryQos* &qos)
This operation sets the value of the *DomainParticipantFactory* QoS policies. These policies control the behavior of the object a factory for entities.

Note that despite having QoS, the *DomainParticipantFactory* is not an *Entity*.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

Parameters **qos** – *DomainParticipantFactoryQos* to be set.

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Public Static Functions

static *DomainParticipantFactory* ***get_instance**()
Returns the *DomainParticipantFactory* singleton instance.

Returns A raw pointer to the *DomainParticipantFactory* singleton instance.

static std::shared_ptr<*DomainParticipantFactory*> **get_shared_instance**()
Returns the *DomainParticipantFactory* singleton instance.

Returns A shared pointer to the *DomainParticipantFactory* singleton instance.

DomainParticipantFactoryQos

class **DomainParticipantFactoryQos**
Class *DomainParticipantFactoryQos*, contains all the possible Qos that can be set for a determined participant. Please consult each of them to check for implementation details and default values.

Public Functions

inline **DomainParticipantFactoryQos**()
Constructor.

inline virtual **~DomainParticipantFactoryQos**()
Destructor.

inline const *EntityFactoryQosPolicy* &**entity_factory**() const
Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline *EntityFactoryQosPolicy* &**entity_factory**()
Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline void **entity_factory**(const *EntityFactoryQosPolicy* &entity_factory)
Setter for *EntityFactoryQosPolicy*

Parameters **entity_factory** – *EntityFactoryQosPolicy*

inline rtps::ThreadSettings &**shm_watchdog_thread**()
Getter for SHM watchdog ThreadSettings

Returns rtps::ThreadSettings reference

inline const rtps::ThreadSettings &**shm_watchdog_thread**() const
Getter for SHM watchdog ThreadSettings

Returns rtps::ThreadSettings reference

inline void **shm_watchdog_thread**(const rtps::ThreadSettings &value)
Setter for the SHM watchdog ThreadSettings

Parameters **value** – New ThreadSettings to be set

inline rtps::ThreadSettings &**file_watch_threads**()
Getter for file watch related ThreadSettings

Returns rtps::ThreadSettings reference

inline const rtps::ThreadSettings &**file_watch_threads**() const
Getter for file watch related ThreadSettings

Returns rtps::ThreadSettings reference

inline void **file_watch_threads**(const rtps::ThreadSettings &value)
Setter for the file watch related ThreadSettings

Parameters **value** – New ThreadSettings to be set

DomainParticipantListener

class **DomainParticipantListener** : public eprosima::fastdds::dds::*PublisherListener*, public eprosima::fastdds::dds::*SubscriberListener*, public eprosima::fastdds::dds::*TopicListener*
Class *DomainParticipantListener*, overrides behaviour towards certain events.

Public Functions

inline **DomainParticipantListener**()
Constructor.

inline virtual **~DomainParticipantListener**()
Destructor.

inline virtual void **on_participant_discovery**(*DomainParticipant* *participant, fastrtps::rtps::*ParticipantDiscoveryInfo* &&info, bool &should_be_ignored)
This method is called when a new Participant is discovered, or a previously discovered participant changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote participant.
- **info** – **[out]** Remote participant information. User can take ownership of the object.
- **should_be_ignored** – **[out]** Flag to indicate the library to automatically ignore the discovered Participant.

inline virtual void **onParticipantAuthentication**(*DomainParticipant* *participant, fastrtps::rtps::*ParticipantAuthenticationInfo* &&info)
This method is called when a new Participant is authenticated.

Parameters

- **participant** – **[out]** Pointer to the authenticated Participant.
- **info** – **[out]** Remote participant authentication information. User can take ownership of the object.

inline virtual void **on_subscriber_discovery**(*DomainParticipant* *participant, fastrtps::rtps::*ReaderDiscoveryInfo* &&info)
This method is called when a new *DataReader* is discovered, or a previously discovered *DataReader* changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote *DataReader*.
- **info** – **[out]** Remote *DataReader* information. User can take ownership of the object.

inline virtual void **on_subscriber_discovery**(*DomainParticipant* *participant, fastrtps::rtps::*ReaderDiscoveryInfo* &&info, bool &should_be_ignored)

This method is called when a new *DataReader* is discovered, or a previously discovered *DataReader* changes its QOS or is removed.

> **Warning:** Not Supported. This callback will never be called in the current version.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote *DataReader*.
- **info** – **[out]** Remote *DataReader* information. User can take ownership of the object.
- **should_be_ignored** – **[out]** Flag to indicate the library to automatically ignore the discovered *DataReader*.

inline virtual void **on_publisher_discovery**(*DomainParticipant* *participant, fastrtps::rtps::*WriterDiscoveryInfo* &&info)

This method is called when a new *DataWriter* is discovered, or a previously discovered *DataWriter* changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote *DataWriter*.
- **info** – **[out]** Remote *DataWriter* information. User can take ownership of the object.

inline virtual void **on_publisher_discovery**(*DomainParticipant* *participant, fastrtps::rtps::*WriterDiscoveryInfo* &&info, bool &should_be_ignored)

This method is called when a new *DataWriter* is discovered, or a previously discovered *DataWriter* changes its QOS or is removed.

> **Warning:** Not Supported. This callback will never be called in the current version.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote *DataWriter*.
- **info** – **[out]** Remote *DataWriter* information. User can take ownership of the object.
- **should_be_ignored** – **[out]** Flag to indicate the library to automatically ignore the discovered *DataWriter*.

inline virtual void **on_type_discovery**(*DomainParticipant* *participant, const fastrtps::rtps::*SampleIdentity* &request_sample_id, const fastrtps::string_255 &topic, const fastrtps::types::TypeIdentifier *identifier, const fastrtps::types::TypeObject *object, fastrtps::types::DynamicType_ptr dyn_type)

This method is called when a participant discovers a new Type The ownership of all object belongs to the caller so if needs to be used after the method ends, a full copy should be perform (except for dyn_type due to its shared_ptr nature.

For example: fastrtps::types::TypeIdentifier new_type_id = *identifier;

inline virtual void **on_type_dependencies_reply**(*DomainParticipant* *participant, const fastrtps::rtps::*SampleIdentity* &request_sample_id, const fastrtps::types::TypeIdentifierWithSizeSeq &dependencies)

This method is called when the typelookup client received a reply to a getTypeDependencies request.

The user may want to retrieve these new types using the getTypes request and create a new DynamicType using the retrieved TypeObject.

inline virtual void **on_type_information_received**(*DomainParticipant* *participant, const fastrtps::string_255 topic_name, const fastrtps::string_255 type_name, const fastrtps::types::TypeInformation &type_information)

This method is called when a participant receives a TypeInformation while discovering another participant.

DomainParticipantQos

class **DomainParticipantQos**

Class *DomainParticipantQos*, contains all the possible Qos that can be set for a determined participant. Please consult each of them to check for implementation details and default values.

Public Types

using **FlowControllerDescriptorList** = std::vector<std::shared_ptr<fastdds::rtps::*FlowControllerDescriptor*>>

User defined flow controllers to use alongside.

Since 2.4.0

Public Functions

inline **DomainParticipantQos**()

Constructor.

inline virtual **~DomainParticipantQos**()

Destructor.

inline const *UserDataQosPolicy* &**user_data**() const

Getter for *UserDataQosPolicy*

Returns *UserDataQosPolicy* reference

inline *UserDataQosPolicy* &**user_data**()

Getter for *UserDataQosPolicy*

Returns *UserDataQosPolicy* reference

inline void **user_data**(const *UserDataQosPolicy* &value)

Setter for *UserDataQosPolicy*

Parameters **value** – *UserDataQosPolicy*

inline const *EntityFactoryQosPolicy* &**entity_factory**() const

Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline *EntityFactoryQosPolicy* &**entity_factory**()
Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline void **entity_factory**(const *EntityFactoryQosPolicy* &value)
Setter for *EntityFactoryQosPolicy*

Parameters **value** – *EntityFactoryQosPolicy*

inline const *ParticipantResourceLimitsQos* &**allocation**() const
Getter for ParticipantResourceLimitsQos

Returns ParticipantResourceLimitsQos reference

inline *ParticipantResourceLimitsQos* &**allocation**()
Getter for ParticipantResourceLimitsQos

Returns ParticipantResourceLimitsQos reference

inline void **allocation**(const *ParticipantResourceLimitsQos* &allocation)
Setter for ParticipantResourceLimitsQos

Parameters **allocation** – ParticipantResourceLimitsQos

inline const *PropertyPolicyQos* &**properties**() const
Getter for PropertyPolicyQos

Returns PropertyPolicyQos reference

inline *PropertyPolicyQos* &**properties**()
Getter for PropertyPolicyQos

Returns PropertyPolicyQos reference

inline void **properties**(const *PropertyPolicyQos* &properties)
Setter for PropertyPolicyQos

Parameters **properties** – PropertyPolicyQos

inline const *WireProtocolConfigQos* &**wire_protocol**() const
Getter for *WireProtocolConfigQos*

Returns *WireProtocolConfigQos* reference

inline *WireProtocolConfigQos* &**wire_protocol**()
Getter for *WireProtocolConfigQos*

Returns *WireProtocolConfigQos* reference

inline void **wire_protocol**(const *WireProtocolConfigQos* &wire_protocol)
Setter for *WireProtocolConfigQos*

Parameters **wire_protocol** – *WireProtocolConfigQos*

inline const *TransportConfigQos* &**transport**() const
Getter for *TransportConfigQos*

Returns *TransportConfigQos* reference

inline *TransportConfigQos* &**transport**()
Getter for *TransportConfigQos*

Returns *TransportConfigQos* reference

inline void **transport**(const *TransportConfigQos* &transport)
Setter for *TransportConfigQos*

Parameters **transport** – *TransportConfigQos*

inline const fastrtps::string_255 &**name**() const
Getter for the Participant name

Returns name

inline fastrtps::string_255 &**name**()
Getter for the Participant name

Returns name

inline void **name**(const fastrtps::string_255 &value)
Setter for the Participant name

Parameters **value** – New name to be set

inline *FlowControllerDescriptorList* &**flow_controllers**()
Getter for FlowControllerDescriptorList

Returns FlowControllerDescriptorList reference

inline const *FlowControllerDescriptorList* &**flow_controllers**() const
Getter for FlowControllerDescriptorList

Returns FlowControllerDescriptorList reference

inline rtps::ThreadSettings &**builtin_controllers_sender_thread**()
Getter for builtin flow controllers sender threads ThreadSettings

Returns rtps::ThreadSettings reference

inline const rtps::ThreadSettings &**builtin_controllers_sender_thread**() const
Getter for builtin flow controllers sender threads ThreadSettings

Returns rtps::ThreadSettings reference

void **setup_transports**(rtps::BuiltinTransports transports, const rtps::BuiltinTransportsOptions &options =
rtps::BuiltinTransportsOptions())
Provides a way of easily configuring transport related configuration on certain pre-defined scenarios with certain options.

Parameters

- **transports** – Defines the transport configuration scenario to setup.
- **options** – Defines the options to be used in the transport configuration.

inline void **builtin_controllers_sender_thread**(const rtps::ThreadSettings &value)
Setter for the builtin flow controllers sender threads ThreadSettings

Parameters **value** – New ThreadSettings to be set

inline rtps::ThreadSettings &**timed_events_thread**()
Getter for timed event ThreadSettings

Returns rtps::ThreadSettings reference

inline const rtps::ThreadSettings &**timed_events_thread**() const
Getter for timed event ThreadSettings

Returns rtps::ThreadSettings reference

inline void **timed_events_thread**(const rtps::ThreadSettings &value)
Setter for the timed event ThreadSettings

Parameters **value** – New ThreadSettings to be set

inline rtps::ThreadSettings &**discovery_server_thread**()
Getter for discovery server ThreadSettings

Returns rtps::ThreadSettings reference

inline const rtps::ThreadSettings &**discovery_server_thread**() const
Getter for discovery server ThreadSettings

Returns rtps::ThreadSettings reference

inline void **discovery_server_thread**(const rtps::ThreadSettings &value)
Setter for the discovery server ThreadSettings

Parameters **value** – New ThreadSettings to be set

inline rtps::ThreadSettings &**security_log_thread**()
Getter for security log ThreadSettings

Returns rtps::ThreadSettings reference

inline const rtps::ThreadSettings &**security_log_thread**() const
Getter for security log ThreadSettings

Returns rtps::ThreadSettings reference

inline void **security_log_thread**(const rtps::ThreadSettings &value)
Setter for the security log ThreadSettings

Parameters **value** – New ThreadSettings to be set

const *DomainParticipantQos* eprosima::fastdds::dds::**PARTICIPANT_QOS_DEFAULT**

Publisher

DataWriter

class **DataWriter** : public eprosima::fastdds::dds::*DomainEntity*
Class *DataWriter*, contains the actual implementation of the behaviour of the *DataWriter*.

Public Types

enum class **LoanInitializationKind**
How to initialize samples loaned with *loan_sample*

Values:

enumerator **NO_LOAN_INITIALIZATION**
Do not perform initialization of sample.

This is the default initialization scheme of loaned samples. It is the fastest scheme, but implies the user should take care of writing every field on the data type before calling *write* on the loaned sample.

enumerator **ZERO_LOAN_INITIALIZATION**
Initialize all memory with zero-valued bytes.

The contents of the loaned sample will be zero-initialized upon return of *loan_sample*.

enumerator **CONSTRUCTED_LOAN_INITIALIZATION**
Use in-place constructor initialization.

This will call the constructor of the data type over the memory space being returned by *loan_sample*.

Public Functions

virtual ReturnCode_t **enable**() override
This operation enables the *DataWriter*.

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the *Publisher* creating this *DataWriter* is not enabled.

bool **write**(void *data)
Write data to the topic.

Parameters **data** – Pointer to the data

Returns True if correct, false otherwise

bool **write**(void *data, fastrtps::rtps::*WriteParams* &params)
Write data with params to the topic.

Parameters

- **data** – Pointer to the data
- **params** – Extra write parameters.

Returns True if correct, false otherwise

ReturnCode_t **write**(void *data, const InstanceHandle_t &handle)
Write data with handle.

The special value HANDLE_NIL can be used for the parameter handle.This indicates that the identity of the instance should be automatically deduced from the instance_data (by means of the key).

Parameters

- **data** – Pointer to the data
- **handle** – InstanceHandle_t.

Returns RETCODE_PRECONDITION_NOT_MET if the handle introduced does not match with the one associated to the data, RETCODE_OK if the data is correctly sent and RETCODE_ERROR otherwise.

ReturnCode_t **write_w_timestamp**(void *data, const InstanceHandle_t &handle, const fastrtps::*Time_t* &timestamp)
This operation performs the same function as write except that it also provides the value for the *source_timestamp* that is made available to *DataReader* objects by means of the *eprosima::fastdds::dds::SampleInfo::source_timestamp* attribute "source_timestamp" inside the *SampleInfo*. The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the *write* operation. This operation may block and return RETCODE_TIMEOUT under the same circumstances described for the *write* operation. This operation

may return RETCODE_OUT_OF_RESOURCES, RETCODE_PRECONDITION_NOT_MET or RETCODE_BAD_PARAMETER under the same circumstances described for the write operation.

Parameters

- **data** – Pointer to the data
- **handle** – InstanceHandle_t
- **timestamp** – Time_t used to set the source_timestamp.

Returns Any of the standard return codes.

InstanceHandle_t **register_instance**(void *instance)

Informs that the application will be modifying a particular instance.

It gives an opportunity to the middleware to pre-configure itself to improve performance.

Parameters **instance** – **[in]** Sample used to get the instance's key.

Returns Handle containing the instance's key. This handle could be used in successive `write` or `dispose` operations. In case of error, HANDLE_NIL will be returned.

InstanceHandle_t **register_instance_w_timestamp**(void *instance, const fastrtps::*Time_t* &timestamp)

This operation performs the same function as register_instance and can be used instead of *register_instance* in the cases where the application desires to specify the value for the *source_timestamp*. The *source_timestamp* potentially affects the relative order in which readers observe events from multiple writers. See the QoS policy *DESTINATION_ORDER*.

This operation may block and return RETCODE_TIMEOUT under the same circumstances described for the *write* operation.

This operation may return RETCODE_OUT_OF_RESOURCES under the same circumstances described for the *write* operation.

Parameters

- **instance** – Sample used to get the instance's key.
- **timestamp** – Time_t used to set the source_timestamp.

Returns Handle containing the instance's key.

ReturnCode_t **unregister_instance**(void *instance, const InstanceHandle_t &handle)

This operation reverses the action of `register_instance`.

It should only be called on an instance that is currently registered. Informs the middleware that the *DataWriter* is not intending to modify any more of that data instance. Also indicates that the middleware can locally remove all information regarding that instance.

Parameters

- **instance** – **[in]** Sample used to deduce instance's key in case of `handle` parameter is HANDLE_NIL.
- **handle** – **[in]** Instance's key to be unregistered.

Returns Returns the operation's result. If the operation finishes successfully, ReturnCode_t::RETCODE_OK is returned.

ReturnCode_t **unregister_instance_w_timestamp**(void *instance, const InstanceHandle_t &handle, const fastrtps::*Time_t* &timestamp)

This operation performs the same function as *unregister_instance* and can be used instead of *unregister_instance* in the cases where the application desires to specify the value for the *source_timestamp*. The

source_timestamp potentially affects the relative order in which readers observe events from multiple writers. See the QoS policy *DESTINATION_ORDER*.

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the *unregister_instance* operation.

This operation may block and return RETCODE_TIMEOUT under the same circumstances described for the write operation

Parameters

- **instance** – Sample used to deduce instance's key in case of `handle` parameter is HANDLE_NIL.
- **handle** – Instance's key to be unregistered.
- **timestamp** – Time_t used to set the source_timestamp.

Returns Handle containing the instance's key.

ReturnCode_t **get_key_value**(void *key_holder, const InstanceHandle_t &handle)
This operation can be used to retrieve the instance key that corresponds to an instance_handle. The operation will only fill the fields that form the key inside the key_holder instance.

This operation may return BAD_PARAMETER if the InstanceHandle_t handle does not correspond to an existing data-object known to the *DataWriter*. If the implementation is not able to check invalid handles then the result in this situation is unspecified.

Parameters

- **key_holder** – **[inout]** Sample where the key fields will be returned.
- **handle** – **[in]** Handle to the instance to retrieve the key values from.

Returns Any of the standard return codes.

InstanceHandle_t **lookup_instance**(const void *instance) const
NOT YET IMPLEMENTED

Takes as a parameter an instance and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The instance parameter is only used for the purpose of examining the fields that define the key.

Parameters **instance** – **[in]** Data pointer to the sample

Returns handle of the given instance

const fastrtps::rtps::*GUID_t* &**guid**() const
Returns the *DataWriter*'s GUID

Returns Reference to the *DataWriter* GUID

InstanceHandle_t **get_instance_handle**() const
Returns the *DataWriter*'s InstanceHandle

Returns Copy of the *DataWriter* InstanceHandle

TypeSupport **get_type**() const
Get data type associated to the *DataWriter*

Returns Copy of the *TypeSupport*

ReturnCode_t **wait_for_acknowledgments**(const fastrtps::*Duration_t* &max_wait)
Waits the current thread until all writers have received their acknowledgments.

Parameters **max_wait** – Maximum blocking time for this operation

Returns RETCODE_OK if the *DataWriter* receive the acknowledgments before the time expires and RETCODE_ERROR otherwise

ReturnCode_t **get_offered_deadline_missed_status**(*OfferedDeadlineMissedStatus* &status)
Returns the offered deadline missed status.

Parameters **status** – **[out]** Deadline missed status struct

Returns RETCODE_OK

ReturnCode_t **get_offered_incompatible_qos_status**(*OfferedIncompatibleQosStatus* &status)
Returns the offered incompatible qos status.

Parameters **status** – **[out]** Offered incompatible qos status struct

Returns RETCODE_OK

ReturnCode_t **get_publication_matched_status**(*PublicationMatchedStatus* &status) const
Returns the publication matched status.

Parameters **status** – **[out]** publication matched status struct

Returns RETCODE_OK

ReturnCode_t **set_qos**(const *DataWriterQos* &qos)
Establishes the *DataWriterQos* for this *DataWriter*.

Parameters **qos** – *DataWriterQos* to be set

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *DataWriterQos* &**get_qos**() const
Retrieves the *DataWriterQos* for this *DataWriter*.

Returns Reference to the current *DataWriterQos*

ReturnCode_t **get_qos**(*DataWriterQos* &qos) const
Fills the *DataWriterQos* with the values of this *DataWriter*.

Parameters **qos** – *DataWriterQos* object where the qos is returned.

Returns RETCODE_OK

Topic ***get_topic**() const
Retrieves the topic for this *DataWriter*.

Returns Pointer to the associated *Topic*

const *DataWriterListener* ***get_listener**() const
Retrieves the listener for this *DataWriter*.

Returns Pointer to the *DataWriterListener*

ReturnCode_t **set_listener**(*DataWriterListener* *listener)
Modifies the *DataWriterListener*, sets the mask to *StatusMask::all()*

Parameters **listener** – new value for the *DataWriterListener*

Returns RETCODE_OK

ReturnCode_t **set_listener**(*DataWriterListener* *listener, const *StatusMask* &mask)
Modifies the *DataWriterListener*.

Parameters

- **listener** – new value for the *DataWriterListener*
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all).

Returns RETCODE_OK

ReturnCode_t **dispose**(void *data, const InstanceHandle_t &handle)

This operation requests the middleware to delete the data (the actual deletion is postponed until there is no more use for that data in the whole system). In general, applications are made aware of the deletion by means of operations on the *DataReader* objects that already knew that instance. This operation does not modify the value of the instance. The instance parameter is passed just for the purposes of identifying the instance. When this operation is used, the Service will automatically supply the value of the source_timestamp that is made available to *DataReader* objects by means of the source_timestamp attribute inside the *SampleInfo*. The constraints on the values of the handle parameter and the corresponding error behavior are the same specified for the unregister_instance operation.

Parameters

- **data** – **[in]** Sample used to deduce instance's key in case of `handle` parameter is HANDLE_NIL.
- **handle** – **[in]** InstanceHandle of the data

Returns RETCODE_PRECONDITION_NOT_MET if the handle introduced does not match with the one associated to the data, RETCODE_OK if the data is correctly sent and RETCODE_ERROR otherwise.

ReturnCode_t **dispose_w_timestamp**(void *instance, const InstanceHandle_t &handle, const fastrtps::*Time_t* &timestamp)

This operation performs the same functions as *dispose* except that the application provides the value for the *source_timestamp* that is made available to *DataReader* objects by means of the *source_timestamp* attribute inside the *SampleInfo*.

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the *dispose* operation.

This operation may return RETCODE_PRECONDITION_NOT_MET and RETCODE_BAD_PARAMETER under the same circumstances described for the *dispose* operation.

This operation may return RETCODE_TIMEOUT and RETCODE_OUT_OF_RESOURCES under the same circumstances described for the *write* operation.

Parameters

- **instance** – Sample used to deduce instance's key in case of `handle` parameter is HANDLE_NIL.
- **handle** – Instance's key to be disposed.
- **timestamp** – Time_t used to set the source_timestamp.

Returns RTPS_DllAPI

ReturnCode_t **get_liveliness_lost_status**(*LivelinessLostStatus* &status)

Returns the liveliness lost status.

Parameters **status** – Liveliness lost status struct

Returns RETCODE_OK

const *Publisher* ***get_publisher**() const

Getter for the *Publisher* that creates this *DataWriter*.

Returns Pointer to the *Publisher*

ReturnCode_t **assert_liveliness**()
This operation manually asserts the liveliness of the *DataWriter*. This is used in combination with the *LivelinessQosPolicy* to indicate to the Service that the entity remains active. This operation need only be used if the LIVELINESS setting is either MANUAL_BY_PARTICIPANT or MANUAL_BY_TOPIC. Otherwise, it has no effect.

Note: Writing data via the write operation on a *DataWriter* asserts liveliness on the *DataWriter* itself and its *DomainParticipant*. Consequently the use of assert_liveliness is only needed if the application is not writing data regularly.

Returns RETCODE_OK if asserted, RETCODE_ERROR otherwise

ReturnCode_t **get_matched_subscription_data**(builtin::SubscriptionBuiltinTopicData &subscription_data, const InstanceHandle_t &subscription_handle) const
Retrieves in a subscription associated with the *DataWriter*.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- **subscription_data** – **[out]** subscription data struct
- **subscription_handle** – InstanceHandle_t of the subscription

Returns RETCODE_OK

ReturnCode_t **get_matched_subscriptions**(std::vector<InstanceHandle_t> &subscription_handles) const
Fills the given vector with the InstanceHandle_t of matched DataReaders.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters **subscription_handles** – **[out]** Vector where the InstanceHandle_t are returned

Returns RETCODE_OK

ReturnCode_t **clear_history**(size_t *removed)
Clears the *DataWriter* history.

Parameters **removed** – size_t pointer to return the size of the data removed

Returns RETCODE_OK if the samples are removed and RETCODE_ERROR otherwise

ReturnCode_t **loan_sample**(void *&sample, *LoanInitializationKind* initialization = *LoanInitializationKind*::*NO_LOAN_INITIALIZATION*)
Get a pointer to the internal pool where the user could directly write.

This method can only be used on a *DataWriter* for a plain data type. It will provide the user with a pointer to an internal buffer where the data type can be prepared for sending.

When using NO_LOAN_INITIALIZATION on the initialization parameter, which is the default, no assumptions should be made on the contents where the pointer points to, as it may be an old pointer being reused. See *LoanInitializationKind* for more details.

Once the sample has been prepared, it can then be published by calling *write*. After a successful call to *write*, the middleware takes ownership of the loaned pointer again, and the user should not access that memory again.

If, for whatever reason, the sample is not published, the loan can be returned by calling *discard_loan*.

Parameters

- **sample** – **[out]** Pointer to the sample on the internal pool.
- **initialization** – **[in]** How to initialize the loaned sample.

Returns ReturnCode_t::RETCODE_ILLEGAL_OPERATION when the data type does not support loans.

Returns ReturnCode_t::RETCODE_NOT_ENABLED if the writer has not been enabled.

Returns ReturnCode_t::RETCODE_OUT_OF_RESOURCES if the pool has been exhausted.

Returns ReturnCode_t::RETCODE_OK if a pointer to a sample is successfully obtained.

ReturnCode_t **discard_loan**(void *&sample)
Discards a loaned sample pointer.

See the description on *loan_sample* for how and when to call this method.

Parameters **sample** – **[inout]** Pointer to the previously loaned sample.

Returns ReturnCode_t::RETCODE_ILLEGAL_OPERATION when the data type does not support loans.

Returns ReturnCode_t::RETCODE_NOT_ENABLED if the writer has not been enabled.

Returns ReturnCode_t::RETCODE_BAD_PARAMETER if the pointer does not correspond to a loaned sample.

Returns ReturnCode_t::RETCODE_OK if the loan is successfully discarded.

ReturnCode_t **get_sending_locators**(rtps::*LocatorList* &locators) const
Get the list of locators from which this *DataWriter* may send data.

Parameters **locators** – **[out]** LocatorList where the list of locators will be stored.

Returns NOT_ENABLED if the reader has not been enabled.

Returns OK if a list of locators is returned.

ReturnCode_t **wait_for_acknowledgments**(void *instance, const InstanceHandle_t &handle, const fastrtps::*Duration_t* &max_wait)
Block the current thread until the writer has received the acknowledgment corresponding to the given instance. Operations performed on the same instance while the current thread is waiting will not be taken into consideration, i.e. this method may return `RETCODE_OK` with those operations unacknowledged.

Parameters

- **instance** – Sample used to deduce instance's key in case of `handle` parameter is HANDLE_NIL.
- **handle** – Instance handle of the data.
- **max_wait** – Maximum blocking time for this operation.

Returns RETCODE_NOT_ENABLED if the writer has not been enabled.

Returns RETCODE_BAD_PARAMETER if `instance` is not a valid pointer.

Returns RETCODE_PRECONDITION_NOT_MET if the topic does not have a key, the key is unknown to the writer, or the key is not consistent with `handle`.

Returns RETCODE_OK if the *DataWriter* received the acknowledgments before the time expired.

Returns RETCODE_TIMEOUT otherwise.

DataWriterListener

class **DataWriterListener**

Class *DataWriterListener*, allows the end user to implement callbacks triggered by certain events.

Subclassed by *eprosima::fastdds::dds::PublisherListener*

Public Functions

inline **DataWriterListener**()

Constructor.

inline virtual **~DataWriterListener**()

Destructor.

inline virtual void **on_publication_matched**(*DataWriter* *writer, const *PublicationMatchedStatus* &info)

This method is called when the *DataWriter* is matched (or unmatched) against an endpoint.

Parameters

- **writer** – Pointer to the associated *DataWriter*
- **info** – Information regarding the matched *DataReader*

inline virtual void **on_offered_deadline_missed**(*DataWriter* *writer, const *OfferedDeadlineMissedStatus* &status)

A method called when a deadline is missed

Parameters

- **writer** – Pointer to the associated *DataWriter*
- **status** – The deadline missed status

inline virtual void **on_offered_incompatible_qos**(*DataWriter* *writer, const *OfferedIncompatibleQosStatus* &status)

A method called when an incompatible QoS is offered

Parameters

- **writer** – Pointer to the associated *DataWriter*
- **status** – The deadline missed status

inline virtual void **on_liveliness_lost**(*DataWriter* *writer, const *LivelinessLostStatus* &status)

Method called when the liveliness of a *DataWriter* is lost.

Parameters

- **writer** – Pointer to the associated *DataWriter*
- **status** – The liveliness lost status

inline virtual void **on_unacknowledged_sample_removed**(*DataWriter* *writer, const InstanceHandle_t &instance)

Method called when a sample has been removed unacknowledged.

Parameters

- **writer** – Pointer to the associated *DataWriter*
- **instance** – Handle to the instance the sample was removed from

DataWriterQos

class **DataWriterQos**

Class *DataWriterQos*, containing all the possible Qos that can be set for a determined *DataWriter*. Although these values can be and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Subclassed by *eprosima::fastdds::statistics::dds::DataWriterQos*

Public Functions

DataWriterQos()

Constructor.

~DataWriterQos() = default

Destructor.

inline *DurabilityQosPolicy* &**durability**()

Getter for *DurabilityQosPolicy*

Returns *DurabilityQosPolicy* reference

inline const *DurabilityQosPolicy* &**durability**() const

Getter for *DurabilityQosPolicy*

Returns *DurabilityQosPolicy* reference

inline void **durability**(const *DurabilityQosPolicy* &durability)

Setter for *DurabilityQosPolicy*

Parameters **durability** – new value for the *DurabilityQosPolicy*

inline *DurabilityServiceQosPolicy* &**durability_service**()

Getter for *DurabilityServiceQosPolicy*

Returns *DurabilityServiceQosPolicy* reference

inline const *DurabilityServiceQosPolicy* &**durability_service**() const

Getter for *DurabilityServiceQosPolicy*

Returns *DurabilityServiceQosPolicy* reference

inline void **durability_service**(const *DurabilityServiceQosPolicy* &durability_service)

Setter for *DurabilityServiceQosPolicy*

Parameters **durability_service** – new value for the *DurabilityServiceQosPolicy*

inline *DeadlineQosPolicy* &**deadline**()

Getter for *DeadlineQosPolicy*

Returns *DeadlineQosPolicy* reference

inline const *DeadlineQosPolicy* &**deadline**() const
Getter for *DeadlineQosPolicy*

Returns *DeadlineQosPolicy* reference

inline void **deadline**(const *DeadlineQosPolicy* &deadline)
Setter for *DeadlineQosPolicy*

Parameters **deadline** – new value for the *DeadlineQosPolicy*

inline *LatencyBudgetQosPolicy* &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Returns *LatencyBudgetQosPolicy* reference

inline const *LatencyBudgetQosPolicy* &**latency_budget**() const
Getter for *LatencyBudgetQosPolicy*

Returns *LatencyBudgetQosPolicy* reference

inline void **latency_budget**(const *LatencyBudgetQosPolicy* &latency_budget)
Setter for *LatencyBudgetQosPolicy*

Parameters **latency_budget** – new value for the *LatencyBudgetQosPolicy*

inline *LivelinessQosPolicy* &**liveliness**()
Getter for *LivelinessQosPolicy*

Returns *LivelinessQosPolicy* reference

inline const *LivelinessQosPolicy* &**liveliness**() const
Getter for *LivelinessQosPolicy*

Returns *LivelinessQosPolicy* reference

inline void **liveliness**(const *LivelinessQosPolicy* &liveliness)
Setter for *LivelinessQosPolicy*

Parameters **liveliness** – new value for the *LivelinessQosPolicy*

inline *ReliabilityQosPolicy* &**reliability**()
Getter for *ReliabilityQosPolicy*

Returns *ReliabilityQosPolicy* reference

inline const *ReliabilityQosPolicy* &**reliability**() const
Getter for *ReliabilityQosPolicy*

Returns *ReliabilityQosPolicy* reference

inline void **reliability**(const *ReliabilityQosPolicy* &reliability)
Setter for *ReliabilityQosPolicy*

Parameters **reliability** – new value for the *ReliabilityQosPolicy*

inline *DestinationOrderQosPolicy* &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Returns *DestinationOrderQosPolicy* reference

inline const *DestinationOrderQosPolicy* &**destination_order**() const
Getter for *DestinationOrderQosPolicy*

Returns *DestinationOrderQosPolicy* reference

inline void **destination_order**(const *DestinationOrderQosPolicy* &destination_order)
Setter for *DestinationOrderQosPolicy*

Parameters **destination_order** – new value for the *DestinationOrderQosPolicy*

inline *HistoryQosPolicy* &**history**()
Getter for *HistoryQosPolicy*

Returns *HistoryQosPolicy* reference

inline const *HistoryQosPolicy* &**history**() const
Getter for *HistoryQosPolicy*

Returns *HistoryQosPolicy* reference

inline void **history**(const *HistoryQosPolicy* &history)
Setter for *HistoryQosPolicy*

Parameters **history** – new value for the *HistoryQosPolicy*

inline *ResourceLimitsQosPolicy* &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Returns *ResourceLimitsQosPolicy* reference

inline const *ResourceLimitsQosPolicy* &**resource_limits**() const
Getter for *ResourceLimitsQosPolicy*

Returns *ResourceLimitsQosPolicy* reference

inline void **resource_limits**(const *ResourceLimitsQosPolicy* &resource_limits)
Setter for *ResourceLimitsQosPolicy*

Parameters **resource_limits** – new value for the *ResourceLimitsQosPolicy*

inline *TransportPriorityQosPolicy* &**transport_priority**()
Getter for *TransportPriorityQosPolicy*

Returns *TransportPriorityQosPolicy* reference

inline const *TransportPriorityQosPolicy* &**transport_priority**() const
Getter for *TransportPriorityQosPolicy*

Returns *TransportPriorityQosPolicy* reference

inline void **transport_priority**(const *TransportPriorityQosPolicy* &transport_priority)
Setter for *TransportPriorityQosPolicy*

Parameters **transport_priority** – new value for the *TransportPriorityQosPolicy*

inline *LifespanQosPolicy* &**lifespan**()
Getter for *LifespanQosPolicy*

Returns *LifespanQosPolicy* reference

inline const *LifespanQosPolicy* &**lifespan**() const
Getter for *LifespanQosPolicy*

Returns *LifespanQosPolicy* reference

inline void **lifespan**(const *LifespanQosPolicy* &lifespan)
Setter for *LifespanQosPolicy*

Parameters **lifespan** – new value for the *LifespanQosPolicy*

inline *UserDataQosPolicy* &**user_data**()
Getter for *UserDataQosPolicy*

Returns *UserDataQosPolicy* reference

inline const *UserDataQosPolicy* &**user_data**() const
Getter for *UserDataQosPolicy*

Returns *UserDataQosPolicy* reference

inline void **user_data**(const *UserDataQosPolicy* &user_data)
Setter for *UserDataQosPolicy*

Parameters **user_data** – new value for the *UserDataQosPolicy*

inline *OwnershipQosPolicy* &**ownership**()
Getter for *OwnershipQosPolicy*

Returns *OwnershipQosPolicy* reference

inline const *OwnershipQosPolicy* &**ownership**() const
Getter for *OwnershipQosPolicy*

Returns *OwnershipQosPolicy* reference

inline void **ownership**(const *OwnershipQosPolicy* &ownership)
Setter for *OwnershipQosPolicy*

Parameters **ownership** – new value for the *OwnershipQosPolicy*

inline *OwnershipStrengthQosPolicy* &**ownership_strength**()
Getter for *OwnershipStrengthQosPolicy*

Returns *OwnershipStrengthQosPolicy* reference

inline const *OwnershipStrengthQosPolicy* &**ownership_strength**() const
Getter for *OwnershipStrengthQosPolicy*

Returns *OwnershipStrengthQosPolicy* reference

inline void **ownership_strength**(const *OwnershipStrengthQosPolicy* &ownership_strength)
Setter for *OwnershipStrengthQosPolicy*

Parameters **ownership_strength** – new value for the *OwnershipStrengthQosPolicy*

inline *WriterDataLifecycleQosPolicy* &**writer_data_lifecycle**()
Getter for *WriterDataLifecycleQosPolicy*

Returns *WriterDataLifecycleQosPolicy* reference

inline const *WriterDataLifecycleQosPolicy* &**writer_data_lifecycle**() const
Getter for *WriterDataLifecycleQosPolicy*

Returns *WriterDataLifecycleQosPolicy* reference

inline void **writer_data_lifecycle**(const *WriterDataLifecycleQosPolicy* &writer_data_lifecycle)
Setter for *WriterDataLifecycleQosPolicy*

Parameters **writer_data_lifecycle** – new value for the *WriterDataLifecycleQosPolicy*

inline *PublishModeQosPolicy* &**publish_mode**()
Getter for *PublishModeQosPolicy*

Returns *PublishModeQosPolicy* reference

inline const *PublishModeQosPolicy* &**publish_mode**() const
Getter for *PublishModeQosPolicy*

Returns *PublishModeQosPolicy* reference

inline void **publish_mode**(const *PublishModeQosPolicy* &publish_mode)
Setter for *PublishModeQosPolicy*

Parameters **publish_mode** – new value for the *PublishModeQosPolicy*

inline *DataRepresentationQosPolicy* &**representation**()
Getter for *DataRepresentationQosPolicy*

Returns *DataRepresentationQosPolicy* reference

inline const *DataRepresentationQosPolicy* &**representation**() const
Getter for *DataRepresentationQosPolicy*

Returns *DataRepresentationQosPolicy* reference

inline void **representation**(const *DataRepresentationQosPolicy* &representation)
Setter for *DataRepresentationQosPolicy*

Parameters **representation** – new value for the *DataRepresentationQosPolicy*

inline *PropertyPolicyQos* &**properties**()
Getter for PropertyPolicyQos

Returns PropertyPolicyQos reference

inline const *PropertyPolicyQos* &**properties**() const
Getter for PropertyPolicyQos

Returns PropertyPolicyQos reference

inline void **properties**(const *PropertyPolicyQos* &properties)
Setter for PropertyPolicyQos

Parameters **properties** – new value for the PropertyPolicyQos

inline *RTPSReliableWriterQos* &**reliable_writer_qos**()
Getter for *RTPSReliableWriterQos*

Returns *RTPSReliableWriterQos* reference

inline const *RTPSReliableWriterQos* &**reliable_writer_qos**() const
Getter for *RTPSReliableWriterQos*

Returns *RTPSReliableWriterQos* reference

inline void **reliable_writer_qos**(const *RTPSReliableWriterQos* &reliable_writer_qos)
Setter for *RTPSReliableWriterQos*

Parameters **reliable_writer_qos** – new value for the *RTPSReliableWriterQos*

inline *RTPSEndpointQos* &**endpoint**()
Getter for *RTPSEndpointQos*

Returns *RTPSEndpointQos* reference

inline const *RTPSEndpointQos* &**endpoint**() const
Getter for *RTPSEndpointQos*

Returns *RTPSEndpointQos* reference

inline void **endpoint**(const *RTPSEndpointQos* &endpoint)
Setter for *RTPSEndpointQos*

Parameters **endpoint** – new value for the *RTPSEndpointQos*

inline *WriterResourceLimitsQos* &**writer_resource_limits**()
Getter for *WriterResourceLimitsQos*

Returns *WriterResourceLimitsQos* reference

inline const *WriterResourceLimitsQos* &**writer_resource_limits**() const
Getter for *WriterResourceLimitsQos*

Returns *WriterResourceLimitsQos* reference

inline void **writer_resource_limits**(const *WriterResourceLimitsQos* &writer_resource_limits)
Setter for *WriterResourceLimitsQos*

Parameters **writer_resource_limits** – new value for the *WriterResourceLimitsQos*

inline fastrtps::rtps::ThroughputControllerDescriptor &**throughput_controller**()
Getter for ThroughputControllerDescriptor

Deprecated:
Use flow_controllers() on DomainParticipantQoS

Returns ThroughputControllerDescriptor reference

inline const fastrtps::rtps::ThroughputControllerDescriptor &**throughput_controller**() const
Getter for ThroughputControllerDescriptor

Deprecated:
Use flow_controllers() on DomainParticipantQoS

Returns ThroughputControllerDescriptor reference

inline void **throughput_controller**(const fastrtps::rtps::ThroughputControllerDescriptor &throughput_controller)
Setter for ThroughputControllerDescriptor

Deprecated:
Use flow_controllers() on DomainParticipantQoS

Parameters **throughput_controller** – new value for the ThroughputControllerDescriptor

inline *DataSharingQosPolicy* &**data_sharing**()
Getter for *DataSharingQosPolicy*

Returns *DataSharingQosPolicy* reference

inline const *DataSharingQosPolicy* &**data_sharing**() const
Getter for *DataSharingQosPolicy*

Returns *DataSharingQosPolicy* reference

inline void **data_sharing**(const *DataSharingQosPolicy* &data_sharing)
Setter for *DataSharingQosPolicy*

Parameters **data_sharing** – new value for the *DataSharingQosPolicy*

const *DataWriterQos* eprosima::fastdds::dds::**DATAWRITER_QOS_DEFAULT**

Publisher

class **Publisher** : public eprosima::fastdds::dds::*DomainEntity*
Class *Publisher*, used to send data to associated subscribers.

Public Functions

virtual **~Publisher**()
Destructor.

virtual ReturnCode_t **enable**() override
This operation enables the *Publisher*.

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the participant creating this *Publisher* is not enabled.

const *PublisherQos* &**get_qos**() const
Allows accessing the *Publisher* Qos.

Returns *PublisherQos* reference

ReturnCode_t **get_qos**(*PublisherQos* &qos) const
Retrieves the *Publisher* Qos.

Returns RETCODE_OK

ReturnCode_t **set_qos**(const *PublisherQos* &qos)
Allows modifying the *Publisher* Qos. The given Qos must be supported by the *PublisherQos*.

Parameters **qos** – *PublisherQos* to be set

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *PublisherListener* ***get_listener**() const
Retrieves the attached *PublisherListener*.

Returns *PublisherListener* pointer

ReturnCode_t **set_listener**(*PublisherListener* *listener)
Modifies the *PublisherListener*, sets the mask to *StatusMask::all()*

Parameters **listener** – new value for the *PublisherListener*

Returns RETCODE_OK

ReturnCode_t **set_listener**(*PublisherListener* *listener, const *StatusMask* &mask)
Modifies the *PublisherListener*.

Parameters

- **listener** – new value for the *PublisherListener*
- **mask** – *StatusMask* that holds statuses the listener responds to

Returns RETCODE_OK

DataWriter ***create_datawriter**(*Topic* *topic, const *DataWriterQos* &qos, *DataWriterListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*(), std::shared_ptr<fastrtps::rtps::*IPayloadPool*> payload_pool = nullptr)

This operation creates a *DataWriter*. The returned *DataWriter* will be attached and belongs to the *Publisher*.

Parameters

- **topic** – *Topic* the *DataWriter* will be listening
- **qos** – QoS of the *DataWriter*.
- **listener** – Pointer to the listener (default: nullptr).
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all).
- **payload_pool** – IPayloadPool shared pointer that defines writer payload (default: nullptr).

Returns Pointer to the created *DataWriter*. nullptr if failed.

DataWriter ***create_datawriter_with_profile**(*Topic* *topic, const std::string &profile_name, *DataWriterListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*(), std::shared_ptr<fastrtps::rtps::*IPayloadPool*> payload_pool = nullptr)

This operation creates a *DataWriter*. The returned *DataWriter* will be attached and belongs to the *Publisher*.

Parameters

- **topic** – *Topic* the *DataWriter* will be listening
- **profile_name** – *DataWriter* profile name.
- **listener** – Pointer to the listener (default: nullptr).
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all).
- **payload_pool** – IPayloadPool shared pointer that defines writer payload (default: nullptr).

Returns Pointer to the created *DataWriter*. nullptr if failed.

ReturnCode_t **delete_datawriter**(const *DataWriter* *writer)

This operation deletes a *DataWriter* that belongs to the *Publisher*.

The delete_datawriter operation must be called on the same *Publisher* object used to create the *DataWriter*. If delete_datawriter is called on a different *Publisher*, the operation will have no effect and it will return false.

The deletion of the *DataWriter* will automatically unregister all instances. Depending on the settings of the WRITER_DATA_LIFECYCLE *QosPolicy*, the deletion of the *DataWriter* may also dispose all instances.

Parameters **writer** – *DataWriter* to delete

Returns RETCODE_PRECONDITION_NOT_MET if it does not belong to this *Publisher*, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

DataWriter ***lookup_datawriter**(const std::string &topic_name) const

This operation retrieves a previously created *DataWriter* belonging to the *Publisher* that is attached to a *Topic* with a matching topic_name. If no such *DataWriter* exists, the operation will return nullptr.

If multiple *DataWriter* attached to the *Publisher* satisfy this condition, then the operation will return one of them. It is not specified which one.

Parameters **topic_name** – Name of the *Topic*

Returns Pointer to a previously created *DataWriter* associated to a *Topic* with the requested topic_name

ReturnCode_t **suspend_publications**()
Indicates to FastDDS that the contained DataWriters are about to be modified.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK if successful, an error code otherwise

ReturnCode_t **resume_publications**()
Indicates to FastDDS that the modifications to the DataWriters are complete.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK if successful, an error code otherwise

ReturnCode_t **begin_coherent_changes**()
Signals the beginning of a set of coherent cache changes using the Datawriters attached to the publisher.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK if successful, an error code otherwise

ReturnCode_t **end_coherent_changes**()
Signals the end of a set of coherent cache changes.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK if successful, an error code otherwise

ReturnCode_t **wait_for_acknowledgments**(const fastrtps::*Duration_t* &max_wait)
This operation blocks the calling thread until either all data written by the reliable *DataWriter* entities is acknowledged by all matched reliable *DataReader* entities, or else the duration specified by the max_wait parameter elapses, whichever happens first. A return value of true indicates that all the samples written have been acknowledged by all reliable matched data readers; a return value of false indicates that max_wait elapsed before all the data was acknowledged.

Parameters **max_wait** – Maximum blocking time for this operation

Returns RETCODE_TIMEOUT if the function takes more than the maximum blocking time established, RETCODE_OK if the *Publisher* receives the acknowledgments and RETCODE_ERROR otherwise.

const *DomainParticipant* ***get_participant**() const
This operation returns the *DomainParticipant* to which the *Publisher* belongs.

Returns Pointer to the *DomainParticipant*

ReturnCode_t **delete_contained_entities**()
Deletes all contained DataWriters.

Returns RETCODE_OK if successful, an error code otherwise

ReturnCode_t **set_default_datawriter_qos**(const *DataWriterQos* &qos)

This operation sets a default value of the *DataWriter* QoS policies which will be used for newly created *DataWriter* entities in the case where the QoS policies are defaulted in the create_datawriter operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value DATAWRITER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_datawriter_qos operation had never been called.

Parameters **qos** – *DataWriterQos* to be set

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *DataWriterQos* &**get_default_datawriter_qos**() const

This operation returns the default value of the *DataWriter* QoS, that is, the QoS policies which will be used for newly created *DataWriter* entities in the case where the QoS policies are defaulted in the create_datawriter operation.

The values retrieved by get_default_datawriter_qos will match the set of values specified on the last successful call to set_default_datawriter_qos, or else, if the call was never made, the default values.

Returns Current default WriterQos

ReturnCode_t **get_default_datawriter_qos**(*DataWriterQos* &qos) const

This operation retrieves the default value of the *DataWriter* QoS, that is, the QoS policies which will be used for newly created *DataWriter* entities in the case where the QoS policies are defaulted in the create_datawriter operation.

The values retrieved by get_default_datawriter_qos will match the set of values specified on the last successful call to set_default_datawriter_qos, or else, if the call was never made, the default values.

Parameters **qos** – Reference to the current default WriterQos.

Returns RETCODE_OK

ReturnCode_t **copy_from_topic_qos**(fastdds::dds::*DataWriterQos* &writer_qos, const fastdds::dds::*TopicQos* &topic_qos) const

Copies *TopicQos* into the corresponding *DataWriterQos*.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- **writer_qos** – **[out]**
- **topic_qos** – **[in]**

Returns RETCODE_OK if successful, an error code otherwise

ReturnCode_t **get_datawriter_qos_from_profile**(const std::string &profile_name, *DataWriterQos* &qos) const

Fills the *DataWriterQos* with the values of the XML profile.

Parameters

- **profile_name** – *DataWriter* profile name.

- **qos** – *DataWriterQos* object where the qos is returned.

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

const InstanceHandle_t &**get_instance_handle**() const
Returns the *Publisher*'s handle.

Returns InstanceHandle of this *Publisher*.

bool **get_datawriters**(std::vector<*DataWriter**> &writers) const
Fills the given vector with all the datawriters of this publisher.

Parameters **writers** – Vector where the DataWriters are returned

Returns true

bool **has_datawriters**() const
This operation checks if the publisher has DataWriters

Returns true if the publisher has one or several DataWriters, false otherwise

PublisherListener

class **PublisherListener** : public eprosima::fastdds::dds::*DataWriterListener*
Class *PublisherListener*, allows the end user to implement callbacks triggered by certain events. It inherits all the *DataWriterListener* callbacks.

Subclassed by *eprosima::fastdds::dds::DomainParticipantListener*

Public Functions

inline **PublisherListener**()
Constructor.

inline virtual **~PublisherListener**()
Destructor.

PublisherQos

class **PublisherQos**
Class *PublisherQos*, containing all the possible Qos that can be set for a determined *Publisher*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

inline **PublisherQos**()
Constructor.

virtual ~**PublisherQos**() = default
Destructor.

inline const *PresentationQosPolicy* &**presentation**() const
Getter for *PresentationQosPolicy*

Returns *PresentationQosPolicy* reference

inline *PresentationQosPolicy* &**presentation**()
Getter for *PresentationQosPolicy*

Returns *PresentationQosPolicy* reference

inline void **presentation**(const *PresentationQosPolicy* &presentation)
Setter for *PresentationQosPolicy*

Parameters **presentation** – *PresentationQosPolicy*

inline const *PartitionQosPolicy* &**partition**() const
Getter for *PartitionQosPolicy*

Returns *PartitionQosPolicy* reference

inline *PartitionQosPolicy* &**partition**()
Getter for *PartitionQosPolicy*

Returns *PartitionQosPolicy* reference

inline void **partition**(const *PartitionQosPolicy* &partition)
Setter for *PartitionQosPolicy*

Parameters **partition** – *PartitionQosPolicy*

inline const *GroupDataQosPolicy* &**group_data**() const
Getter for *GroupDataQosPolicy*

Returns *GroupDataQosPolicy* reference

inline *GroupDataQosPolicy* &**group_data**()
Getter for *GroupDataQosPolicy*

Returns *GroupDataQosPolicy* reference

inline void **group_data**(const *GroupDataQosPolicy* &group_data)
Setter for *GroupDataQosPolicy*

Parameters **group_data** – *GroupDataQosPolicy*

inline const *EntityFactoryQosPolicy* &**entity_factory**() const
Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline *EntityFactoryQosPolicy* &**entity_factory**()
Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline void **entity_factory**(const *EntityFactoryQosPolicy* &entity_factory)
Setter for *EntityFactoryQosPolicy*

Parameters **entity_factory** – *EntityFactoryQosPolicy*

const *PublisherQos* eprosima::fastdds::dds::**PUBLISHER_QOS_DEFAULT**

RTPSReliableWriterQos

class **RTPSReliableWriterQos**

Qos Policy to configure the DisablePositiveACKsQos and the writer timing attributes.

Public Functions

inline **RTPSReliableWriterQos**()

Constructor.

virtual **~RTPSReliableWriterQos**() = default

Destructor.

Public Members

fastrtps::rtps::*WriterTimes* **times**

Writer Timing Attributes.

DisablePositiveACKsQosPolicy **disable_positive_acks**

Disable positive acks QoS, implemented in the library.

bool **disable_heartbeat_piggyback** = false

Disable heartbeat piggyback mechanism.

Subscriber

DataReader

class **DataReader** : public eprosima::fastdds::dds::*DomainEntity*

Class *DataReader*, contains the actual implementation of the behaviour of the *Subscriber*.

Read or take data methods.

Methods to read or take data from the History.

ReturnCode_t **read**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples = LENGTH_UNLIMITED, SampleStateMask sample_states = ANY_SAMPLE_STATE, ViewStateMask view_states = ANY_VIEW_STATE, InstanceStateMask instance_states = ANY_INSTANCE_STATE)

Access a collection of data samples from the *DataReader*.

This operation accesses a collection of Data values from the *DataReader*. The caller can limit the size of the returned collection with the `max_samples` parameter.

The properties of the `data_values` collection and the setting of the *PresentationQosPolicy* may impose further limits on the size of the returned 'list.'

i. If *PresentationQosPolicy::access_scope* is *INSTANCE_PRESENTATION_QOS*, then the returned collection is a 'list' where samples belonging to the same data-instance are consecutive.

ii. If *PresentationQosPolicy::access_scope* is *TOPIC_PRESENTATION_QOS* and *PresentationQosPolicy::ordered_access* is set to `false`, then the returned collection is a 'list' where samples belonging to the same data-instance are consecutive.

iii. If *PresentationQosPolicy::access_scope* is *TOPIC_PRESENTATION_QOS* and *PresentationQosPolicy::ordered_access* is set to `true`, then the returned collection is a 'list' where samples belonging to the same instance may or may not be consecutive. This is because to preserve order it may be necessary to mix samples from different instances.

iv. If *PresentationQosPolicy::access_scope* is *GROUP_PRESENTATION_QOS* and *PresentationQosPolicy::ordered_access* is set to `false`, then the returned collection is a 'list' where samples belonging to the same data instance are consecutive.

v. If *PresentationQosPolicy::access_scope* is *GROUP_PRESENTATION_QOS* and *PresentationQosPolicy::ordered_access* is set to `true`, then the returned collection contains at most one sample. The difference in this case is due to the fact that it is required that the application is able to read samples belonging to different *DataReader* objects in a specific order.

In any case, the relative order between the samples of one instance is consistent with the *DestinationOrderQosPolicy*:

- If *DestinationOrderQosPolicy::kind* is *BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS*, samples belonging to the same instances will appear in the relative order in which there were received (FIFO, earlier samples ahead of the later samples).
- If *DestinationOrderQosPolicy::kind* is *BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS*, samples belonging to the same instances will appear in the relative order implied by the source_timestamp (FIFO, smaller values of source_timestamp ahead of the larger values).

The actual number of samples returned depends on the information that has been received by the middleware as well as the *HistoryQosPolicy*, *ResourceLimitsQosPolicy*, and *ReaderResourceLimitsQos*:

- In the case where the *HistoryQosPolicy::kind* is KEEP_LAST_HISTORY_QOS, the call will return at most *HistoryQosPolicy::depth* samples per instance.
- The maximum number of samples returned is limited by *ResourceLimitsQosPolicy::max_samples*, and by *ReaderResourceLimitsQos::max_samples_per_read*.
- For multiple instances, the number of samples returned is additionally limited by the product (*ResourceLimitsQosPolicy::max_samples_per_instance* * *ResourceLimitsQosPolicy::max_instances*).
- If *ReaderResourceLimitsQos::sample_infos_allocation* has a maximum limit, the number of samples returned may also be limited if insufficient *SampleInfo* resources are available.

If the operation succeeds and the number of samples returned has been limited (by means of a maximum limit, as listed above, or insufficient *SampleInfo* resources), the call will complete successfully and provide those samples the reader is able to return. The user may need to make additional calls, or return outstanding loaned buffers in the case of insufficient resources, in order to access remaining samples.

In addition to the collection of samples, the read operation also uses a collection of *SampleInfo* structures (`sample_infos`).

The initial (input) properties of the `data_values` and `sample_infos` collections will determine the precise behavior of this operation. For the purposes of this description the collections are modeled as having three properties:

- the current length (`len`, see *LoanableCollection::length()*)
- the maximum length (`max_len`, see *LoanableCollection::maximum()*)
- whether the collection container owns the memory of the elements within (`owns`, see *LoanableCollection::has_ownership()*)

The initial (input) values of the `len`, `max_len`, and `owns` properties for the `data_values` and `sample_infos` collections govern the behavior of the read operation as specified by the following rules:

i. The values of `len`, `max_len`, and `owns` for the two collections must be identical. Otherwise read will fail with RETCODE_PRECONDITION_NOT_MET.

ii. On successful output, the values of `len`, `max_len`, and `owns` will be the same for both collections.

iii. If the input `max_len == 0`, then the `data_values` and `sample_infos` collections will be filled with elements that are 'loaned' by the *DataReader*. On output, `owns` will be `false`, `len` will be set to the number of values returned, and `max_len` will be set to a value verifying `max_len >= len`. The use of this variant allows for zero-copy access to the data and the application will need to return the loan to the *DataReader* using the *return_loan* operation.

iv. If the input `max_len > 0` and the input `owns == false`, then the read operation will fail with RETCODE_PRECONDITION_NOT_MET. This avoids the potential hard-to-detect memory leaks caused by an application forgetting to return the loan.

v. If input `max_len > 0` and the input `owns == true`, then the read operation will copy the Data values and *SampleInfo* values into the elements already inside the collections. On output, `owns` will be `true`, `len` will be set to the number of values copied, and `max_len` will remain unchanged. The use of this variant forces a copy but the application can control where the copy is placed and the application will not need to return the loan. The number of samples copied depends on the values of `max_len` and `max_samples`:

   - If `max_samples == LENGTH_UNLIMITED`, then at most `max_len` values will be copied. The use of this variant lets the application limit the number of samples returned to what the sequence can accommodate.
   - If `max_samples <= max_len`, then at most `max_samples` values will be copied. The use of this variant lets the application limit the number of samples returned to fewer that what the sequence can accommodate.
   - If `max_samples > max_len`, then the read operation will fail with RETCODE_PRECONDITION_NOT_MET. This avoids the potential confusion where the application expects to be able to access up to `max_samples`, but that number can never be returned, even if they are available in the *DataReader*, because the output sequence cannot accommodate them.

As described above, upon return the `data_values` and `sample_infos` collections may contain elements 'loaned' from the *DataReader*. If this is the case, the application will need to use the *return_loan* operation to return the loan once it is no longer using the Data in the collection. Upon return from *return_loan*, the collection will have `max_len == 0` and `owns == false`.

The application can determine whether it is necessary to return the loan or not based on the state of the collections when the read operation was called, or by accessing the `owns` property. However, in many cases it may be simpler to always call *return_loan*, as this operation is harmless (i.e., leaves all elements unchanged) if the collection does not have a loan.

On output, the collection of Data values and the collection of *SampleInfo* structures are of the same length and are in a one-to-one correspondence. Each *SampleInfo* provides information, such as the `source_timestamp`, the `sample_state`, `view_state`, and `instance_state`, etc., about the corresponding sample.

Some elements in the returned collection may not have valid data. If the `instance_state` in the *SampleInfo* is *NOT_ALIVE_DISPOSED_INSTANCE_STATE* or *NOT_ALIVE_NO_WRITERS_INSTANCE_STATE*, then the last sample for that instance in the collection, that is, the one whose *SampleInfo* has `sample_rank == 0` does not contain valid data. Samples that contain no data do not count towards the limits imposed by the *ResourceLimitsQosPolicy*.

The act of reading a sample changes its `sample_state` to *READ_SAMPLE_STATE*. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to be *NOT_NEW_VIEW_STATE*. It will not affect the `instance_state` of the instance.

If the *DataReader* has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Important: If the samples "returned" by this method are loaned from the middleware (see *take* for more information on memory loaning), it is important that their contents not be changed. Because the memory in which the data is stored belongs to the middleware, any modifications made to the data will be seen the next time the same samples are read or taken; the samples will no longer reflect the state that was received from the network.

Parameters

- **`data_values`** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **`sample_infos`** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **`max_samples`** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described above.
- **`sample_states`** – **[in]** Only data samples with `sample_state` matching one of these will be returned.
- **`view_states`** – **[in]** Only data samples with `view_state` matching one of these will be returned.
- **`instance_states`** – **[in]** Only data samples with `instance_state` matching one of these will be returned.

Returns Any of the standard return codes.

ReturnCode_t **`read_w_condition`**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples, *ReadCondition* *a_condition)

This operation accesses via 'read' the samples that match the criteria specified in the *ReadCondition*. This operation is especially useful in combination with QueryCondition to filter data samples based on the content.

The specified *ReadCondition* must be attached to the *DataReader*; otherwise the operation will fail and return RETCODE_PRECONDITION_NOT_MET.

In case the *ReadCondition* is a 'plain' *ReadCondition* and not the specialized QueryCondition, the operation is equivalent to calling read and passing as `sample_states`, `view_states` and `instance_states` the value of the corresponding attributes in `a_condition`. Using this operation the application can avoid repeating the same parameters specified when creating the *ReadCondition*.

The samples are accessed with the same semantics as the read operation. If the *DataReader* has no samples that meet the constraints, the return value will be RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned.
- **a_condition** – **[in]** A *ReadCondition* that returned `data_values` must pass

Returns Any of the standard return codes.

ReturnCode_t **read_instance**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples = LENGTH_UNLIMITED, const InstanceHandle_t &a_handle = HANDLE_NIL, SampleStateMask sample_states = ANY_SAMPLE_STATE, ViewStateMask view_states = ANY_VIEW_STATE, InstanceStateMask instance_states = ANY_INSTANCE_STATE)

Access a collection of data samples from the *DataReader*.

This operation accesses a collection of data values from the *DataReader*. The behavior is identical to *read*, except that all samples returned belong to the single specified instance whose handle is `a_handle`.

Upon successful completion, the data collection will contain samples all belonging to the same instance. The corresponding *SampleInfo* verifies *SampleInfo::instance_handle* == `a_handle`.

This operation is semantically equivalent to the *read* operation, except in building the collection. The *DataReader* will check that the sample belongs to the specified instance and otherwise it will not place the sample in the returned collection.

The behavior of this operation follows the same rules as the *read* operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos`. Similar to *read*, this operation may 'loan' elements to the output collections, which must then be returned by means of *return_loan*.

If the *DataReader* has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described in the documentation for *read()*.
- **a_handle** – **[in]** The specified instance to return samples for. The method will fail with RETCODE_BAD_PARAMETER if the handle does not correspond to an existing data-object known to the *DataReader*.
- **sample_states** – **[in]** Only data samples with `sample_state` matching one of these will be returned.
- **view_states** – **[in]** Only data samples with `view_state` matching one of these will be returned.

- **instance_states** – **[in]** Only data samples with `instance_state` matching one of these will be returned.

Returns Any of the standard return codes.

ReturnCode_t **read_next_instance**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples = LENGTH_UNLIMITED, const InstanceHandle_t &previous_handle = HANDLE_NIL, SampleStateMask sample_states = ANY_SAMPLE_STATE, ViewStateMask view_states = ANY_VIEW_STATE, InstanceStateMask instance_states = ANY_INSTANCE_STATE)

Access a collection of data samples from the *DataReader*.

This operation accesses a collection of data values from the *DataReader* where all the samples belong to a single instance. The behavior is similar to *read_instance*, except that the actual instance is not directly specified. Rather, the samples will all belong to the 'next' instance with `instance_handle` 'greater' than the specified 'previous_handle' that has available samples.

This operation implies the existence of a total order 'greater-than' relationship between the instance handles. The specifics of this relationship are not all important and are implementation specific. The important thing is that, according to the middleware, all instances are ordered relative to each other. This ordering is between the instance handles, and should not depend on the state of the instance (e.g. whether it has data or not) and must be defined even for instance handles that do not correspond to instances currently managed by the *DataReader*. For the purposes of the ordering, it should be 'as if' each instance handle was represented as an integer.

The behavior of this operation is 'as if' the *DataReader* invoked *read_instance*, passing the smallest `instance_handle` among all the ones that: (a) are greater than `previous_handle`, and (b) have available samples (i.e. samples that meet the constraints imposed by the specified states).

The special value HANDLE_NIL is guaranteed to be 'less than' any valid `instance_handle`. So the use of the parameter value `previous_handle` == HANDLE_NIL will return the samples for the instance which has the smallest `instance_handle` among all the instances that contain available samples.

This operation is intended to be used in an application-driven iteration, where the application starts by passing `previous_handle` == HANDLE_NIL, examines the samples returned, and then uses the `instance_handle` returned in the *SampleInfo* as the value of the `previous_handle` argument to the next call to *read_next_instance*. The iteration continues until *read_next_instance* fails with RETCODE_NO_DATA.

Note that it is possible to call the *read_next_instance* operation with a `previous_handle` that does not correspond to an instance currently managed by the *DataReader*. This is because as stated earlier the 'greater-than' relationship is defined even for handles not managed by the *DataReader*. One practical situation where this may occur is when an application is iterating through all the instances, takes all the samples of a *NOT_ALIVE_NO_WRITERS_INSTANCE_STATE* instance, returns the loan (at which point the instance information may be removed, and thus the handle becomes invalid), and tries to read the next instance.

The behavior of this operation follows the same rules as the *read* operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos`. Similar to *read*, this operation may 'loan' elements to the output collections, which must then be returned by means of *return_loan*.

If the *DataReader* has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.

- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described in the documentation for *read()*.
- **previous_handle** – **[in]** The 'next smallest' instance with a value greater than this value that has available samples will be returned.
- **sample_states** – **[in]** Only data samples with `sample_state` matching one of these will be returned.
- **view_states** – **[in]** Only data samples with `view_state` matching one of these will be returned.
- **instance_states** – **[in]** Only data samples with `instance_state` matching one of these will be returned.

Returns Any of the standard return codes.

ReturnCode_t **read_next_instance_w_condition**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples, const InstanceHandle_t &previous_handle, *ReadCondition* *a_condition)

This operation accesses a collection of Data values from the *DataReader*. The behavior is identical to *read_next_instance* except that all samples returned satisfy the specified condition. In other words, on success all returned samples belong to the same instance, and the instance is the instance with 'smallest' `instance_handle` among the ones that verify (a) `instance_handle >= previous_handle` and (b) have samples for which the specified *ReadCondition* evaluates to TRUE.

Similar to the operation *read_next_instance* it is possible to call *read_next_instance_w_condition* with a `previous_handle` that does not correspond to an instance currently managed by the *DataReader*.

The behavior of the *read_next_instance_w_condition* operation follows the same rules than the read operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos` collections. Similar to read, the *read_next_instance_w_condition* operation may 'loan' elements to the output collections which must then be returned by means of *return_loan*.

If the *DataReader* has no samples that meet the constraints, the return value will be RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described in the documentation for *read()*.
- **previous_handle** – **[in]** The 'next smallest' instance with a value greater than this value that has available samples will be returned.
- **a_condition** – **[in]** A *ReadCondition* that returned `data_values` must pass

Returns Any of the standard return codes.

ReturnCode_t **read_next_sample**(void *data, *SampleInfo* *info)

This operation copies the next, non-previously accessed Data value from the *DataReader*; the operation also copies the corresponding *SampleInfo*. The implied order among the samples stored in the *DataReader*

is the same as for the read operation.

The read_next_sample operation is semantically equivalent to the read operation where the input Data sequence has `max_length = 1`, the `sample_states = NOT_READ_SAMPLE_STATE`, the `view_states = ANY_VIEW_STATE`, and the `instance_states = ANY_INSTANCE_STATE`.

The read_next_sample operation provides a simplified API to 'read' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the *DataReader*, the operation will return RETCODE_NO_DATA and nothing is copied

Parameters

- **data** – **[out]** Data pointer to store the sample
- **info** – **[out]** *SampleInfo* pointer to store the sample information

Returns Any of the standard return codes.

ReturnCode_t **take**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples = LENGTH_UNLIMITED, SampleStateMask sample_states = ANY_SAMPLE_STATE, ViewStateMask view_states = ANY_VIEW_STATE, InstanceStateMask instance_states = ANY_INSTANCE_STATE)

Access a collection of data samples from the *DataReader*.

This operation accesses a collection of data-samples from the *DataReader* and a corresponding collection of *SampleInfo* structures, and 'removes' them from the *DataReader*. The operation will return either a 'list' of samples or else a single sample. This is controlled by the *PresentationQosPolicy* using the same logic as for the *read* operation.

The act of taking a sample removes it from the *DataReader* so it cannot be 'read' or 'taken' again. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to NOT_NEW. It will not affect the `instance_state` of the instance.

The behavior of the take operation follows the same rules than the *read* operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos` collections. Similar to *read*, the take operation may 'loan' elements to the output collections which must then be returned by means of *return_loan*. The only difference with *read* is that, as stated, the samples returned by take will no longer be accessible to successive calls to read or take.

If the *DataReader* has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described in the documentation for *read()*.
- **sample_states** – **[in]** Only data samples with `sample_state` matching one of these will be returned.
- **view_states** – **[in]** Only data samples with `view_state` matching one of these will be returned.

- **instance_states** – **[in]** Only data samples with `instance_state` matching one of these will be returned.

Returns Any of the standard return codes.

ReturnCode_t **take_w_condition**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples, *ReadCondition* *a_condition)

This operation is analogous to *read_w_condition* except it accesses samples via the 'take' operation.

The specified *ReadCondition* must be attached to the *DataReader*; otherwise the operation will fail and return RETCODE_PRECONDITION_NOT_MET.

The samples are accessed with the same semantics as the *take* operation.

This operation is especially useful in combination with QueryCondition to filter data samples based on the content.

If the *DataReader* has no samples that meet the constraints, the return value will be RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are.
- **a_condition** – **[in]** A *ReadCondition* that returned `data_values` must pass

Returns Any of the standard return codes.

ReturnCode_t **take_instance**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples = LENGTH_UNLIMITED, const InstanceHandle_t &a_handle = HANDLE_NIL, SampleStateMask sample_states = ANY_SAMPLE_STATE, ViewStateMask view_states = ANY_VIEW_STATE, InstanceStateMask instance_states = ANY_INSTANCE_STATE)

Access a collection of data samples from the *DataReader*.

This operation accesses a collection of data values from the *DataReader* and 'removes' them from the *DataReader*.

This operation has the same behavior as *read_instance*, except that the samples are 'taken' from the *DataReader* such that they are no longer accessible via subsequent 'read' or 'take' operations.

The behavior of this operation follows the same rules as the *read* operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos`. Similar to *read*, this operation may 'loan' elements to the output collections, which must then be returned by means of *return_loan*.

If the *DataReader* has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.

- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described in the documentation for *read()*.
- **a_handle** – **[in]** The specified instance to return samples for. The method will fail with RETCODE_BAD_PARAMETER if the handle does not correspond to an existing data-object known to the *DataReader*.
- **sample_states** – **[in]** Only data samples with `sample_state` matching one of these will be returned.
- **view_states** – **[in]** Only data samples with `view_state` matching one of these will be returned.
- **instance_states** – **[in]** Only data samples with `instance_state` matching one of these will be returned.

Returns Any of the standard return codes.

ReturnCode_t **take_next_instance**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples = LENGTH_UNLIMITED, const InstanceHandle_t &previous_handle = HANDLE_NIL, SampleStateMask sample_states = ANY_SAMPLE_STATE, ViewStateMask view_states = ANY_VIEW_STATE, InstanceStateMask instance_states = ANY_INSTANCE_STATE)

Access a collection of data samples from the *DataReader*.

This operation accesses a collection of data values from the *DataReader* and 'removes' them from the *DataReader*.

This operation has the same behavior as *read_next_instance*, except that the samples are 'taken' from the *DataReader* such that they are no longer accessible via subsequent 'read' or 'take' operations.

Similar to the operation *read_next_instance*, it is possible to call this operation with a `previous_handle` that does not correspond to an instance currently managed by the *DataReader*.

The behavior of this operation follows the same rules as the *read* operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos`. Similar to *read*, this operation may 'loan' elements to the output collections, which must then be returned by means of *return_loan*.

If the *DataReader* has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described in the documentation for *read()*.
- **previous_handle** – **[in]** The 'next smallest' instance with a value greater than this value that has available samples will be returned.
- **sample_states** – **[in]** Only data samples with `sample_state` matching one of these will be returned.
- **view_states** – **[in]** Only data samples with `view_state` matching one of these will be returned.

- **instance_states** – **[in]** Only data samples with `instance_state` matching one of these will be returned.

Returns Any of the standard return codes.

ReturnCode_t **take_next_instance_w_condition**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos, int32_t max_samples, const InstanceHandle_t &previous_handle, *ReadCondition* *a_condition)

This operation accesses a collection of Data values from the *DataReader*. The behavior is identical to *read_next_instance* except that all samples returned satisfy the specified condition. In other words, on success all returned samples belong to the same instance, and the instance is the instance with 'smallest' `instance_handle` among the ones that verify (a) `instance_handle >= previous_handle` and (b) have samples for which the specified *ReadCondition* evaluates to TRUE.

Similar to the operation *read_next_instance* it is possible to call *read_next_instance_w_condition* with a `previous_handle` that does not correspond to an instance currently managed by the *DataReader*.

The behavior of the *read_next_instance_w_condition* operation follows the same rules than the read operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos` collections. Similar to read, the *read_next_instance_w_condition* operation may 'loan' elements to the output collections which must then be returned by means of *return_loan*.

If the *DataReader* has no samples that meet the constraints, the return value will be RETCODE_NO_DATA

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples will be returned.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample info will be returned.
- **max_samples** – **[in]** The maximum number of samples to be returned. If the special value LENGTH_UNLIMITED is provided, as many samples will be returned as are available, up to the limits described in the documentation for *read()*.
- **previous_handle** – **[in]** The 'next smallest' instance with a value greater than this value that has available samples will be returned.
- **a_condition** – **[in]** A *ReadCondition* that returned `data_values` must pass

Returns Any of the standard return codes.

ReturnCode_t **take_next_sample**(void *data, *SampleInfo* *info)

This operation copies the next, non-previously accessed Data value from the *DataReader* and 'removes' it from the *DataReader* so it is no longer accessible. The operation also copies the corresponding *SampleInfo*.

This operation is analogous to *read_next_sample* except for the fact that the sample is 'removed' from the *DataReader*.

This operation is semantically equivalent to the *take* operation where the input sequence has `max_length = 1`, the `sample_states = NOT_READ_SAMPLE_STATE`, the `view_states = ANY_VIEW_STATE`, and the `instance_states = ANY_INSTANCE_STATE`.

This operation provides a simplified API to 'take' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the *DataReader*, the operation will return RETCODE_NO_DATA and nothing is copied.

Parameters

- **data** – **[out]** Data pointer to store the sample
- **info** – **[out]** *SampleInfo* pointer to store the sample information

Returns Any of the standard return codes.

Public Functions

virtual **~DataReader**()
Destructor.

virtual ReturnCode_t **enable**() override
This operation enables the *DataReader*.

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the *Subscriber* creating this *DataReader* is not enabled.

bool **wait_for_unread_message**(const fastrtps::*Duration_t* &timeout)
Method to block the current thread until an unread message is available.

Parameters **timeout** – **[in]** Max blocking time for this operation.

Returns true if there is new unread message, false if timeout

ReturnCode_t **wait_for_historical_data**(const fastrtps::*Duration_t* &max_wait) const
Method to block the current thread until an unread message is available.

NOT YET IMPLEMENTED

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters **max_wait** – **[in]** Max blocking time for this operation.

Returns RETCODE_OK if there is new unread message, ReturnCode_t::RETCODE_TIMEOUT if timeout

ReturnCode_t **return_loan**(*LoanableCollection* &data_values, SampleInfoSeq &sample_infos)
This operation indicates to the *DataReader* that the application is done accessing the collection of `data_values` and `sample_infos` obtained by some earlier invocation of *read* or *take* on the *DataReader*.

The `data_values` and `sample_infos` must belong to a single related 'pair'; that is, they should correspond to a pair returned from a single call to read or take. The `data_values` and `sample_infos` must also have been obtained from the same *DataReader* to which they are returned. If either of these conditions is not met, the operation will fail and return RETCODE_PRECONDITION_NOT_MET.

This operation allows implementations of the *read* and *take* operations to "loan" buffers from the *DataReader* to the application and in this manner provide "zero-copy" access to the data. During the loan, the *DataReader* will guarantee that the data and sample-information are not modified.

It is not necessary for an application to return the loans immediately after the read or take calls. However, as these buffers correspond to internal resources inside the *DataReader*, the application should not retain them indefinitely.

The use of the *return_loan* operation is only necessary if the read or take calls "loaned" buffers to the application. This only occurs if the `data_values` and `sample_infos` collections had `max_len == 0` at the time read or take was called. The application may also examine the `owns` property of the collection to determine if there is an outstanding loan. However, calling *return_loan* on a collection that does not have a loan is safe and has no side effects.

If the collections had a loan, upon return from return_loan the collections will have `max_len == 0` .

Parameters

- **data_values** – **[inout]** A *LoanableCollection* object where the received data samples were obtained from an earlier invocation of read or take on this *DataReader*.
- **sample_infos** – **[inout]** A SampleInfoSeq object where the received sample infos were obtained from an earlier invocation of read or take on this *DataReader*.

Returns Any of the standard return codes.

ReturnCode_t **get_key_value**(void *key_holder, const InstanceHandle_t &handle)
NOT YET IMPLEMENTED

This operation can be used to retrieve the instance key that corresponds to an `instance_handle`. The operation will only fill the fields that form the key inside the key_holder instance.

This operation may return BAD_PARAMETER if the InstanceHandle_t a_handle does not correspond to an existing data-object known to the *DataReader*. If the implementation is not able to check invalid handles then the result in this situation is unspecified.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- **key_holder** – **[inout]**
- **handle** – **[in]**

Returns Any of the standard return codes.

InstanceHandle_t **lookup_instance**(const void *instance) const
Takes as a parameter an instance and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The instance parameter is only used for the purpose of examining the fields that define the key.

Parameters **instance** – **[in]** Data pointer to the sample

Returns handle of the given `instance`.

Returns HANDLE_NIL if `instance` is nullptr.

Returns HANDLE_NIL if there is no instance on the *DataReader*'s history with the same key as `instance`.

ReturnCode_t **get_first_untaken_info**(*SampleInfo* *info)
Returns information about the first untaken sample.

Parameters **info** – **[out]** Pointer to a SampleInfo_t structure to store first untaken sample information.

Returns RETCODE_OK if sample info was returned. RETCODE_NO_DATA if there is no sample to take.

uint64_t **get_unread_count**() const
Get the number of samples pending to be read. The number includes samples that may not yet be available to be read or taken by the user, due to samples being received out of order.

Returns the number of samples on the reader history that have never been read.

uint64_t **get_unread_count**(bool mark_as_read) const
Get the number of samples pending to be read.

Parameters **mark_as_read** – Whether the unread samples should be marked as read or not.

Returns the number of samples on the reader history that have never been read.

const fastrtps::rtps::*GUID_t* &**guid**()
Get associated GUID.

Returns Associated GUID

const fastrtps::rtps::*GUID_t* &**guid**() const
Get associated GUID.

Returns Associated GUID

InstanceHandle_t **get_instance_handle**() const
Getter for the associated InstanceHandle.

Returns Copy of the InstanceHandle

TypeSupport **type**()
Getter for the data type.

Returns *TypeSupport* associated to the *DataReader*.

const *TopicDescription* ***get_topicdescription**() const
Get *TopicDescription*.

Returns *TopicDescription* pointer.

ReturnCode_t **get_requested_deadline_missed_status**(*RequestedDeadlineMissedStatus* &status)
Get the requested deadline missed status.

Returns The deadline missed status.

ReturnCode_t **get_requested_incompatible_qos_status**(*RequestedIncompatibleQosStatus* &status)
Get the requested incompatible qos status.

Parameters **status** – **[out]** Requested incompatible qos status.

Returns RETCODE_OK

ReturnCode_t **set_qos**(const *DataReaderQos* &qos)
Setter for the *DataReaderQos*.

Parameters **qos** – **[in]** new value for the *DataReaderQos*.

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *DataReaderQos* &**get_qos**() const
Getter for the *DataReaderQos*.

Returns Pointer to the *DataReaderQos*.

ReturnCode_t **get_qos**(*DataReaderQos* &qos) const
Getter for the *DataReaderQos*.

Parameters **qos** – **[in]** *DataReaderQos* where the qos is returned.

Returns RETCODE_OK

ReturnCode_t **set_listener**(*DataReaderListener* *listener)
Modifies the *DataReaderListener*, sets the mask to *StatusMask::all()*.

Parameters `listener` – **[in]** new value for the *DataReaderListener*.

Returns RETCODE_OK

ReturnCode_t **set_listener**(*DataReaderListener* *listener, const *StatusMask* &mask)
Modifies the *DataReaderListener*.

Parameters

- `listener` – **[in]** new value for the *DataReaderListener*.
- `mask` – **[in]** *StatusMask* that holds statuses the listener responds to (default: all).

Returns RETCODE_OK

const *DataReaderListener* ***get_listener**() const
Getter for the *DataReaderListener*.

Returns Pointer to the *DataReaderListener*

ReturnCode_t **get_liveliness_changed_status**(*LivelinessChangedStatus* &status) const
Get the liveliness changed status.

Parameters `status` – **[out]** *LivelinessChangedStatus* object where the status is returned.

Returns RETCODE_OK

ReturnCode_t **get_sample_lost_status**(*SampleLostStatus* &status) const
Get the SAMPLE_LOST communication status.

Parameters `status` – **[out]** SampleLostStatus object where the status is returned.

Returns RETCODE_OK

ReturnCode_t **get_sample_rejected_status**(*SampleRejectedStatus* &status) const
Get the SAMPLE_REJECTED communication status.

Parameters `status` – **[out]** *SampleRejectedStatus* object where the status is returned.

Returns RETCODE_OK

ReturnCode_t **get_subscription_matched_status**(*SubscriptionMatchedStatus* &status) const
Returns the subscription matched status.

Parameters `status` – **[out]** subscription matched status struct

Returns RETCODE_OK

ReturnCode_t **get_matched_publication_data**(builtin::PublicationBuiltinTopicData &publication_data,
const fastrtps::rtps::*InstanceHandle_t*
&publication_handle) const
Retrieves in a publication associated with the *DataWriter*.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- `publication_data` – **[out]** publication data struct
- `publication_handle` – InstanceHandle_t of the publication

Returns RETCODE_OK

ReturnCode_t **get_matched_publications**(std::vector<InstanceHandle_t> &publication_handles) const
Fills the given vector with the InstanceHandle_t of matched DataReaders.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters **publication_handles** – **[out]** Vector where the InstanceHandle_t are returned

Returns RETCODE_OK

ReadCondition ***create_readcondition**(SampleStateMask sample_states, ViewStateMask view_states, InstanceStateMask instance_states)
This operation creates a *ReadCondition*. The returned *ReadCondition* will be attached and belong to the *DataReader*.

Parameters

- **sample_states** – **[in]** Only data samples with `sample_state` matching one of these will trigger the created condition.
- **view_states** – **[in]** Only data samples with `view_state` matching one of these will trigger the created condition.
- **instance_states** – **[in]** Only data samples with `instance_state` matching one of these will trigger the created condition.

Returns pointer to the created *ReadCondition*, nullptr in case of error.

QueryCondition ***create_querycondition**(SampleStateMask sample_states, ViewStateMask view_states, InstanceStateMask instance_states, const std::string &query_expression, const std::vector<std::string> &query_parameters)
This operation creates a QueryCondition. The returned QueryCondition will be attached and belong to the *DataReader*.

Parameters

- **sample_states** – **[in]** Only data samples with `sample_state` matching one of these will trigger the created condition.
- **view_states** – **[in]** Only data samples with `view_state` matching one of these will trigger the created condition.
- **instance_states** – **[in]** Only data samples with `instance_state` matching one of these will trigger the created condition.
- **query_expression** – **[in]** Only data samples matching this query will trigger the created condition.
- **query_parameters** – **[in]** Value of the parameters on the query expression.

Returns pointer to the created QueryCondition, nullptr in case of error.

ReturnCode_t **delete_readcondition**(*ReadCondition* *a_condition)
This operation deletes a *ReadCondition* attached to the *DataReader*.

Parameters **a_condition** – pointer to a *ReadCondition* belonging to the *DataReader*

Returns RETCODE_OK

const *Subscriber* ***get_subscriber**() const
Getter for the *Subscriber*.

Returns *Subscriber* pointer

ReturnCode_t **delete_contained_entities**()
This operation deletes all the entities that were created by means of the "create" operations on the *DataReader*. That is, it deletes all contained *ReadCondition* and QueryCondition objects.

The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted.

Returns Any of the standard return codes.

bool **is_sample_valid**(const void *data, const *SampleInfo* *info) const
Checks whether a loaned sample is still valid or is corrupted. Calling this method on a sample which has not been loaned, or one for which the loan has been returned yields undefined behavior.

Parameters

- **data** – Pointer to the sample data to check
- **info** – Pointer to the *SampleInfo* related to `data`

Returns true if the sample is valid

ReturnCode_t **get_listening_locators**(rtps::*LocatorList* &locators) const
Get the list of locators on which this *DataReader* is listening.

Parameters **locators** – **[out]** LocatorList where the list of locators will be stored.

Returns NOT_ENABLED if the reader has not been enabled.

Returns OK if a list of locators is returned.

DataReaderListener

class **DataReaderListener**
Class *DataReaderListener*, it should be used by the end user to implement specific callbacks to certain actions.

Subclassed by *eprosima::fastdds::dds::SubscriberListener*

Public Functions

inline **DataReaderListener**()
Constructor.

inline virtual **~DataReaderListener**()
Destructor.

inline virtual void **on_data_available**(*DataReader* *reader)
Virtual function to be implemented by the user containing the actions to be performed when new Data Messages are received.

Parameters **reader** – *DataReader*

inline virtual void **on_subscription_matched**(*DataReader* *reader, const fastdds::dds::*SubscriptionMatchedStatus* &info)
Virtual method to be called when the subscriber is matched with a new Writer (or unmatched); i.e., when a writer publishing in the same topic is discovered.

Parameters

- **reader** – *DataReader*

- **info** – The subscription matched status

inline virtual void **on_requested_deadline_missed**(*DataReader* *reader, const fastrtps::RequestedDeadlineMissedStatus &status)

Virtual method to be called when a topic misses the deadline period

Parameters

- **reader** – *DataReader*
- **status** – The requested deadline missed status

inline virtual void **on_liveliness_changed**(*DataReader* *reader, const fastrtps::LivelinessChangedStatus &status)

Method called when the liveliness status associated to a subscriber changes.

Parameters

- **reader** – The *DataReader*
- **status** – The liveliness changed status

inline virtual void **on_sample_rejected**(*DataReader* *reader, const fastrtps::SampleRejectedStatus &status)

Method called when a sample was rejected.

Parameters

- **reader** – The *DataReader*
- **status** – The rejected status

inline virtual void **on_requested_incompatible_qos**(*DataReader* *reader, const *RequestedIncompatibleQosStatus* &status)

Method called an incompatible QoS was requested.

Parameters

- **reader** – The *DataReader*
- **status** – The requested incompatible QoS status

inline virtual void **on_sample_lost**(*DataReader* *reader, const *SampleLostStatus* &status)

Method called when a sample was lost.

Parameters

- **reader** – The *DataReader*
- **status** – The sample lost status

DataReaderQos

class **DataReaderQos**

Class *DataReaderQos*, containing all the possible Qos that can be set for a determined *DataReader*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Subclassed by *eprosima::fastdds::statistics::dds::DataReaderQos*, eprosima::fastdds::statistics::dds::MonitorServiceDataReaderQ

Public Functions

inline **DataReaderQos**()
Constructor.

inline *DurabilityQosPolicy* &**durability**()
Getter for *DurabilityQosPolicy*

Returns *DurabilityQosPolicy* reference

inline const *DurabilityQosPolicy* &**durability**() const
Getter for *DurabilityQosPolicy*

Returns *DurabilityQosPolicy* const reference

inline void **durability**(const *DurabilityQosPolicy* &new_value)
Setter for *DurabilityQosPolicy*

Parameters **new_value** – new value for the *DurabilityQosPolicy*

inline *DeadlineQosPolicy* &**deadline**()
Getter for *DeadlineQosPolicy*

Returns *DeadlineQosPolicy* reference

inline const *DeadlineQosPolicy* &**deadline**() const
Getter for *DeadlineQosPolicy*

Returns *DeadlineQosPolicy* const reference

inline void **deadline**(const *DeadlineQosPolicy* &new_value)
Setter for *DeadlineQosPolicy*

Parameters **new_value** – new value for the *DeadlineQosPolicy*

inline *LatencyBudgetQosPolicy* &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Returns *LatencyBudgetQosPolicy* reference

inline const *LatencyBudgetQosPolicy* &**latency_budget**() const
Getter for *LatencyBudgetQosPolicy*

Returns *LatencyBudgetQosPolicy* const reference

inline void **latency_budget**(const *LatencyBudgetQosPolicy* &new_value)
Setter for *LatencyBudgetQosPolicy*

Parameters **new_value** – new value for the *LatencyBudgetQosPolicy*

inline *LivelinessQosPolicy* &**liveliness**()
Getter for *LivelinessQosPolicy*

Returns *LivelinessQosPolicy* reference

inline const *LivelinessQosPolicy* &**liveliness**() const
Getter for *LivelinessQosPolicy*

Returns *LivelinessQosPolicy* const reference

inline void **liveliness**(const *LivelinessQosPolicy* &new_value)
Setter for *LivelinessQosPolicy*

Parameters **new_value** – new value for the *LivelinessQosPolicy*

inline *ReliabilityQosPolicy* &**reliability**()
Getter for *ReliabilityQosPolicy*

Returns *ReliabilityQosPolicy* reference

inline const *ReliabilityQosPolicy* &**reliability**() const
Getter for *ReliabilityQosPolicy*

Returns *ReliabilityQosPolicy* const reference

inline void **reliability**(const *ReliabilityQosPolicy* &new_value)
Setter for *ReliabilityQosPolicy*

Parameters **new_value** – new value for the *ReliabilityQosPolicy*

inline *DestinationOrderQosPolicy* &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Returns *DestinationOrderQosPolicy* reference

inline const *DestinationOrderQosPolicy* &**destination_order**() const
Getter for *DestinationOrderQosPolicy*

Returns *DestinationOrderQosPolicy* const reference

inline void **destination_order**(const *DestinationOrderQosPolicy* &new_value)
Setter for *DestinationOrderQosPolicy*

Parameters **new_value** – new value for the *DestinationOrderQosPolicy*

inline *HistoryQosPolicy* &**history**()
Getter for *HistoryQosPolicy*

Returns *HistoryQosPolicy* reference

inline const *HistoryQosPolicy* &**history**() const
Getter for *HistoryQosPolicy*

Returns *HistoryQosPolicy* const reference

inline void **history**(const *HistoryQosPolicy* &new_value)
Setter for *HistoryQosPolicy*

Parameters **new_value** – new value for the *HistoryQosPolicy*

inline *ResourceLimitsQosPolicy* &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Returns *ResourceLimitsQosPolicy* reference

inline const *ResourceLimitsQosPolicy* &**resource_limits**() const
Getter for *ResourceLimitsQosPolicy*

Returns *ResourceLimitsQosPolicy* const reference

inline void **resource_limits**(const *ResourceLimitsQosPolicy* &new_value)
Setter for *ResourceLimitsQosPolicy*

Parameters **new_value** – new value for the *ResourceLimitsQosPolicy*

inline *UserDataQosPolicy* &**user_data**()
Getter for *UserDataQosPolicy*

Returns *UserDataQosPolicy* reference

inline const *UserDataQosPolicy* &**user_data**() const
Getter for *UserDataQosPolicy*

Returns *UserDataQosPolicy* const reference

inline void **user_data**(const *UserDataQosPolicy* &new_value)
Setter for *UserDataQosPolicy*

Parameters **new_value** – new value for the *UserDataQosPolicy*

inline *OwnershipQosPolicy* &**ownership**()
Getter for *OwnershipQosPolicy*

Returns *OwnershipQosPolicy* reference

inline const *OwnershipQosPolicy* &**ownership**() const
Getter for *OwnershipQosPolicy*

Returns *OwnershipQosPolicy* const reference

inline void **ownership**(const *OwnershipQosPolicy* &new_value)
Setter for *OwnershipQosPolicy*

Parameters **new_value** – new value for the *OwnershipQosPolicy*

inline *TimeBasedFilterQosPolicy* &**time_based_filter**()
Getter for *TimeBasedFilterQosPolicy*

Returns *TimeBasedFilterQosPolicy* reference

inline const *TimeBasedFilterQosPolicy* &**time_based_filter**() const
Getter for *TimeBasedFilterQosPolicy*

Returns *TimeBasedFilterQosPolicy* const reference

inline void **time_based_filter**(const *TimeBasedFilterQosPolicy* &new_value)
Setter for *TimeBasedFilterQosPolicy*

Parameters **new_value** – new value for the *TimeBasedFilterQosPolicy*

inline *ReaderDataLifecycleQosPolicy* &**reader_data_lifecycle**()
Getter for *ReaderDataLifecycleQosPolicy*

Returns *ReaderDataLifecycleQosPolicy* reference

inline const *ReaderDataLifecycleQosPolicy* &**reader_data_lifecycle**() const
Getter for *ReaderDataLifecycleQosPolicy*

Returns *ReaderDataLifecycleQosPolicy* const reference

inline void **reader_data_lifecycle**(const *ReaderDataLifecycleQosPolicy* &new_value)
Setter for *ReaderDataLifecycleQosPolicy*

Parameters **new_value** – new value for the *ReaderDataLifecycleQosPolicy*

inline *LifespanQosPolicy* &**lifespan**()
Getter for *LifespanQosPolicy*

Returns *LifespanQosPolicy* reference

inline const *LifespanQosPolicy* &**lifespan**() const
Getter for *LifespanQosPolicy*

Returns *LifespanQosPolicy* const reference

inline void **lifespan**(const *LifespanQosPolicy* &new_value)
Setter for *LifespanQosPolicy*

Parameters **new_value** – new value for the *LifespanQosPolicy*

inline *DurabilityServiceQosPolicy* &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Returns *DurabilityServiceQosPolicy* reference

inline const *DurabilityServiceQosPolicy* &**durability_service**() const
Getter for *DurabilityServiceQosPolicy*

Returns *DurabilityServiceQosPolicy* const reference

inline void **durability_service**(const *DurabilityServiceQosPolicy* &new_value)
Setter for *DurabilityServiceQosPolicy*

Parameters **new_value** – new value for the *DurabilityServiceQosPolicy*

inline *RTPSReliableReaderQos* &**reliable_reader_qos**()
Getter for *RTPSReliableReaderQos*

Returns *RTPSReliableReaderQos* reference

inline const *RTPSReliableReaderQos* &**reliable_reader_qos**() const
Getter for *RTPSReliableReaderQos*

Returns *RTPSReliableReaderQos* const reference

inline void **reliable_reader_qos**(const *RTPSReliableReaderQos* &new_value)
Setter for *RTPSReliableReaderQos*

Parameters **new_value** – new value for the *RTPSReliableReaderQos*

inline *TypeConsistencyQos* &**type_consistency**()
Getter for *TypeConsistencyQos*

Returns *TypeConsistencyQos* reference

inline const *TypeConsistencyQos* &**type_consistency**() const
Getter for *TypeConsistencyQos*

Returns *TypeConsistencyQos* const reference

inline void **type_consistency**(const *TypeConsistencyQos* &new_value)
Setter for *TypeConsistencyQos*

Parameters **new_value** – new value for the *TypeConsistencyQos*

inline bool **expects_inline_qos**() const
Getter for expectsInlineQos

Returns expectsInlineQos

inline void **expects_inline_qos**(bool new_value)
Setter for expectsInlineQos

Parameters **new_value** – new value for the expectsInlineQos

inline *PropertyPolicyQos* &**properties**()
Getter for PropertyPolicyQos

Returns PropertyPolicyQos reference

inline const *PropertyPolicyQos* &**properties**() const
Getter for PropertyPolicyQos

Returns PropertyPolicyQos const reference

inline void **properties**(const *PropertyPolicyQos* &new_value)
Setter for PropertyPolicyQos

Parameters **new_value** – new value for the PropertyPolicyQos

inline *RTPSEndpointQos* &**endpoint**()
Getter for *RTPSEndpointQos*

Returns *RTPSEndpointQos* reference

inline const *RTPSEndpointQos* &**endpoint**() const
Getter for *RTPSEndpointQos*

Returns *RTPSEndpointQos* const reference

inline void **endpoint**(const *RTPSEndpointQos* &new_value)
Setter for *RTPSEndpointQos*

Parameters **new_value** – new value for the *RTPSEndpointQos*

inline *ReaderResourceLimitsQos* &**reader_resource_limits**()
Getter for *ReaderResourceLimitsQos*

Returns *ReaderResourceLimitsQos* reference

inline const *ReaderResourceLimitsQos* &**reader_resource_limits**() const
Getter for *ReaderResourceLimitsQos*

Returns *ReaderResourceLimitsQos* const reference

inline void **reader_resource_limits**(const *ReaderResourceLimitsQos* &new_value)
Setter for *ReaderResourceLimitsQos*

Parameters **new_value** – new value for the *ReaderResourceLimitsQos*

inline *DataSharingQosPolicy* &**data_sharing**()
Getter for *DataSharingQosPolicy*

Returns *DataSharingQosPolicy* reference

inline const *DataSharingQosPolicy* &**data_sharing**() const
Getter for *DataSharingQosPolicy*

Returns *DataSharingQosPolicy* reference

inline void **data_sharing**(const *DataSharingQosPolicy* &data_sharing)
Setter for *DataSharingQosPolicy*

Parameters **data_sharing** – new value for the *DataSharingQosPolicy*

const *DataReaderQos* eprosima::fastdds::dds::**DATAREADER_QOS_DEFAULT**

InstanceStateKind

enum eprosima::fastdds::dds::**InstanceStateKind**
Indicates if the samples are from an alive *DataWriter* or not.

For each instance, the middleware internally maintains an instance state. The instance state can be:

- *ALIVE_INSTANCE_STATE* indicates that (a) samples have been received for the instance, (b) there are alive *DataWriter* entities writing the instance, and (c) the instance has not been explicitly disposed (or else more samples have been received after it was disposed).

- *NOT_ALIVE_DISPOSED_INSTANCE_STATE* indicates the instance was explicitly disposed by a *DataWriter* by means of the dispose operation.

- *NOT_ALIVE_NO_WRITERS_INSTANCE_STATE* indicates the instance has been declared as not-alive by the *DataReader* because it detected that there are no alive *DataWriter* entities writing that instance.

The precise behavior events that cause the instance state to change depends on the setting of the OWNERSHIP QoS:

- If OWNERSHIP is set to EXCLUSIVE_OWNERSHIP_QOS, then the instance state becomes *NOT_ALIVE_DISPOSED_INSTANCE_STATE* only if the *DataWriter* that "owns" the instance explicitly disposes it. The instance state becomes *ALIVE_INSTANCE_STATE* again only if the *DataWriter* that owns the instance writes it.

- If OWNERSHIP is set to SHARED_OWNERSHIP_QOS, then the instance state becomes *NOT_ALIVE_DISPOSED_INSTANCE_STATE* if any *DataWriter* explicitly disposes the instance. The instance state becomes *ALIVE_INSTANCE_STATE* as soon as any *DataWriter* writes the instance again.

The instance state available in the *SampleInfo* is a snapshot of the instance state of the instance at the time the collection was obtained (i.e. at the time read or take was called). The instance state is therefore the same for all samples in the returned collection that refer to the same instance.

Values:

enumerator **ALIVE_INSTANCE_STATE**
Instance is currently in existence.

enumerator **NOT_ALIVE_DISPOSED_INSTANCE_STATE**
Not alive disposed instance. The instance has been disposed by a *DataWriter*.

enumerator **NOT_ALIVE_NO_WRITERS_INSTANCE_STATE**
Not alive no writers for instance. None of the *DataWriter* objects that are currently alive (according to the LIVELINESS QoS) are writing the instance.

ReadCondition

class **ReadCondition** : public eprosima::fastdds::dds::*Condition*
A *Condition* specifically dedicated to read operations and attached to one *DataReader*.

ReadCondition objects allow an application to specify the data samples it is interested in (by specifying the desired sample_states, view_states, and instance_states). The condition will only be triggered when suitable information is available. They are to be used in conjunction with a *WaitSet* as normal conditions. More than one *ReadCondition* may be attached to the same *DataReader*.

Public Functions

virtual bool **get_trigger_value**() const noexcept override
Retrieves the trigger_value of the *Condition*.

Returns true if trigger_value is set to 'true', 'false' otherwise

DataReader ***get_datareader**() const noexcept
Retrieves the *DataReader* associated with the *ReadCondition*.

Note that there is exactly one *DataReader* associated with each *ReadCondition*.

Returns pointer to the *DataReader* associated with this *ReadCondition*.

SampleStateMask **get_sample_state_mask**() const noexcept
Retrieves the set of sample_states taken into account to determine the trigger_value of this condition.

Returns the sample_states specified when the *ReadCondition* was created.

ViewStateMask **get_view_state_mask**() const noexcept
Retrieves the set of view_states taken into account to determine the trigger_value of this condition.

Returns the view_states specified when the *ReadCondition* was created.

InstanceStateMask **get_instance_state_mask**() const noexcept
Retrieves the set of instance_states taken into account to determine the trigger_value of this condition.

Returns the instance_states specified when the *ReadCondition* was created.

ReaderResourceLimitsQos

class **ReaderResourceLimitsQos**
Qos Policy to configure the limit of the reader resources.

Public Functions

ReaderResourceLimitsQos() = default
Constructor.

virtual **~ReaderResourceLimitsQos**() = default
Destructor.

Public Members

fastrtps::ResourceLimitedContainerConfig **matched_publisher_allocation**
Matched publishers allocation limits.

fastrtps::ResourceLimitedContainerConfig **sample_infos_allocation** = {32u}
SampleInfo allocation limits.

fastrtps::ResourceLimitedContainerConfig **outstanding_reads_allocation** = {2u}
Loaned collections allocation limits.

int32_t **max_samples_per_read** = 32
Maximum number of samples to return on a single call to read / take.

This attribute is a signed integer to be consistent with the `max_samples` argument of *DataReader* methods, but should always have a strict positive value. Bear in mind that a big number here may cause the creation of the *DataReader* to fail due to pre-allocation of internal resources.

Default value: 32.

RTPSReliableReaderQos

class **RTPSReliableReaderQos**
Qos Policy to configure the DisablePositiveACKsQos and the reader attributes.

Public Functions

inline **RTPSReliableReaderQos**()
Constructor.

virtual **~RTPSReliableReaderQos**() = default
Destructor.

Public Members

fastrtps::rtps::*ReaderTimes* **times**
Times associated with the Reliable Readers events.

DisablePositiveACKsQosPolicy **disable_positive_ACKs**
Control the sending of positive ACKs.

SampleInfo

struct **SampleInfo**
SampleInfo is the information that accompanies each sample that is ‘read’ or ‘taken.
,

Public Members

SampleStateKind **sample_state**
indicates whether or not the corresponding data sample has already been read

ViewStateKind **view_state**
indicates whether the *DataReader* has already seen samples for the most-current generation of the related instance.

InstanceStateKind **instance_state**
indicates whether the instance is currently in existence or, if it has been disposed, the reason why it was disposed.

int32_t **disposed_generation_count**
number of times the instance had become alive after it was disposed

int32_t **no_writers_generation_count**
number of times the instance had become alive after it was disposed because no writers

int32_t **sample_rank**
number of samples related to the same instance that follow in the collection

int32_t **generation_rank**
the generation difference between the time the sample was received, and the time the most recent sample in the collection was received.

int32_t **absolute_generation_rank**
the generation difference between the time the sample was received, and the time the most recent sample was received. The most recent sample used for the calculation may or may not be in the returned collection

fastrtps::rtps::*Time_t* **source_timestamp**
time provided by the *DataWriter* when the sample was written

fastrtps::rtps::*Time_t* **reception_timestamp**
time provided by the *DataReader* when the sample was added to its history

InstanceHandle_t **instance_handle**
identifies locally the corresponding instance

InstanceHandle_t **publication_handle**
identifies locally the *DataWriter* that modified the instance

Is the same InstanceHandle_t that is returned by the operation get_matched_publications on the *DataReader*

bool **valid_data**
whether the DataSample contains data or is only used to communicate of a change in the instance

fastrtps::rtps::*SampleIdentity* **sample_identity**
Sample Identity (Extension for RPC)

fastrtps::rtps::*SampleIdentity* **related_sample_identity**
Related Sample Identity (Extension for RPC)

SampleStateKind

enum `eprosima::fastdds::dds::SampleStateKind`
Indicates whether or not a sample has ever been read.

For each sample received, the middleware internally maintains a sample state relative to each *DataReader*. This sample state can have the following values:

- *READ_SAMPLE_STATE* indicates that the *DataReader* has already accessed that sample by means of a read or take operation

- *NOT_READ_SAMPLE_STATE* indicates that the *DataReader* has not accessed that sample before.

The sample state will, in general, be different for each sample in the collection returned by read or take.

Values:

enumerator **READ_SAMPLE_STATE**
Sample has been read.

enumerator **NOT_READ_SAMPLE_STATE**
Sample has not been read.

Subscriber

class **Subscriber** : public eprosima::fastdds::dds::*DomainEntity*
Class *Subscriber*, contains the public API that allows the user to control the reception of messages. This class should not be instantiated directly. DomainRTPSParticipant class should be used to correctly create this element.

Public Functions

inline virtual **~Subscriber**()
Destructor.

virtual ReturnCode_t **enable**() override
This operation enables the *Subscriber*.

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the participant creating this *Subscriber* is not enabled.

const *SubscriberQos* &**get_qos**() const
Allows accessing the *Subscriber* Qos.

Returns *SubscriberQos* reference

ReturnCode_t **get_qos**(*SubscriberQos* &qos) const
Retrieves the *Subscriber* Qos.

Parameters **qos** – *SubscriberQos* where the qos is returned

Returns RETCODE_OK

ReturnCode_t **set_qos**(const *SubscriberQos* &qos)
Allows modifying the *Subscriber* Qos. The given Qos must be supported by the *SubscriberQos*.

Parameters **qos** – new value for *SubscriberQos*

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *SubscriberListener* ***get_listener**() const
Retrieves the attached *SubscriberListener*.

Returns Pointer to the *SubscriberListener*

ReturnCode_t **set_listener**(*SubscriberListener* *listener)
Modifies the *SubscriberListener*, sets the mask to *StatusMask::all()*

Parameters **listener** – new value for *SubscriberListener*

Returns RETCODE_OK

ReturnCode_t **set_listener**(*SubscriberListener* *listener, const *StatusMask* &mask)

Modifies the *SubscriberListener*.

Parameters

- **listener** – new value for the *SubscriberListener*
- **mask** – *StatusMask* that holds statuses the listener responds to.

Returns RETCODE_OK

DataReader ***create_datareader**(*TopicDescription* *topic, const *DataReaderQos* &reader_qos, *DataReaderListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*(), std::shared_ptr<fastrtps::rtps::*IPayloadPool*> payload_pool = nullptr)

This operation creates a *DataReader*. The returned *DataReader* will be attached and belong to the *Subscriber*.

Parameters

- **topic** – *Topic* the *DataReader* will be listening.
- **reader_qos** – QoS of the *DataReader*.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all).
- **payload_pool** – IPayloadPool shared pointer that defines reader payload (default: nullptr).

Returns Pointer to the created *DataReader*. nullptr if failed.

DataReader ***create_datareader_with_profile**(*TopicDescription* *topic, const std::string &profile_name, *DataReaderListener* *listener = nullptr, const *StatusMask* &mask = *StatusMask*::*all*(), std::shared_ptr<fastrtps::rtps::*IPayloadPool*> payload_pool = nullptr)

This operation creates a *DataReader*. The returned *DataReader* will be attached and belongs to the *Subscriber*.

Parameters

- **topic** – *Topic* the *DataReader* will be listening.
- **profile_name** – *DataReader* profile name.
- **listener** – Pointer to the listener (default: nullptr)
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all).
- **payload_pool** – IPayloadPool shared pointer that defines reader payload (default: nullptr).

Returns Pointer to the created *DataReader*. nullptr if failed.

ReturnCode_t **delete_datareader**(const *DataReader* *reader)

This operation deletes a *DataReader* that belongs to the *Subscriber*.

The delete_datareader operation must be called on the same *Subscriber* object used to create the *DataReader*. If delete_datareader is called on a different *Subscriber*, the operation will have no effect and it will return an error.

Parameters **reader** – *DataReader* to delete

Returns RETCODE_PRECONDITION_NOT_MET if the datareader does not belong to this subscriber, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

DataReader ***lookup_datareader**(const std::string &topic_name) const

This operation retrieves a previously-created *DataReader* belonging to the *Subscriber* that is attached to a *Topic* with a matching topic_name. If no such *DataReader* exists, the operation will return nullptr.

If multiple DataReaders attached to the *Subscriber* satisfy this condition, then the operation will return one of them. It is not specified which one.

Parameters **topic_name** – Name of the topic associated to the *DataReader*

Returns Pointer to a previously created *DataReader* created on a *Topic* with that topic_name

ReturnCode_t **get_datareaders**(std::vector<*DataReader**> &readers) const

This operation allows the application to access the *DataReader* objects.

Parameters **readers** – Vector of *DataReader* where the list of existing readers is returned

Returns RETCODE_OK

ReturnCode_t **get_datareaders**(std::vector<*DataReader**> &readers, const std::vector<*SampleStateKind*> &sample_states, const std::vector<*ViewStateKind*> &view_states, const std::vector<*InstanceStateKind*> &instance_states) const

This operation allows the application to access the *DataReader* objects that contain samples with the specified sample_states, view_states, and instance_states.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- **readers** – **[out]** Vector of *DataReader* where the list of existing readers is returned
- **sample_states** – Vector of SampleStateKind
- **view_states** – Vector of ViewStateKind
- **instance_states** – Vector of InstanceStateKind

Returns RETCODE_OK

bool **has_datareaders**() const

This operation checks if the subscriber has DataReaders

Returns true if the subscriber has one or several DataReaders, false in other case

ReturnCode_t **begin_access**()

Indicates that the application is about to access the data samples in any of the *DataReader* objects attached to the *Subscriber*.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK

ReturnCode_t **end_access**()

Indicates that the application has finished accessing the data samples in *DataReader* objects managed by the *Subscriber*.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK

ReturnCode_t **notify_datareaders**() const
This operation invokes the operation on_data_available on the *DataReaderListener* objects attached to contained *DataReader* entities.

This operation is typically invoked from the on_data_on_readers operation in the *SubscriberListener*. That way the *SubscriberListener* can delegate to the *DataReaderListener* objects the handling of the data.

Returns RETCODE_OK

ReturnCode_t **delete_contained_entities**()
Deletes all contained DataReaders. If the DataReaders have any QueryCondition or *ReadCondition*, they are deleted before the *DataReader* itself.

Returns RETCODE_OK if successful, an error code otherwise

ReturnCode_t **set_default_datareader_qos**(const *DataReaderQos* &qos)
This operation sets a default value of the *DataReader* QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value DATAREADER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_datareader_qos operation had never been called.

Parameters **qos** – new value for *DataReaderQos* to set as default

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

const *DataReaderQos* &**get_default_datareader_qos**() const
This operation returns the default value of the *DataReader* QoS, that is, the QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Returns Current default *DataReaderQos*.

DataReaderQos &**get_default_datareader_qos**()
This operation returns the default value of the *DataReader* QoS, that is, the QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Returns Current default *DataReaderQos*.

ReturnCode_t **get_default_datareader_qos**(*DataReaderQos* &qos) const
This operation retrieves the default value of the *DataReader* QoS, that is, the QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Parameters **qos** – *DataReaderQos* where the default_qos is returned

Returns RETCODE_OK

ReturnCode_t **get_datareader_qos_from_profile**(const std::string &profile_name, *DataReaderQos* &qos) const

Fills the *DataReaderQos* with the values of the XML profile.

Parameters

- **profile_name** – *DataReader* profile name.
- **qos** – *DataReaderQos* object where the qos is returned.

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

const *DomainParticipant* ***get_participant**() const

This operation returns the *DomainParticipant* to which the *Subscriber* belongs.

Returns *DomainParticipant* Pointer

const InstanceHandle_t &**get_instance_handle**() const

Returns the *Subscriber*'s handle.

Returns InstanceHandle of this *Subscriber*.

Public Static Functions

static ReturnCode_t **copy_from_topic_qos**(*DataReaderQos* &reader_qos, const *TopicQos* &topic_qos)

Copies *TopicQos* into the corresponding *DataReaderQos*.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters

- **reader_qos** – **[inout]**
- **topic_qos** – **[in]**

Returns RETCODE_OK if successful, an error code otherwise

SubscriberListener

class **SubscriberListener** : public eprosima::fastdds::dds::*DataReaderListener*

Class *SubscriberListener*, it should be used by the end user to implement specific callbacks to certain actions. It also inherits all *DataReaderListener* callbacks.

Subclassed by *eprosima::fastdds::dds::DomainParticipantListener*

Public Functions

inline **SubscriberListener**()
Constructor.

inline virtual **~SubscriberListener**()
Destructor.

inline virtual void **on_data_on_readers**(*Subscriber* *sub)
Virtual function to be implemented by the user containing the actions to be performed when a new Data Message is available on any reader.

Parameters **sub** – *Subscriber*

SubscriberQos

class **SubscriberQos**
Class *SubscriberQos*, contains all the possible Qos that can be set for a determined *Subscriber*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

inline **SubscriberQos**()
Constructor.

inline virtual **~SubscriberQos**()
Destructor.

inline const *PresentationQosPolicy* &**presentation**() const
Getter for *PresentationQosPolicy*

Returns *PresentationQosPolicy* reference

inline *PresentationQosPolicy* &**presentation**()
Getter for *PresentationQosPolicy*

Returns *PresentationQosPolicy* reference

inline void **presentation**(const *PresentationQosPolicy* &presentation)
Setter for *PresentationQosPolicy*

Parameters **presentation** – new value for the *PresentationQosPolicy*

inline const *PartitionQosPolicy* &**partition**() const
Getter for *PartitionQosPolicy*

Returns *PartitionQosPolicy* reference

inline *PartitionQosPolicy* &**partition**()
Getter for *PartitionQosPolicy*

Returns *PartitionQosPolicy* reference

inline void **partition**(const *PartitionQosPolicy* &partition)
Setter for *PartitionQosPolicy*

Parameters **partition** – new value for the *PartitionQosPolicy*

inline const *GroupDataQosPolicy* &**group_data**() const
Getter for *GroupDataQosPolicy*

Returns *GroupDataQosPolicy* reference

inline *GroupDataQosPolicy* &**group_data**()
Getter for *GroupDataQosPolicy*

Returns *GroupDataQosPolicy* reference

inline void **group_data**(const *GroupDataQosPolicy* &group_data)
Setter for *GroupDataQosPolicy*

Parameters **group_data** – new value for the *GroupDataQosPolicy*

inline const *EntityFactoryQosPolicy* &**entity_factory**() const
Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline *EntityFactoryQosPolicy* &**entity_factory**()
Getter for *EntityFactoryQosPolicy*

Returns *EntityFactoryQosPolicy* reference

inline void **entity_factory**(const *EntityFactoryQosPolicy* &entity_factory)
Setter for *EntityFactoryQosPolicy*

Parameters **entity_factory** – new value for the *EntityFactoryQosPolicy*

const *SubscriberQos* eprosima::fastdds::dds::**SUBSCRIBER_QOS_DEFAULT**

TypeConsistencyQos

class **TypeConsistencyQos** : public eprosima::fastdds::dds::*QosPolicy*
Qos Policy to configure the XTypes Qos associated to the *DataReader*.

Public Functions

inline **TypeConsistencyQos**()
Constructor.

virtual **~TypeConsistencyQos**() = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

Public Members

TypeConsistencyEnforcementQosPolicy **type_consistency**
Type consistency enforcement Qos.

DataRepresentationQosPolicy **representation**
Data Representation Qos.

ViewStateKind

enum eprosima::fastdds::dds::**ViewStateKind**
Indicates whether or not an instance is new.

For each instance (identified by the key), the middleware internally maintains a view state relative to each *DataReader*. This view state can have the following values:

- *NEW_VIEW_STATE* indicates that either this is the first time that the *DataReader* has ever accessed samples of that instance, or else that the *DataReader* has accessed previous samples of the instance, but the instance has since been reborn (i.e. become not-alive and then alive again). These two cases are distinguished by examining the *SampleInfo::disposed_generation_count* and the *SampleInfo::no_writers_generation_count*.
- *NOT_NEW_VIEW_STATE* indicates that the *DataReader* has already accessed samples of the same instance and that the instance has not been reborn since.

The view_state available in the *SampleInfo* is a snapshot of the view state of the instance relative to the *DataReader* used to access the samples at the time the collection was obtained (i.e. at the time read or take was called). The view_state is therefore the same for all samples in the returned collection that refer to the same instance.

Once an instance has been detected as not having any "live" writers and all the samples associated with the instance are "taken" from the DDSDataReader, the middleware can reclaim all local resources regarding the instance. Future samples will be treated as "never seen."

Values:

enumerator **NEW_VIEW_STATE**
New instance.This latest generation of the instance has not previously been accessed.

enumerator **NOT_NEW_VIEW_STATE**
Not a new instance. This latest generation of the instance has previously been accessed.

Topic

TopicDataType

class **TopicDataType**
Class *TopicDataType* used to provide the DomainRTPSParticipant with the methods to serialize, deserialize and get the key of a specific data type. The user should created a class that inherits from this one, where Serialize and deserialize methods MUST be implemented. ,

Subclassed by eprosima::fastdds::dds::builtin::TypeLookup_ReplyPubSubType, eprosima::fastdds::dds::builtin::TypeLookup_RequestPubSubType

Public Functions

TopicDataType()
Constructor.

virtual **~TopicDataType**()
Destructor.

virtual bool **serialize**(void *data, fastrtps::rtps::*SerializedPayload_t* *payload) = 0
Serialize method, it should be implemented by the user, since it is abstract. It is VERY IMPORTANT that the user sets the SerializedPayload length correctly.

Parameters

- **data** – **[in]** Pointer to the data
- **payload** – **[out]** Pointer to the payload

Returns True if correct.

virtual bool **serialize**(void *data, fastrtps::rtps::*SerializedPayload_t* *payload, DataRepresentationId_t data_representation)
Serialize method, it should be implemented by the user, since it is abstract. If not implemented, this method will call the serialize method in which the topic data representation is not considered. It is VERY IMPORTANT that the user sets the SerializedPayload length correctly.

Parameters

- **data** – **[in]** Pointer to the data
- **payload** – **[out]** Pointer to the payload
- **data_representation** – **[in]** Representation that should be used to encode the data into the payload.

Returns True if correct.

virtual bool **deserialize**(fastrtps::rtps::*SerializedPayload_t* *payload, void *data) = 0
Deserialize method, it should be implemented by the user, since it is abstract.

Parameters

- **payload** – **[in]** Pointer to the payload
- **data** – **[out]** Pointer to the data

Returns True if correct.

virtual std::function<uint32_t()> **getSerializedSizeProvider**(void *data) = 0
Returns a function which can be used to calculate the serialized size of the provided data.

Parameters **data** – **[in]** Pointer to data.

Returns Functor which calculates the serialized size of the data.

virtual std::function<uint32_t()> **getSerializedSizeProvider**(void *data, DataRepresentationId_t data_representation)
Returns a function which can be used to calculate the serialized size of the provided data.

Parameters

- **data** – **[in]** Pointer to data.

- **data_representation** – **[in]** Representation that should be used for calculating the serialized size.

Returns Functor which calculates the serialized size of the data.

virtual void ***createData**() = 0
Create a Data Type.

Returns Void pointer to the created object.

virtual void **deleteData**(void *data) = 0
Remove a previously created object.

Parameters **data** – Pointer to the created Data.

virtual bool **getKey**(void *data, fastrtps::rtps::*InstanceHandle_t* *ihandle, bool force_md5 = false) = 0
Get the key associated with the data.

Parameters

- **data** – **[in]** Pointer to the data.
- **ihandle** – **[out]** Pointer to the Handle.
- **force_md5** – **[in]** Force MD5 checking.

Returns True if correct.

inline void **setName**(const char *nam)
Set topic data type name

Parameters **nam** – *Topic* data type name

inline const char ***getName**() const
Get topic data type name

Returns *Topic* data type name

inline bool **auto_fill_type_object**() const
Get the type object auto-fill configuration

Returns true if the type object should be auto-filled

inline void **auto_fill_type_object**(bool auto_fill_type_object)
Set the type object auto-fill configuration

Parameters **auto_fill_type_object** – new value to set

inline bool **auto_fill_type_information**() const
Get the type information auto-fill configuration

Returns true if the type information should be auto-filled

inline void **auto_fill_type_information**(bool auto_fill_type_information)
Set type information auto-fill configuration

Parameters **auto_fill_type_information** – new value to set

inline const std::shared_ptr<*TypeIdV1*> **type_identifier**() const
Get the type identifier

Returns *TypeIdV1*

inline void **type_identifier**(const *TypeIdV1* &id)
Set type identifier

Parameters **id** – new value for *TypeIdV1*

inline void **type_identifier**(const std::shared_ptr<*TypeIdV1*> id)
Set type identifier

Parameters **id** – shared pointer to *TypeIdV1*

inline const std::shared_ptr<*TypeObjectV1*> **type_object**() const
Get the type object

Returns *TypeObjectV1*

inline void **type_object**(const *TypeObjectV1* &object)
Set type object

Parameters **object** – new value for *TypeObjectV1*

inline void **type_object**(std::shared_ptr<*TypeObjectV1*> object)
Set type object

Parameters **object** – shared pointer to *TypeObjectV1*

inline const std::shared_ptr<xtypes::*TypeInformation*> **type_information**() const
Get the type information

Returns TypeInformation

inline void **type_information**(const xtypes::*TypeInformation* &info)
Set type information

Parameters **info** – new value for TypeInformation

inline void **type_information**(std::shared_ptr<xtypes::*TypeInformation*> info)
Set type information

Parameters **info** – shared pointer to TypeInformation

inline virtual bool **is_bounded**() const
Checks if the type is bounded.

inline virtual bool **is_plain**() const
Checks if the type is plain when using default encoding.

inline virtual bool **is_plain**(DataRepresentationId_t) const
Checks if the type is plain when using a specific encoding.

inline virtual bool **construct_sample**(void *memory) const
Construct a sample on a memory location.

Parameters **memory** – Pointer to the memory location where the sample should be constructed.

Returns whether this type supports in-place construction or not.

Public Members

uint32_t **m_typeSize**
Maximum serialized size of the type in bytes. If the type has unbounded fields, and therefore cannot have a maximum size, use 0.

bool **m_isGetKeyDefined**
Indicates whether the method to obtain the key has been implemented.

TypeSupport

class **TypeSupport** : public std::shared_ptr<fastdds::dds::*TopicDataType*>
Class *TypeSupport* used to provide the DomainRTPSParticipant with the methods to serialize, deserialize and get the key of a specific data type. The user should created a class that inherits from this one, where Serialize and deserialize methods MUST be implemented.

Note: This class inherits from std::shared_ptr<TopicDataType>.

Subclassed by eprosima::fastdds::dds::builtin::TypeLookup_ReplyTypeSupport, eprosima::fastdds::dds::builtin::TypeLookup_RequestTypeSupport

Public Functions

TypeSupport() noexcept = default
Constructor.

TypeSupport(const *TypeSupport* &type) noexcept = default
Copy Constructor.

Parameters **type** – Another instance of *TypeSupport*

TypeSupport(*TypeSupport* &&type) noexcept = default
Move Constructor.

Parameters **type** – Another instance of *TypeSupport*

TypeSupport &**operator=**(const *TypeSupport* &type) noexcept = default
Copy Assignment.

Parameters **type** – Another instance of *TypeSupport*

TypeSupport &**operator=**(*TypeSupport* &&type) noexcept = default
Move Assignment.

Parameters **type** – Another instance of *TypeSupport*

inline explicit **TypeSupport**(fastdds::dds::*TopicDataType* *ptr)
TypeSupport constructor that receives a *TopicDataType* pointer.

The passed pointer will be managed by the *TypeSupport* object, so creating two *TypeSupport* from the same pointer or deleting the passed pointer will produce a runtime error.

Parameters **ptr** –

inline **TypeSupport**(fastrtps::types::DynamicPubSubType ptr)
TypeSupport constructor that receives a DynamicPubSubType.

It will copy the instance so the user will keep the ownership of his object.

Parameters **ptr** –

virtual ReturnCode_t **register_type**(*DomainParticipant* *participant) const
Registers the type on a participant.

Parameters **participant** – *DomainParticipant* where the type is going to be registered

Returns RETCODE_BAD_PARAMETER if the type name is empty, RETCODE_PRECONDITION_NOT_MET if there is another type with the same name registered on the *DomainParticipant* and RETCODE_OK if it is registered correctly

virtual ReturnCode_t **register_type**(*DomainParticipant* *participant, std::string type_name) const
Registers the type on a participant.

Parameters

- **participant** – *DomainParticipant* where the type is going to be registered
- **type_name** – Name of the type to register

Returns RETCODE_BAD_PARAMETER if the type name is empty, RETCODE_PRECONDITION_NOT_MET if there is another type with the same name registered on the *DomainParticipant* and RETCODE_OK if it is registered correctly

inline virtual const std::string &**get_type_name**() const
Getter for the type name.

Returns name of the data type

inline virtual bool **serialize**(void *data, fastrtps::rtps::*SerializedPayload_t* *payload)
Serializes the data.

Parameters

- **data** – Pointer to data
- **payload** – Pointer to payload

Returns true if it is serialized correctly, false if not

virtual bool **serialize**(void *data, fastrtps::rtps::*SerializedPayload_t* *payload, DataRepresentationId_t data_representation)
Serializes the data.

Parameters

- **data** – Pointer to data
- **payload** – Pointer to payload
- **data_representation** – **[in]** Representation that should be used to encode the data into the payload.

Returns true if it is serialized correctly, false if not

virtual bool **deserialize**(fastrtps::rtps::*SerializedPayload_t* *payload, void *data)
Deserializes the data.

Parameters

- **payload** – Pointer to payload
- **data** – Pointer to data

Returns true if it is deserialized correctly, false if not

inline virtual std::function<uint32_t()> **get_serialized_size_provider**(void *data)
Returns a function which can be used to calculate the serialized size of the provided data.

Parameters **data** – **[in]** Pointer to data.

Returns Functor which calculates the serialized size of the data.

inline virtual std::function<uint32_t()> **get_serialized_size_provider**(void *data, DataRepresentationId_t data_representation)
Returns a function which can be used to calculate the serialized size of the provided data.

Parameters

- **data** – **[in]** Pointer to data.
- **data_representation** – **[in]** Representation that should be used for calculating the serialized size.

Returns Functor which calculates the serialized size of the data.

inline virtual void *create_data()
Creates new data.

Returns Pointer to the data

inline virtual void delete_data(void *data)
Deletes data.

Parameters **data** – Pointer to the data to delete

inline virtual bool get_key(void *data, InstanceHandle_t *i_handle, bool force_md5 = false)
Getter for the data key.

Parameters

- **data** – Pointer to data
- **i_handle** – InstanceHandle pointer to store the key
- **force_md5** – boolean to force md5 (default: false)

Returns true if the key is returned, false if not

inline bool empty() const
Check if the *TypeSupport* is empty.

Returns true if empty, false if not

inline virtual bool is_bounded() const
Checks if the type is bounded.

inline virtual bool is_plain() const
Checks if the type is plain when using default encoding.

inline virtual bool is_plain(DataRepresentationId_t data_representation) const
Checks if the type is plain when using a specific encoding.

TopicDescription

class TopicDescription
Class *TopicDescription*, represents the fact that both publications and subscriptions are tied to a single data-type

Subclassed by *eprosima::fastdds::dds::ContentFilteredTopic*, *eprosima::fastdds::dds::Topic*

Public Functions

virtual *DomainParticipant* ***get_participant**() const = 0
Get the *DomainParticipant* to which the *TopicDescription* belongs.

Returns The *DomainParticipant* to which the *TopicDescription* belongs.

inline const std::string &**get_name**() const
Get the name used to create this *TopicDescription*.

Returns the name used to create this *TopicDescription*.

inline const std::string &**get_type_name**() const
Get the associated type name.

Returns the type name.

Topic

class **Topic** : public eprosima::fastdds::dds::*DomainEntity*, public eprosima::fastdds::dds::*TopicDescription*
Class *Topic*, represents the fact that both publications and subscriptions are tied to a single data-type

Public Functions

virtual *DomainParticipant* ***get_participant**() const override
Getter for the *DomainParticipant*.

Returns *DomainParticipant* pointer

ReturnCode_t **get_inconsistent_topic_status**(*InconsistentTopicStatus* &status)
Allows the application to retrieve the INCONSISTENT_TOPIC_STATUS status of a *Topic*.

> **Warning:** Not supported yet. Currently returns RETCODE_UNSUPPORTED

Parameters **status** – [out] Status to be retrieved.

Returns RETCODE_OK

const *TopicQos* &**get_qos**() const
Allows accessing the *Topic* Qos.

Returns reference to *TopicQos*

ReturnCode_t **get_qos**(*TopicQos* &qos) const
Retrieves the *Topic* Qos.

Parameters **qos** – *TopicQos* where the qos is returned

Returns RETCODE_OK

ReturnCode_t **set_qos**(const *TopicQos* &qos)
Allows modifying the *Topic* Qos. The given Qos must be supported by the *Topic*.

Parameters **qos** – new *TopicQos* value to set for the *Topic*.

Return values

- **RETCODE_IMMUTABLE_POLICY** – if a change was not allowed.

- **RETCODE_INCONSISTENT_POLICY** – if new qos has inconsistent values.
- **RETCODE_OK** – if qos was updated.

const *TopicListener* ***get_listener**() const
Retrieves the attached *TopicListener*.

Returns pointer to *TopicListener*

ReturnCode_t **set_listener**(*TopicListener* *listener, const *StatusMask* &mask = *StatusMask*::*all*())
Modifies the *TopicListener*.

Parameters

- **listener** – new value for the *TopicListener*
- **mask** – *StatusMask* that holds statuses the listener responds to (default: all).

Returns RETCODE_OK

ContentFilteredTopic

class **ContentFilteredTopic** : public eprosima::fastdds::dds::*TopicDescription*
Specialization of *TopicDescription* that allows for content-based subscriptions.

Public Functions

Topic ***get_related_topic**() const
Getter for the related topic.

This operation returns the *Topic* associated with the *ContentFilteredTopic*. That is, the *Topic* specified when the *ContentFilteredTopic* was created.

const std::string &**get_filter_expression**() const
Get the filter expression.

This operation returns filter expression associated with this *ContentFilteredTopic*. It will return the `filter_expression` specified on the last successful call to `set_expression` or, if that method is never called, the expression specified when the *ContentFilteredTopic* was created.

Returns the `filter_expression`.

ReturnCode_t **get_expression_parameters**(std::vector<std::string> &expression_parameters) const
Get the expression parameters.

This operation returns expression parameters associated with this *ContentFilteredTopic*. These will be the `expression_parameters` specified on the last successful call to `set_expression` or `set_expression_parameters`. If those methods have never been called, the expression parameters specified when the *ContentFilteredTopic* was created will be returned.

Parameters **expression_parameters** – **[out]** The expression parameters currently associated with the *ContentFilteredTopic*.

Returns RETCODE_OK

ReturnCode_t **set_expression_parameters**(const std::vector<std::string> &expression_parameters)
Set the expression parameters.

This operation changes expression parameters associated with this *ContentFilteredTopic*.

Parameters **expression_parameters** – **[in]** The expression parameters to set.

Returns RETCODE_OK if the expression parameters where correctly updated.

Returns RETCODE_BAD_PARAMETER if the expression parameters do not match with the current `filter_expression`.

ReturnCode_t **set_filter_expression**(const std::string &filter_expression, const std::vector<std::string> &expression_parameters)

Set the filter expression and the expression parameters.

This operation changes the filter expression and the expression parameters associated with this *ContentFilteredTopic*.

Parameters

- **filter_expression** – **[in]** The filter expression to set.
- **expression_parameters** – **[in]** The expression parameters to set.

Returns RETCODE_OK if the expression and parameters where correctly updated.

Returns RETCODE_BAD_PARAMETER if `filter_expression` is not valid for this *ContentFilteredTopic*.

Returns RETCODE_BAD_PARAMETER if the expression parameters do not match with the `filter_expression`.

virtual *DomainParticipant* ***get_participant**() const override

Getter for the *DomainParticipant*.

Returns *DomainParticipant* pointer

FASTDDS_SQLFILTER_NAME eprosima::fastdds::dds::sqlfilter_name

IContentFilter

struct **IContentFilter**

The interface that content filter objects should implement.

Public Functions

virtual bool **evaluate**(const SerializedPayload &payload, const *FilterSampleInfo* &sample_info, const GUID_t &reader_guid) const = 0

Evaluate if a serialized payload should be accepted by certain reader.

Parameters

- **payload** – **[in]** The serialized payload of the sample being evaluated.
- **sample_info** – **[in]** The accompanying sample information.
- **reader_guid** – **[in]** The GUID of the reader for which the filter is being evaluated.

Returns whether the sample should be accepted for the specified reader.

struct **FilterSampleInfo**

Selected information from the cache change that is passed to the content filter object on payload evaluation.

Public Members

SampleIdentity **sample_identity**
Identity of the sample being filtered.

SampleIdentity **related_sample_identity**
Identity of a sample related to the one being filtered.

IContentFilterFactory

struct **IContentFilterFactory**
The interface that a factory of *IContentFilter* objects should implement.

Public Functions

virtual ReturnCode_t **create_content_filter**(const char *filter_class_name, const char *type_name, const *TopicDataType* *data_type, const char *filter_expression, const ParameterSeq &filter_parameters, *IContentFilter* *&filter_instance) = 0
Create or update an *IContentFilter* instance.

Parameters

- **filter_class_name** – **[in]** Filter class name for which the factory is being called. Allows using the same factory for different filter classes.
- **type_name** – **[in]** Type name of the topic being filtered.
- **data_type** – **[in]** Type support object of the topic being filtered.
- **filter_expression** – **[in]** Content filter expression. May be nullptr when updating the parameters of a filter instance.
- **filter_parameters** – **[in]** Values to set for the filter parameters (n on the filter expression).
- **filter_instance** – **[inout]** When a filter is being created, it will be nullptr on input, and will have the pointer to the created filter instance on output. The caller takes ownership of the filter instance returned. When a filter is being updated, it will have a previously returned pointer on input. The method takes ownership of the filter instance during its execution, and can update the filter instance or even destroy it and create a new one. The caller takes ownership of the filter instance returned. It should always have a valid pointer upon return. The original state of the filter instance should be preserved when an error is returned.

Returns A return code indicating the result of the operation.

virtual ReturnCode_t **delete_content_filter**(const char *filter_class_name, *IContentFilter* *filter_instance) = 0
Delete an *IContentFilter* instance.

Parameters

- **filter_class_name** – **[in]** Filter class name for which the factory is being called. Allows using the same factory for different filter classes.

- **filter_instance** – **[in]** A pointer to a filter instance previously returned by create_content_filter. The factory takes ownership of the filter instance, and can decide to destroy it or keep it for future use. In case of deletion, note this pointer must be downcasted to the derived class.

Returns A return code indicating the result of the operation.

TopicListener

class **TopicListener**

Class *TopicListener*, it should be used by the end user to implement specific callbacks to certain actions.

Subclassed by *eprosima::fastdds::dds::DomainParticipantListener*

Public Functions

inline **TopicListener**()

Constructor.

inline virtual **~TopicListener**()

Destructor.

inline virtual void **on_inconsistent_topic**(*Topic* *topic, *InconsistentTopicStatus* status)

Virtual function to be implemented by the user containing the actions to be performed when another topic exists with the same name but different characteristics.

Parameters

- **topic** – *Topic*
- **status** – The inconsistent topic status

TopicQos

class **TopicQos**

Class *TopicQos*, containing all the possible Qos that can be set for a determined *Topic*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

TopicQos()

Constructor.

inline const *TopicDataQosPolicy* &**topic_data**() const

Getter for *TopicDataQosPolicy*

Returns TopicDataQos reference

inline *TopicDataQosPolicy* &**topic_data**()

Getter for *TopicDataQosPolicy*

Returns TopicDataQos reference

inline void **topic_data**(const *TopicDataQosPolicy* &value)
Setter for *TopicDataQosPolicy*

Parameters **value** – new value for the *TopicDataQosPolicy*

inline const *DurabilityQosPolicy* &**durability**() const
Getter for *DurabilityQosPolicy*

Returns DurabilityQos reference

inline *DurabilityQosPolicy* &**durability**()
Getter for *DurabilityQosPolicy*

Returns DurabilityQos reference

inline void **durability**(const *DurabilityQosPolicy* &durability)
Setter for *DurabilityQosPolicy*

Parameters **durability** – new value for the *DurabilityQosPolicy*

inline const *DurabilityServiceQosPolicy* &**durability_service**() const
Getter for *DurabilityServiceQosPolicy*

Returns DurabilityServiceQos reference

inline *DurabilityServiceQosPolicy* &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Returns DurabilityServiceQos reference

inline void **durability_service**(const *DurabilityServiceQosPolicy* &durability_service)
Setter for *DurabilityServiceQosPolicy*

Parameters **durability_service** – new value for the *DurabilityServiceQosPolicy*

inline const *DeadlineQosPolicy* &**deadline**() const
Getter for *DeadlineQosPolicy*

Returns DeadlineQos reference

inline *DeadlineQosPolicy* &**deadline**()
Getter for *DeadlineQosPolicy*

Returns DeadlineQos reference

inline void **deadline**(const *DeadlineQosPolicy* &deadline)
Setter for *DeadlineQosPolicy*

Parameters **deadline** – new value for the *DeadlineQosPolicy*

inline const *LatencyBudgetQosPolicy* &**latency_budget**() const
Getter for *LatencyBudgetQosPolicy*

Returns LatencyBudgetQos reference

inline *LatencyBudgetQosPolicy* &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Returns LatencyBudgetQos reference

inline void **latency_budget**(const *LatencyBudgetQosPolicy* &latency_budget)
Setter for *LatencyBudgetQosPolicy*

Parameters **latency_budget** – new value for the *LatencyBudgetQosPolicy*

inline const *LivelinessQosPolicy* &**liveliness**() const
Getter for *LivelinessQosPolicy*

Returns LivelinessQos reference

inline *LivelinessQosPolicy* &**liveliness**()
Getter for *LivelinessQosPolicy*

Returns LivelinessQos reference

inline void **liveliness**(const *LivelinessQosPolicy* &liveliness)
Setter for *LivelinessQosPolicy*

Parameters **liveliness** – new value for the *LivelinessQosPolicy*

inline const *ReliabilityQosPolicy* &**reliability**() const
Getter for *ReliabilityQosPolicy*

Returns ReliabilityQos reference

inline *ReliabilityQosPolicy* &**reliability**()
Getter for *ReliabilityQosPolicy*

Returns ReliabilityQos reference

inline void **reliability**(const *ReliabilityQosPolicy* &reliability)
Setter for *ReliabilityQosPolicy*

Parameters **reliability** – new value for the *ReliabilityQosPolicy*

inline const *DestinationOrderQosPolicy* &**destination_order**() const
Getter for *DestinationOrderQosPolicy*

Returns DestinationOrderQos reference

inline *DestinationOrderQosPolicy* &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Returns DestinationOrderQos reference

inline void **destination_order**(const *DestinationOrderQosPolicy* &destination_order)
Setter for *DestinationOrderQosPolicy*

Parameters **destination_order** – new value for the *DestinationOrderQosPolicy*

inline const *HistoryQosPolicy* &**history**() const
Getter for *HistoryQosPolicy*

Returns HistoryQos reference

inline *HistoryQosPolicy* &**history**()
Getter for *HistoryQosPolicy*

Returns HistoryQos reference

inline void **history**(const *HistoryQosPolicy* &history)
Setter for *HistoryQosPolicy*

Parameters **history** – new value for the *HistoryQosPolicy*

inline const *ResourceLimitsQosPolicy* &**resource_limits**() const
Getter for *ResourceLimitsQosPolicy*

Returns ResourceLimitsQos reference

inline *ResourceLimitsQosPolicy* &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Returns ResourceLimitsQos reference

inline void **resource_limits**(const *ResourceLimitsQosPolicy* &resource_limits)
Setter for *ResourceLimitsQosPolicy*

Parameters **resource_limits** – new value for the *ResourceLimitsQosPolicy*

inline const *TransportPriorityQosPolicy* &**transport_priority**() const
Getter for *TransportPriorityQosPolicy*

Returns TransportPriorityQos reference

inline *TransportPriorityQosPolicy* &**transport_priority**()
Getter for *TransportPriorityQosPolicy*

Returns TransportPriorityQos reference

inline void **transport_priority**(const *TransportPriorityQosPolicy* &transport_priority)
Setter for *TransportPriorityQosPolicy*

Parameters **transport_priority** – new value for the *TransportPriorityQosPolicy*

inline const *LifespanQosPolicy* &**lifespan**() const
Getter for *LifespanQosPolicy*

Returns LifespanQos reference

inline *LifespanQosPolicy* &**lifespan**()
Getter for *LifespanQosPolicy*

Returns LifespanQos reference

inline void **lifespan**(const *LifespanQosPolicy* &lifespan)
Setter for *LifespanQosPolicy*

Parameters **lifespan** – new value for the *LifespanQosPolicy*

inline const *OwnershipQosPolicy* &**ownership**() const
Getter for *OwnershipQosPolicy*

Returns OwnershipQos reference

inline *OwnershipQosPolicy* &**ownership**()
Getter for *OwnershipQosPolicy*

Returns OwnershipQos reference

inline void **ownership**(const *OwnershipQosPolicy* &ownership)
Setter for *OwnershipQosPolicy*

Parameters **ownership** – new value for the *OwnershipQosPolicy*

inline const *DataRepresentationQosPolicy* &**representation**() const
Getter for *DataRepresentationQosPolicy*

Returns *DataRepresentationQosPolicy* reference

inline *DataRepresentationQosPolicy* &**representation**()
Getter for *DataRepresentationQosPolicy*

Returns *DataRepresentationQosPolicy* reference

inline void **representation**(const *DataRepresentationQosPolicy* &representation)
Setter for *DataRepresentationQosPolicy*

Parameters **representation** – new value for the *DataRepresentationQosPolicy*

const *TopicQos* eprosima::fastdds::dds::**TOPIC_QOS_DEFAULT**

TypeIdV1

class **TypeIdV1** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Class *TypeIdV1*

Public Functions

inline **TypeIdV1**()
Constructor without parameters.

inline **TypeIdV1**(const *TypeIdV1* &type)
Copy constructor.

Parameters **type** – Another instance of *TypeIdV1*

inline **TypeIdV1**(const fastrtps::types::TypeIdentifier &identifier)
Constructor using a TypeIndentifier.

Parameters **identifier** – TypeIdentifier to be set

inline **TypeIdV1**(*TypeIdV1* &&type)
Move constructor.

Parameters **type** – Another instance of *TypeIdV1*

virtual **~TypeIdV1**() override = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

inline const fastrtps::types::TypeIdentifier &**get**() const
Getter for the TypeIndentifier.

Returns TypeIdentifier reference

Public Members

fastrtps::types::TypeIdentifier **m_type_identifier**
Type Identifier.

TypeInformation

class **TypeInformation** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Class *xtypes::TypeInformation*

Public Functions

inline **TypeInformation**()
Constructor.

inline **TypeInformation**(const *TypeInformation* &type)
Copy constructor.

Parameters **type** – Another instance of *TypeInformation*

inline **TypeInformation**(const fastrtps::types::TypeInformation &info)
Constructor using a fastrtps::types::TypeInformation.

Parameters **info** – fastrtps::types::TypeInformation to be set

inline **TypeInformation**(*TypeInformation* &&type)
Move Constructor.

Parameters **type** – Another instance of *TypeInformation*

virtual **~TypeInformation**() override = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

inline bool **assigned**() const
Check if it is assigned.

Returns true if assigned, false if not

inline void **assigned**(bool value)
Setter for assigned boolean.

Parameters **value** – Boolean to be set

Public Members

fastrtps::types::TypeInformation **type_information**
Type Information.

TypeObjectV1

class **TypeObjectV1** : public eprosima::fastdds::dds::Parameter_t, public eprosima::fastdds::dds::*QosPolicy*
Class *TypeObjectV1*

Public Functions

inline **TypeObjectV1**()
Constructor.

inline **TypeObjectV1**(const *TypeObjectV1* &type)
Copy constructor.

Parameters **type** – Another instance of *TypeObjectV1*

inline **TypeObjectV1**(const fastrtps::types::TypeObject &type)
Constructor using a TypeObject.

Parameters **type** – TypeObject to be set

inline **TypeObjectV1**(*TypeObjectV1* &&type)
Move constructor.

Parameters **type** – Another instance of *TypeObjectV1*

virtual **~TypeObjectV1**() override = default
Destructor.

inline virtual void **clear**() override
Clears the *QosPolicy* object.

inline const fastrtps::types::TypeObject &**get**() const
Getter for the TypeObject.

Returns TypeObject reference

Public Members

fastrtps::types::TypeObject **m_type_object**
Type Object.

6.32.2 RTPS

eProsima Fast DDS Real-Time Publish-Subscribe (RTPS) layer API.

Attributes

BuiltinAttributes

class **BuiltinAttributes**
Class *BuiltinAttributes*, to define the behavior of the *RTPSParticipant* builtin protocols.

Public Members

DiscoverySettings **discovery_config**
Discovery protocol related attributes.

bool **use_WriterLivelinessProtocol** = true
Indicates to use the WriterLiveliness protocol.

TypeLookupSettings **typelookup_config**
TypeLookup Service settings.

NetworkConfigSet_t **network_configuration** = 0
Network Configuration.

LocatorList_t **metatrafficUnicastLocatorList**
Metatraffic Unicast Locator List.

LocatorList_t **metatrafficMulticastLocatorList**
Metatraffic Multicast Locator List.

fastdds::rtps::*ExternalLocators* **metatraffic_external_unicast_locators**
The collection of external locators to use for communication on metatraffic topics.

LocatorList_t **initialPeersList**
Initial peers.

MemoryManagementPolicy_t **readerHistoryMemoryPolicy** =
MemoryManagementPolicy_t::PREALLOCATED_WITH_REALLOC_MEMORY_MODE
Memory policy for builtin readers.

uint32_t **readerPayloadSize** = BUILTIN_DATA_MAX_SIZE
Maximum payload size for builtin readers.

MemoryManagementPolicy_t **writerHistoryMemoryPolicy** =
MemoryManagementPolicy_t::PREALLOCATED_WITH_REALLOC_MEMORY_MODE
Memory policy for builtin writers.

uint32_t **writerPayloadSize** = BUILTIN_DATA_MAX_SIZE
Maximum payload size for builtin writers.

uint32_t **mutation_tries** = 100u
Mutation tries if the port is being used.

bool **avoid_builtin_multicast** = true
Set to true to avoid multicast traffic on builtin endpoints.

c_default_RTPSParticipantAllocationAttributes

const *RTPSParticipantAllocationAttributes*
eprosima::fastrtps::rtps::**c_default_RTPSParticipantAllocationAttributes** =
RTPSParticipantAllocationAttributes()

DiscoveryProtocol

enum eprosima::fastrtps::rtps::**DiscoveryProtocol**
PDP subclass choice.

Values:

enumerator **NONE**
NO discovery whatsoever would be used.

Publisher and Subscriber defined with the same topic name would NOT be linked. All matching must be done manually through the addReaderLocator, addReaderProxy, addWriterProxy methods.

enumerator **SIMPLE**
Discovery works according to 'The Real-time Publish-Subscribe Protocol(RTPS) DDS Interoperability Wire Protocol Specification'.

enumerator **EXTERNAL**
A user defined PDP subclass object must be provided in the attributes that deals with the discovery.

Framework is not responsible of this object lifetime.

enumerator **CLIENT**
The participant will behave as a client concerning discovery operation.

Server locators should be specified as attributes.

enumerator **SERVER**
The participant will behave as a server concerning discovery operation.

Discovery operation is volatile (discovery handshake must take place if shutdown).

enumerator **BACKUP**
The participant will behave as a server concerning discovery operation.

Discovery operation persist on a file (discovery handshake wouldn't repeat if shutdown).

enumerator **SUPER_CLIENT**
The participant will behave as a client concerning all internal behaviour.

Remote servers will treat it as a server and will share every discovery information.

DiscoverySettings

class **DiscoverySettings**
Class *DiscoverySettings*, to define the attributes of the several discovery protocols available

Public Functions

inline const char *__getStaticEndpointXMLFilename__() const
Get the static endpoint XML filename

Returns Static endpoint XML filename

inline void __setStaticEndpointXMLFilename__(const char *str)
Set the static endpoint XML filename

Deprecated:

Parameters **str** – Static endpoint XML filename

inline void __static_edp_xml_config__(const char *str)
Set the static endpoint XML configuration.

Parameters **str** – URI specifying the static endpoint XML configuration. The string could contain a filename (file://) or the XML content directly (data://).

inline const char *static_edp_xml_config() const
Get the static endpoint XML configuration.

Returns URI specifying the static endpoint XML configuration. The string could contain a filename (file://) or the XML content directly (data://).

Public Members

DiscoveryProtocol_t **discoveryProtocol** = DiscoveryProtocol_t::SIMPLE
Chosen discovery protocol.

bool **use_SIMPLE_EndpointDiscoveryProtocol** = true
If set to true, SimpleEDP would be used.

bool **use_STATIC_EndpointDiscoveryProtocol** = false
If set to true, StaticEDP based on an XML file would be implemented. The XML filename must be provided.

Duration_t **leaseDuration** = {20, 0}
Lease Duration of the *RTPSParticipant*, indicating how much time remote RTPSParticipants should consider this *RTPSParticipant* alive.

Duration_t **leaseDuration_announcementperiod** = {3, 0}
The period for the *RTPSParticipant* to send its Discovery Message to all other discovered RTPSParticipants as well as to all Multicast ports.

InitialAnnouncementConfig **initial_announcements**
Initial announcements configuration.

SimpleEDPAttributes **m_simpleEDP**
Attributes of the SimpleEDP protocol.

PDPFactory **m_PDPfactory** = {}
function that returns a PDP object (only if EXTERNAL selected)

Duration_t **discoveryServer_client_syncperiod** = {0, 450 * 1000000}
The period for the *RTPSParticipant* to: send its Discovery Message to its servers check for EDP endpoints matching

eprosima::fastdds::rtps::*RemoteServerList_t* **m_DiscoveryServers**
Discovery Server settings, only needed if use_CLIENT_DiscoveryProtocol=true.

ParticipantFilteringFlags_t **ignoreParticipantFlags** = *ParticipantFilteringFlags*::*NO_FILTER*
Filtering participants out depending on location.

EndpointAttributes

class **EndpointAttributes**
Structure *EndpointAttributes*, describing the attributes associated with an RTPS *Endpoint*.

Public Functions

inline int16_t **getUserDefinedID**() const
Get the user defined ID

Returns User defined ID

inline int16_t **getEntityID**() const
Get the entity defined ID

Returns Entity ID

inline void **setUserDefinedID**(int16_t id)
Set the user defined ID

Parameters **id** – User defined ID to be set

inline void **setEntityID**(int16_t id)
Set the entity ID

Parameters **id** – Entity ID to be set

inline void **set_data_sharing_configuration**(DataSharingQosPolicy cfg)
Set the DataSharing configuration

Parameters **cfg** – Configuration to be set

inline const DataSharingQosPolicy &**data_sharing_configuration**() const
Get the DataSharing configuration

Returns Configuration of data sharing

Public Members

EndpointKind_t **endpointKind** = *EndpointKind_t*::WRITER
Endpoint kind, default value WRITER.

TopicKind_t **topicKind** = *TopicKind_t*::NO_KEY
Topic kind, default value NO_KEY.

ReliabilityKind_t **reliabilityKind** = *ReliabilityKind_t*::BEST_EFFORT
Reliability kind, default value BEST_EFFORT.

DurabilityKind_t **durabilityKind** = *DurabilityKind_t*::VOLATILE
Durability kind, default value VOLATILE.

GUID_t **persistence_guid**
GUID used for persistence.

fastdds::rtps::*ExternalLocators* **external_unicast_locators**
The collection of external locators to use for communication.

bool **ignore_non_matching_locators** = false
Whether locators that don't match with the announced locators should be kept.

LocatorList_t **unicastLocatorList**
Unicast locator list.

LocatorList_t **multicastLocatorList**
Multicast locator list.

LocatorList_t **remoteLocatorList**
Remote locator list.

PropertyPolicy **properties**
Properties.

OwnershipQosPolicyKind **ownershipKind** = SHARED_OWNERSHIP_QOS
Ownership.

ExternalLocators

using eprosima::fastdds::rtps::**ExternalLocators** = std::map<uint8_t, std::map<uint8_t, std::vector<*LocatorWithMask*>>, std::greater<uint8_t>>
A collection of *LocatorWithMask* grouped by externality and cost.

HistoryAttributes

class **HistoryAttributes**
Class *HistoryAttributes*, to specify the attributes of a *WriterHistory* or a *ReaderHistory*. This class is only intended to be used with the RTPS API. The Publisher-Subscriber API has other fields to define this values (HistoryQosPolicy and ResourceLimitsQosPolicy).

Public Functions

inline **HistoryAttributes**()
Default constructor.

inline **HistoryAttributes**(MemoryManagementPolicy_t memoryPolicy, uint32_t payload, int32_t initial, int32_t maxRes)
Constructor

Parameters

- **memoryPolicy** – Set whether memory can be dynamically reallocated or not
- **payload** – Maximum payload size. It is used when memory management policy is PREALLOCATED_MEMORY_MODE or PREALLOCATED_WITH_REALLOC_MEMORY_MODE.
- **initial** – Initial reserved caches. It is used when memory management policy is PREALLOCATED_MEMORY_MODE or PREALLOCATED_WITH_REALLOC_MEMORY_MODE.

- **maxRes** – Maximum reserved caches.

inline **HistoryAttributes**(MemoryManagementPolicy_t memoryPolicy, uint32_t payload, int32_t initial, int32_t maxRes, int32_t extra)

Constructor

Parameters

- **memoryPolicy** – Set whether memory can be dynamically reallocated or not
- **payload** – Maximum payload size. It is used when memory management policy is PREALLOCATED_MEMORY_MODE or PREALLOCATED_WITH_REALLOC_MEMORY_MODE.
- **initial** – Initial reserved caches. It is used when memory management policy is PREALLOCATED_MEMORY_MODE or PREALLOCATED_WITH_REALLOC_MEMORY_MODE.
- **maxRes** – Maximum reserved caches.
- **extra** – Extra reserved caches.

Public Members

MemoryManagementPolicy_t **memoryPolicy**

Memory management policy.

uint32_t **payloadMaxSize**

Maximum payload size of the history, default value 500.

int32_t **initialReservedCaches**

Number of the initial Reserved Caches, default value 500.

int32_t **maximumReservedCaches**

Maximum number of reserved caches. Default value is 0 that indicates to keep reserving until something breaks.

int32_t **extraReservedCaches**

Number of extra caches that can be reserved for other purposes than the history. For example, on a full history, the writer could give as many as these to be used by the application but they will not be able to be inserted in the history unless some cache from the history is released.

Default value is 1.

InitialAnnouncementConfig

struct **InitialAnnouncementConfig**

Struct *InitialAnnouncementConfig* defines the behavior of the *RTPSParticipant* initial announcements.

Public Members

uint32_t **count** = 5u
Number of initial announcements with specific period (default 5)

Duration_t **period** = {0, 100000000u}
Specific period for initial announcements (default 100ms)

ParticipantFilteringFlags

enum eprosima::fastrtps::rtps::**ParticipantFilteringFlags**
Filtering flags when discovering participants.

Values:

enumerator **NO_FILTER**

enumerator **FILTER_DIFFERENT_HOST**

enumerator **FILTER_DIFFERENT_PROCESS**

enumerator **FILTER_SAME_PROCESS**

PropertyPolicy

class **PropertyPolicy**

Public Functions

inline const *PropertySeq* &**properties**() const
Get properties.

inline *PropertySeq* &**properties**()
Set properties.

inline const *BinaryPropertySeq* &**binary_properties**() const
Get binary_properties.

inline *BinaryPropertySeq* &**binary_properties**()
Set binary_properties.

PropertyPolicyHelper

class **PropertyPolicyHelper**

Public Static Functions

static *PropertyPolicy* **get_properties_with_prefix**(const *PropertyPolicy* &property_policy, const std::string &prefix)
Returns only the properties whose name starts with the prefix.

Prefix is removed in returned properties.

Parameters

- **property_policy** – *PropertyPolicy* where properties will be searched.
- **prefix** – Prefix used to search properties.

Returns A copy of properties whose name starts with the prefix.

static size_t **length**(const *PropertyPolicy* &property_policy)
Get the length of the property_policy.

static std::string ***find_property**(*PropertyPolicy* &property_policy, const std::string &name)
Look for a property_policy by name.

static const std::string ***find_property**(const *PropertyPolicy* &property_policy, const std::string &name)
Retrieves a property_policy by name.

static const *Property* ***get_property**(const *PropertyPolicy* &property_policy, const std::string &name)
Retrieves a property by name.

Parameters

- **property_policy** – *PropertyPolicy* where the property will be searched.
- **name** – Name of the property to be searched.

Returns A pointer to the property if found, nullptr otherwise.

ReaderAttributes

class **ReaderAttributes**
Class *ReaderAttributes*, to define the attributes of a *RTPSReader*.

Public Members

EndpointAttributes **endpoint**
Attributes of the associated endpoint.

ReaderTimes **times**
Times associated with this reader (only for stateful readers)

LivelinessQosPolicyKind **liveliness_kind_**
Liveliness kind.

Duration_t **liveliness_lease_duration**
Liveliness lease duration.

bool **expectsInlineQos**
Indicates if the reader expects Inline qos, default value 0.

bool **disable_positive_acks**
Disable positive ACKs.

ResourceLimitedContainerConfig **matched_writers_allocation**
Define the allocation behaviour for matched-writer-dependent collections.

fastdds::rtps::ThreadSettings **data_sharing_listener_thread**
Thread settings for the data-sharing listener thread.

ReaderTimes

class **ReaderTimes**
Class *ReaderTimes*, defining the times associated with the Reliable Readers events.

Public Members

Duration_t **initialAcknackDelay**
Initial AckNack delay. Default value 70ms.

Duration_t **heartbeatResponseDelay**
Delay to be applied when a HEARTBEAT message is received, default value 5ms.

RemoteLocatorsAllocationAttributes

struct **RemoteLocatorsAllocationAttributes**
Holds limits for collections of remote locators.

Public Members

size_t **max_unicast_locators** = 4u
Maximum number of unicast locators per remote entity.

This attribute controls the maximum number of unicast locators to keep for each discovered remote entity (be it a participant, reader of writer). It is recommended to use the highest number of local addresses found on all the systems belonging to the same domain as this participant.

size_t **max_multicast_locators** = 1u
Maximum number of multicast locators per remote entity.

This attribute controls the maximum number of multicast locators to keep for each discovered remote entity (be it a participant, reader of writer). The default value of 1 is usually enough, as it doesn't make sense to add more than one multicast locator per entity.

RemoteServerAttributes

class **RemoteServerAttributes**
Class *RemoteServerAttributes*, to define the attributes of the Discovery Server Protocol.

Public Members

LocatorList **metatrafficUnicastLocatorList**
Metatraffic Unicast Locator List.

LocatorList **metatrafficMulticastLocatorList**
Metatraffic Multicast Locator List.

fastrtps::rtps::*GuidPrefix_t* **guidPrefix**
Guid prefix.

RemoteServerList_t

typedef std::list<*RemoteServerAttributes*> `eprosima::fastdds::rtps::`**RemoteServerList_t**

RTPSParticipantAllocationAttributes

struct **RTPSParticipantAllocationAttributes**
Holds allocation limits affecting collections managed by a participant.

Public Functions

inline ResourceLimitedContainerConfig **total_readers**() const

Returns the allocation config for the total of readers in the system (participants * readers)

inline ResourceLimitedContainerConfig **total_writers**() const

Returns the allocation config for the total of writers in the system (participants * writers)

Public Members

RemoteLocatorsAllocationAttributes **locators**
Holds limits for collections of remote locators.

ResourceLimitedContainerConfig **participants**
Defines the allocation behaviour for collections dependent on the total number of participants.

ResourceLimitedContainerConfig **readers**
Defines the allocation behaviour for collections dependent on the total number of readers per participant.

ResourceLimitedContainerConfig **writers**
Defines the allocation behaviour for collections dependent on the total number of writers per participant.

SendBuffersAllocationAttributes **send_buffers**
Defines the allocation behaviour for the send buffer manager.

VariableLengthDataLimits **data_limits**
Holds limits for variable-length data.

fastdds::rtps::*ContentFilterProperty*::*AllocationConfiguration* **content_filter**
Defines the allocation behavior of content filter discovery information.

RTPSParticipantAttributes

class **RTPSParticipantAttributes**
Class *RTPSParticipantAttributes* used to define different aspects of a *RTPSParticipant*.

Public Functions

void **setup_transports**(fastdds::rtps::BuiltinTransports transports, const fastdds::rtps::BuiltinTransportsOptions &options = fastdds::rtps::BuiltinTransportsOptions())
Provides a way of easily configuring transport related configuration on certain pre-defined scenarios with certain options.

Parameters

- **transports** – Defines the transport configuration scenario to setup.
- **options** – Defines the options to be used in the transport configuration.

inline void **setName**(const char *nam)
Set the name of the participant.

inline const char ***getName**() const
Get the name of the participant.

Public Members

LocatorList_t **defaultUnicastLocatorList**
Default list of Unicast Locators to be used for any *Endpoint* defined inside this *RTPSParticipant* in the case that it was defined with NO UnicastLocators. At least ONE locator should be included in this list.

LocatorList_t **defaultMulticastLocatorList**
Default list of Multicast Locators to be used for any *Endpoint* defined inside this *RTPSParticipant* in the case that it was defined with NO MulticastLocators. This is usually left empty.

fastdds::rtps::*ExternalLocators* **default_external_unicast_locators**
The collection of external locators to use for communication on user created topics.

bool **ignore_non_matching_locators** = false
: Whether locators that don't match with the announced locators should be kept.

uint32_t **sendSocketBufferSize** = 0
: Send socket buffer size for the send resource.

  Zero value indicates to use default system buffer size. Default value: 0.

uint32_t **listenSocketBufferSize** = 0
: Listen socket buffer for all listen resources.

  Zero value indicates to use default system buffer size. Default value: 0.

fastdds::rtps::NetmaskFilterKind **netmaskFilter** = fastdds::rtps::NetmaskFilterKind::AUTO
: Netmask filter configuration.

GuidPrefix_t **prefix**
: Optionally allows user to define the *GuidPrefix_t*.

BuiltinAttributes **builtin**
: Builtin parameters.

PortParameters **port**
: Port Parameters.

std::vector<*octet*> **userData**
: User Data of the participant.

int32_t **participantID** = -1
: Participant ID.

ThroughputControllerDescriptor **throughputController**
: Throughput controller parameters. Leave default for uncontrolled flow.

  Deprecated:
  : Use flow_controllers on *RTPSParticipantAttributes*

std::vector<std::shared_ptr<fastdds::rtps::*TransportDescriptorInterface*>> **userTransports**
: User defined transports to use alongside or in place of builtins.

bool **useBuiltinTransports** = true
: Set as false to disable the creation of the default transports.

RTPSParticipantAllocationAttributes **allocation**
: Holds allocation limits affecting collections managed by a participant.

PropertyPolicy **properties**
: *Property* policies.

FlowControllerDescriptorList **flow_controllers**
: Flow controllers.

fastdds::rtps::ThreadSettings **builtin_controllers_sender_thread**
Thread settings for the builtin flow controllers sender threads.

fastdds::rtps::ThreadSettings **timed_events_thread**
Thread settings for the timed events thread.

fastdds::rtps::ThreadSettings **discovery_server_thread**
Thread settings for the discovery server thread.

fastdds::rtps::ThreadSettings **builtin_transports_reception_threads**
Thread settings for the builtin transports reception threads.

fastdds::rtps::ThreadSettings **security_log_thread**
Thread settings for the security log thread.

uint32_t **max_msg_size_no_frag** = 0
Maximum message size used to avoid fragmentation, set ONLY in LARGE_DATA.

If this value is not zero, the network factory will allow the initialization of UDP transports with maxMessageSize higher than 65500K.

RTPSWriterPublishMode

enum `eprosima::fastrtps::rtps::`**RTPSWriterPublishMode**
Values:

enumerator **SYNCHRONOUS_WRITER**

enumerator **ASYNCHRONOUS_WRITER**

SendBuffersAllocationAttributes

struct **SendBuffersAllocationAttributes**
Holds limits for send buffers allocations.

Public Members

size_t **preallocated_number** = 0u
Initial number of send buffers to allocate.

This attribute controls the initial number of send buffers to be allocated. The default value of 0 will perform an initial guess of the number of buffers required, based on the number of threads from which a send operation could be started.

bool **dynamic** = false
Whether the number of send buffers is allowed to grow.

This attribute controls how the buffer manager behaves when a send buffer is not available. When true, a new buffer will be created. When false, it will wait for a buffer to be returned. This is a trade-off between latency and dynamic allocations.

SimpleEDPAttributes

class **SimpleEDPAttributes**
Class *SimpleEDPAttributes*, to define the attributes of the Simple *Endpoint* Discovery Protocol.

Public Members

bool **use_PublicationWriterANDSubscriptionReader**
Default value true.

bool **use_PublicationReaderANDSubscriptionWriter**
Default value true.

TypeLookupSettings

class **TypeLookupSettings**
TypeLookupService settings.

Public Members

bool **use_client** = false
Indicates to use the TypeLookup Service client endpoints.

bool **use_server** = false
Indicates to use the TypeLookup Service server endpoints.

VariableLengthDataLimits

struct **VariableLengthDataLimits**
Holds limits for variable-length data.

Public Members

size_t **max_properties** = 0
Defines the maximum size (in octets) of properties data in the local or remote participant.

size_t **max_user_data** = 0
Defines the maximum size (in octets) of user data in the local or remote participant.

size_t **max_partitions** = 0
Defines the maximum size (in octets) of partitions data.

size_t **max_datasharing_domains** = 0
Defines the maximum size (in elements) of the list of data sharing domain IDs.

WriterAttributes

class **WriterAttributes**
Class *WriterAttributes*, defining the attributes of a *RTPSWriter*.

Public Members

EndpointAttributes **endpoint**
Attributes of the associated endpoint.

WriterTimes **times**
Writer Times (only used for RELIABLE).

fastrtps::LivelinessQosPolicyKind **liveliness_kind**
Liveliness kind.

Duration_t **liveliness_lease_duration**
Liveliness lease duration.

Duration_t **liveliness_announcement_period**
Liveliness announcement period.

RTPSWriterPublishMode **mode**
Indicates if the Writer is synchronous or asynchronous.

ThroughputControllerDescriptor **throughputController**
Throughput controller, always the last one to apply.

Deprecated:
Use flow_controllers on *RTPSParticipantAttributes*

bool **disable_heartbeat_piggyback**
Disable the sending of heartbeat piggybacks.

ResourceLimitedContainerConfig **matched_readers_allocation**
Define the allocation behaviour for matched-reader-dependent collections.

bool **disable_positive_acks**
Disable the sending of positive ACKs.

Duration_t **keep_duration**
Keep duration to keep a sample before considering it has been acked.

const char ***flow_controller_name** = fastdds::rtps::*FASTDDS_FLOW_CONTROLLER_DEFAULT*
Flow controller name. Default: *fastdds::rtps::FASTDDS_FLOW_CONTROLLER_DEFAULT*.

WriterTimes

struct **WriterTimes**
Struct *WriterTimes*, defining the times associated with the Reliable Writers events.

Public Members

Duration_t **initialHeartbeatDelay**
Initial heartbeat delay. Default value ~11ms.

Duration_t **heartbeatPeriod**
Periodic HB period, default value 3s.

Duration_t **nackResponseDelay**
Delay to apply to the response of a ACKNACK message, default value ~5ms.

Duration_t **nackSupressionDuration**
This time allows the *RTPSWriter* to ignore nack messages too soon after the data as sent, default value 0s.

Builtin data

ContentFilterProperty

class **ContentFilterProperty**
Information about the content filter being applied by a reader.

Public Functions

inline explicit **ContentFilterProperty**(const *AllocationConfiguration* &config)
Construct a *ContentFilterProperty*.

Parameters **config** – Allocation configuration for the new object.

Public Members

fastrtps::string_255 **content_filtered_topic_name**
Name of the content filtered topic on which the reader was created.

fastrtps::string_255 **related_topic_name**
Name of the related topic being filtered.

fastrtps::string_255 **filter_class_name**
Class name of the filter being used. May be empty to indicate the *ContentFilterProperty* is not present.

std::string **filter_expression**
Filter expression indicating which content the reader wants to receive. May be empty to indicate the *ContentFilterProperty* is not present.

fastrtps::ResourceLimitedVector<fastrtps::string_255, std::true_type> **expression_parameters**
List of values for the parameters present on the filter expression.

struct **AllocationConfiguration**
Allocation configuration for a *ContentFilterProperty*.

Public Members

size_t **expression_initial_size** = 0
Preallocated size of the filter expression.

fastrtps::ResourceLimitedContainerConfig **expression_parameters** = {0, 100, 1}
Allocation configuration for the list of expression parameters.

Common

BinaryProperty

BinaryProperty

class **BinaryProperty**

BinaryPropertyHelper

class **BinaryPropertyHelper**

BinaryPropertySeq

typedef std::vector<*BinaryProperty*> eprosima::fastrtps::rtps::**BinaryPropertySeq**

CacheChange

CacheChange_t

struct **CacheChange_t**
Structure *CacheChange_t*, contains information on a specific CacheChange.

Public Functions

inline **CacheChange_t**()
Default constructor.

Creates an empty *CacheChange_t*.

inline **CacheChange_t**(uint32_t payload_size, bool is_untyped = false)
Constructor with payload size

Parameters

- **payload_size** – Serialized payload size
- **is_untyped** – Flag to mark the change as untyped.

inline bool **copy**(const *CacheChange_t* *ch_ptr)
Copy a different change into this one.

All the elements are copied, included the data, allocating new memory.

Parameters **ch_ptr** – **[in]** Pointer to the change.

Returns True if correct.

inline void **copy_not_memcpy**(const *CacheChange_t* *ch_ptr)
Copy information form a different change into this one.

All the elements are copied except data.

Parameters **ch_ptr** – **[in]** Pointer to the change.

inline uint32_t **getFragmentCount**() const
Get the number of fragments this change is split into.

Returns number of fragments.

inline uint16_t **getFragmentSize**() const
Get the size of each fragment this change is split into.

Returns size of fragment (0 means change is not fragmented).

inline bool **is_fully_assembled**()
Checks if all fragments have been received.

Returns true when change is fully assembled (i.e. no missing fragments).

inline bool **contains_first_fragment**()
Checks if the first fragment is present.

Returns true when it contains the first fragment. In other case, false.

inline void **get_missing_fragments**(*FragmentNumberSet_t* &frag_sns)
Fills a FragmentNumberSet_t with the list of missing fragments.

Parameters **frag_sns** – **[out]** FragmentNumberSet_t where result is stored.

inline void **setFragmentSize**(uint16_t fragment_size, bool create_fragment_list = false)
Set fragment size for this change.

Remark

Parameter create_fragment_list should only be true when receiving the first fragment of a change.

Parameters

- **fragment_size** – Size of fragments.
- **create_fragment_list** – Whether to create missing fragments list or not.

Public Members

ChangeKind_t **kind** = ALIVE
Kind of change, default value ALIVE.

GUID_t **writerGUID** = {}
GUID_t of the writer that generated this change.

InstanceHandle_t **instanceHandle** = {}
Handle of the data associated with this change.

SequenceNumber_t **sequenceNumber** = {}
SequenceNumber of the change.

SerializedPayload_t **serializedPayload** = {}
Serialized Payload associated with the change.

SerializedPayload_t **inline_qos** = {}
CDR serialization of inlined QoS for this change.

bool **isRead** = false
Indicates if the cache has been read (only used in READERS)

Time_t **sourceTimestamp** = {}
Source TimeStamp.

fastdds::rtps::*VendorId_t* **vendor_id** = c_VendorId_Unknown
Vendor Id of the writer that generated this change.

ChangeForReader_t

class **ChangeForReader_t**
Struct *ChangeForReader_t* used to represent the state of a specific change with respect to a specific reader, as well as its relevance.

Public Functions

inline *CacheChange_t* ***getChange**() const
Get the cache change

Returns Cache change

ChangeForReaderCmp

struct **ChangeForReaderCmp**

ChangeForReaderStatus_t

enum eprosima::fastrtps::rtps::**ChangeForReaderStatus_t**
Enum ChangeForReaderStatus_t, possible states for a *CacheChange_t* in a ReaderProxy.

Values:

enumerator **UNSENT**
UNSENT.

enumerator **REQUESTED**
REQUESTED.

enumerator **UNACKNOWLEDGED**
UNACKNOWLEDGED.

enumerator **ACKNOWLEDGED**
ACKNOWLEDGED.

enumerator **UNDERWAY**
UNDERWAY.

ChangeKind_t

enum eprosima::fastrtps::rtps::**ChangeKind_t**
, different types of *CacheChange_t*.

Values:

enumerator **ALIVE**
ALIVE.

enumerator **NOT_ALIVE_DISPOSED**
NOT_ALIVE_DISPOSED.

enumerator **NOT_ALIVE_UNREGISTERED**
NOT_ALIVE_UNREGISTERED.

enumerator **NOT_ALIVE_DISPOSED_UNREGISTERED**
NOT_ALIVE_DISPOSED_UNREGISTERED.

CDRMessage

CDRMessage_t

struct **CDRMessage_t**
Structure *CDRMessage_t*, contains a serialized message.

Public Functions

inline explicit **CDRMessage_t**(uint32_t size)
Constructor with maximum size

Parameters **size** – Maximum size

inline explicit **CDRMessage_t**(const *SerializedPayload_t* &payload)
Constructor to wrap a serialized payload

Parameters **payload** – Payload to wrap

Public Members

octet ***buffer**
Pointer to the buffer where the data is stored.

uint32_t **pos**
Read or write position.

uint32_t **max_size**
Max size of the message.

uint32_t **reserved_size**
Size allocated on buffer. May be higher than max_size.

uint32_t **length**
Current length of the message.

Endianness_t **msg_endian**
Endianness of the message.

Macro definitions (#define)

RTPSMESSAGE_DEFAULT_SIZE 10500
Max size of RTPS message in bytes.

RTPSMESSAGE_COMMON_RTPS_PAYLOAD_SIZE 536

RTPSMESSAGE_COMMON_DATA_PAYLOAD_SIZE 10000

RTPSMESSAGE_HEADER_SIZE 20

RTPSMESSAGE_SUBMESSAGEHEADER_SIZE 4

RTPSMESSAGE_DATA_EXTRA_INLINEQOS_SIZE 4

RTPSMESSAGE_INFOTS_SIZE 12

RTPSMESSAGE_OCTETSTOINLINEQOS_DATASUBMSG 16

RTPSMESSAGE_OCTETSTOINLINEQOS_DATAFRAGSUBMSG 28

RTPSMESSAGE_DATA_MIN_LENGTH 24

CDRSerialization

constexpr eprosima::fastcdr::CdrVersion `eprosima::fastdds::rtps::`**DEFAULT_XCDR_VERSION** = {eprosima::fastcdr::CdrVersion::XCDRv1}
Default XCDR encoding version used in Fast DDS.

EntityId

Const values

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_Unknown** = ENTITYID_UNKNOWN

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_SPDPReader** = ENTITYID_SPDP_BUILTIN_RTPSParticipant_READER

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_SPDPWriter** = ENTITYID_SPDP_BUILTIN_RTPSParticipant_WRITER

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_SEDPPubWriter** = ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_SEDPPubReader** = ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_SEDPSubWriter** = ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_SEDPSubReader** = ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_RTPSParticipant** = ENTITYID_RTPSParticipant

const *EntityId_t* `eprosima::fastrtps::rtps::`**c_EntityId_WriterLiveliness** = ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_ReaderLiveliness** = ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_READER

const *EntityId_t* eprosima::fastrtps::rtps::**participant_stateless_message_writer_entity_id** = ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**participant_stateless_message_reader_entity_id** = ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_READER

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_request_writer** = ENTITYID_TL_SVC_REQ_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_request_reader** = ENTITYID_TL_SVC_REQ_READER

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_reply_writer** = ENTITYID_TL_SVC_REPLY_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_reply_reader** = ENTITYID_TL_SVC_REPLY_READER

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_publications_secure_writer** = ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_publications_secure_reader** = ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_READER

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_subscriptions_secure_writer** = ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_subscriptions_secure_reader** = ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_READER

const *EntityId_t* eprosima::fastrtps::rtps::**participant_volatile_message_secure_writer_entity_id** = ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**participant_volatile_message_secure_reader_entity_id** = ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_READER

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_WriterLivelinessSecure** = ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_WRITER

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_ReaderLivelinessSecure** = ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_READER

Macro definitions (#define)

ENTITYID_UNKNOWN 0x00000000

ENTITYID_RTPSParticipant 0x000001c1

ENTITYID_SEDP_BUILTIN_TOPIC_WRITER 0x000002c2

ENTITYID_SEDP_BUILTIN_TOPIC_READER 0x000002c7

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER 0x000003c2

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER 0x000003c7

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER 0x000004c2

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER 0x000004c7

ENTITYID_SPDP_BUILTIN_RTPSParticipant_WRITER 0x000100c2

ENTITYID_SPDP_BUILTIN_RTPSParticipant_READER 0x000100c7

ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_WRITER 0x000200C2

ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_READER 0x000200C7

ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_WRITER 0x000201C3

ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_READER 0x000201C4

ENTITYID_TL_SVC_REQ_WRITER 0x000300C3

ENTITYID_TL_SVC_REQ_READER 0x000300C4

ENTITYID_TL_SVC_REPLY_WRITER 0x000301C3

ENTITYID_TL_SVC_REPLY_READER 0x000301C4

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_WRITER 0xff0003c2

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_READER 0xff0003c7

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_WRITER 0xff0004c2

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_READER 0xff0004c7

ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_WRITER 0xff0200c2

ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_READER 0xff0200c7

ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_WRITER 0xff0202C3

ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_READER 0xff0202C4

ENTITYID_SPDP_RELIABLE_BUILTIN_PARTICIPANT_SECURE_WRITER 0xff0101c2

ENTITYID_SPDP_RELIABLE_BUILTIN_PARTICIPANT_SECURE_READER 0xff0101c7

EntityId_t

struct **EntityId_t**
Structure *EntityId_t*, entity id part of *GUID_t*.

Public Functions

inline **EntityId_t**()
Default constructor. Unknown entity.

inline **EntityId_t**(uint32_t id)
Main constructor.

Parameters **id** – Entity id

inline **EntityId_t**(const *EntityId_t* &id)
Copy constructor.

inline **EntityId_t**(*EntityId_t* &&id)
Move constructor.

inline *EntityId_t* &**operator=**(uint32_t id)
Assignment operator.

Parameters **id** – Entity id to copy

inline uint32_t **to_uint32**() const
conversion to uint32_t

Returns uint32_t representation

inline bool **operator<**(const *EntityId_t* &other) const
Entity Id minor operator

Parameters **other** – Second entity id to compare

Returns True if `other` is higher than this

Public Static Functions

static inline int **cmp**(const *EntityId_t* &entity1, const *EntityId_t* &entity2)
Entity Id compare static method.

Parameters

- **entity1** – First entity id to compare
- **entity2** – Second entity id to compare

Returns 0 if `entity1` is equal to `entity2` .

Returns < 0 if `entity1` is lower than `entity2` .

Returns > 0 if `entity1` is higher than `entity2` .

EntityId_t Operators

inline bool `eprosima::fastrtps::rtps::`**operator==**(*EntityId_t* &id1, const uint32_t id2)
Entity Id comparison operator

Parameters

- **id1** – EntityId to compare
- **id2** – ID prefix to compare

Returns True if equal

inline bool `eprosima::fastrtps::rtps::`**operator==**(const *EntityId_t* &id1, const *EntityId_t* &id2)
Entity Id comparison operator

Parameters

- **id1** – First EntityId to compare
- **id2** – Second EntityId to compare

Returns True if equal

inline bool `eprosima::fastrtps::rtps::`**operator!=**(const *EntityId_t* &id1, const *EntityId_t* &id2)
Guid prefix comparison operator

Parameters

- **id1** – First EntityId to compare

- **id2** – Second EntityId to compare

Returns True if not equal

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *EntityId_t* &enI)

inline std::istream &eprosima::fastrtps::rtps::**operator>>**(std::istream &input, *EntityId_t* &enP)

FragmentNumber

FragmentNumber_t

using eprosima::fastrtps::rtps::**FragmentNumber_t** = uint32_t

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *FragmentNumberSet_t* &fns)

FragmentNumberSet_t

using eprosima::fastrtps::rtps::**FragmentNumberSet_t** = BitmapRange<*FragmentNumber_t*>

Structure FragmentNumberSet_t, contains a group of fragmentnumbers.

Guid

c_Guid_Unknown

const *GUID_t* eprosima::fastrtps::rtps::**c_Guid_Unknown**

GUID_t

struct **GUID_t**

Structure *GUID_t*, entity identifier, unique in DDS-RTPS Domain.

Public Functions

inline **GUID_t**() noexcept

Default constructor.

Contructs an unknown GUID.

inline **GUID_t**(const *GuidPrefix_t* &guid_prefix, uint32_t id) noexcept

Construct

Parameters

- **guid_prefix** – Guid prefix

- **id** – Entity id

inline **GUID_t**(const *GuidPrefix_t* &guid_prefix, const *EntityId_t* &entity_id) noexcept

Parameters

- **guid_prefix** – Guid prefix
- **entity_id** – Entity id

inline bool **is_on_same_host_as**(const *GUID_t* &other_guid) const
Checks whether this guid is from an entity on the same host as another guid.

Note: This method assumes the value of `other_guid` was originally assigned by Fast-DDS vendor.

Parameters **other_guid** – *GUID_t* to compare to.

Returns true when this guid is on the same host, false otherwise.

inline bool **is_from_this_host**() const
Checks whether this guid is from a (Fast-DDS) entity created on this host (from where this method is called).

Returns true when this guid is from a (Fast-DDS) entity created on this host, false otherwise.

inline bool **is_on_same_process_as**(const *GUID_t* &other_guid) const
Checks whether this guid is for an entity on the same host and process as another guid.

Note: This method assumes the value of `other_guid` was originally assigned by Fast-DDS vendor.

Parameters **other_guid** – *GUID_t* to compare to.

Returns true when this guid is on the same host and process, false otherwise.

inline bool **is_from_this_process**() const
Checks whether this guid is from a (Fast-DDS) entity created on this process (from where this method is called).

Returns true when this guid is from a (Fast-DDS) entity created on this process, false otherwise.

inline bool **is_builtin**() const
Checks whether this guid corresponds to a builtin entity.

Returns true when this guid corresponds to a builtin entity, false otherwise.

Public Members

GuidPrefix_t **guidPrefix**
Guid prefix.

EntityId_t **entityId**
Entity id.

GUID_t Operators

inline bool eprosima::fastrtps::rtps::**operator==**(const *GUID_t* &g1, const *GUID_t* &g2)
GUID comparison operator

Parameters

- **g1** – First GUID to compare
- **g2** – Second GUID to compare

Returns True if equal

inline bool eprosima::fastrtps::rtps::**operator!=**(const *GUID_t* &g1, const *GUID_t* &g2)
GUID comparison operator

Parameters

- **g1** – First GUID to compare
- **g2** – Second GUID to compare

Returns True if not equal

inline bool eprosima::fastrtps::rtps::**operator<**(const *GUID_t* &g1, const *GUID_t* &g2)

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *GUID_t* &guid)
Stream operator, prints a GUID.

Parameters

- **output** – Output stream.
- **guid** – *GUID_t* to print.

Returns Stream operator.

inline std::istream &eprosima::fastrtps::rtps::**operator>>**(std::istream &input, *GUID_t* &guid)
Stream operator, retrieves a GUID.

Parameters

- **input** – Input stream.
- **guid** – *GUID_t* to print.

Returns Stream operator.

GuidPrefix

c_GuidPrefix_Unknown

const *GuidPrefix_t* eprosima::fastrtps::rtps::**c_GuidPrefix_Unknown**

GuidPrefix_t

struct **GuidPrefix_t**
Structure *GuidPrefix_t*, Guid Prefix of *GUID_t*.

Public Functions

inline **GuidPrefix_t**()
Default constructor. Set the Guid prefix to 0.

bool **is_on_same_host_as**(const *GuidPrefix_t* &other_guid_prefix) const
Checks whether this guid prefix is from an entity on the same host as another guid prefix.

Note: This method assumes the value of `other_guid_prefix` was originally assigned by Fast-DDS vendor.

Parameters `other_guid_prefix` – *GuidPrefix_t* to compare to.

Returns true when this guid prefix is on the same host, false otherwise.

bool **is_from_this_host**() const
Checks whether this guid prefix is from a (Fast-DDS) entity created on this host (from where this method is called).

Returns true when this guid prefix is from a (Fast-DDS) entity created on this host, false otherwise.

bool **is_on_same_process_as**(const *GuidPrefix_t* &other_guid_prefix) const
Checks whether this guid prefix is for an entity on the same host and process as another guid prefix.

Note: This method assumes the value of `other_guid_prefix` was originally assigned by Fast-DDS vendor.

Parameters `other_guid_prefix` – *GuidPrefix_t* to compare to.

Returns true when this guid prefix is on the same host and process, false otherwise.

bool **is_from_this_process**() const
Checks whether this guid prefix is from a (Fast-DDS) entity created on this host and process (from where this method is called).

Returns true when this guid prefix is from a (Fast-DDS) entity created on this host and process, false otherwise.

inline bool **operator==**(const *GuidPrefix_t* &prefix) const
Guid prefix comparison operator

Parameters `prefix` – guid prefix to compare

Returns True if the guid prefixes are equal

inline bool **operator!=**(const *GuidPrefix_t* &prefix) const
Guid prefix comparison operator

Parameters `prefix` – Second guid prefix to compare

Returns True if the guid prefixes are not equal

inline bool **operator<**(const *GuidPrefix_t* &prefix) const
Guid prefix minor operator

Parameters **prefix** – Second guid prefix to compare

Returns True if prefix is higher than this

Public Static Functions

static inline int **cmp**(const *GuidPrefix_t* &prefix1, const *GuidPrefix_t* &prefix2)
Guid Prefix compare static method.

Parameters

- **prefix1** – First guid prefix to compare
- **prefix2** – Second guid prefix to compare

Returns 0 if `prefix1` is equal to `prefix2` .

Returns < 0 if `prefix1` is lower than `prefix2` .

Returns > 0 if `prefix1` is higher than `prefix2` .

GuidPrefix_t Operators

inline std::ostream &`eprosima::fastrtps::rtps::`**operator<<**(std::ostream &output, const *GuidPrefix_t* &guiP)

inline std::istream &`eprosima::fastrtps::rtps::`**operator>>**(std::istream &input, *GuidPrefix_t* &guiP)

InstanceHandle

c_InstanceHandle_Unknown

const *InstanceHandle_t* `eprosima::fastrtps::rtps::`**c_InstanceHandle_Unknown**

InstanceHandle_t

struct **InstanceHandle_t**
Struct *InstanceHandle_t*, used to contain the key for WITH_KEY topics.

Public Functions

InstanceHandle_t &**operator=**(const *InstanceHandle_t* &ihandle) noexcept = default
Assignment operator

Parameters **ihandle** – Instance handle to copy the data from

inline *InstanceHandle_t* &**operator=**(const *GUID_t* &guid) noexcept
Assignment operator

Parameters **guid** – GUID to copy the data from

inline bool **isDefined**() const noexcept
Know if the instance handle is defined

Returns True if the values are not zero.

Public Members

InstanceHandleValue_t **value**
Value.

InstanceHandle_t Operators

inline bool eprosima::fastrtps::rtps::**operator==**(const *InstanceHandle_t* &ihandle1, const *InstanceHandle_t* &ihandle2) noexcept
Comparison operator

Parameters

- **ihandle1** – First *InstanceHandle_t* to compare
- **ihandle2** – Second *InstanceHandle_t* to compare

Returns True if equal

inline bool eprosima::fastrtps::rtps::**operator!=**(const *InstanceHandle_t* &ihandle1, const *InstanceHandle_t* &ihandle2) noexcept
Comparison operator.

Parameters

- **ihandle1** – First *InstanceHandle_t* to compare
- **ihandle2** – Second *InstanceHandle_t* to compare

Returns True if not equal

inline bool eprosima::fastrtps::rtps::**operator<**(const *InstanceHandle_t* &h1, const *InstanceHandle_t* &h2) noexcept
Comparison operator: checks if a *InstanceHandle_t* is less than another.

Parameters

- **h1** – First *InstanceHandle_t* to compare.
- **h2** – Second *InstanceHandle_t* to compare.

Returns True if the first *InstanceHandle_t* is less than the second.

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *InstanceHandle_t* &iHandle)

Stream operator: print an *InstanceHandle_t*.

Parameters

- **output** – Output stream.
- **iHandle** – *InstanceHandle_t* to print.

Returns Stream operator.

inline std::istream &eprosima::fastrtps::rtps::**operator>>**(std::istream &input, *InstanceHandle_t* &iHandle)

Stream operator: retrieve an *InstanceHandle_t*.

Parameters

- **input** – Input stream.
- **iHandle** – *InstanceHandle_t* that will receive the input as its new value.

Returns Stream operator.

inline void eprosima::fastrtps::rtps::**iHandle2GUID**(*GUID_t* &guid, const *InstanceHandle_t* &ihandle) noexcept

Convert *InstanceHandle_t* to GUID

Parameters

- **guid** – GUID to store the results
- **ihandle** – *InstanceHandle_t* to copy

inline *GUID_t* eprosima::fastrtps::rtps::**iHandle2GUID**(const *InstanceHandle_t* &ihandle) noexcept

Convert GUID to *InstanceHandle_t*

Parameters **ihandle** – *InstanceHandle_t* to store the results

Returns *GUID_t*

Locator

Macro definitions (#define)

}]
Initialize locator with invalid values.

LOCATOR_KIND_INVALID -1
Invalid locator kind.

LOCATOR_ADDRESS_INVALID(a) {std::memset(a, 0x00, 16 * sizeof(octet));}
Set locator IP address to 0.

LOCATOR_PORT_INVALID 0
Invalid locator port.

LOCATOR_KIND_RESERVED 0
Reserved locator kind.

LOCATOR_KIND_UDPv4 1
UDP over IPv4 locator kind.

LOCATOR_KIND_UDPv6 2
UDP over IPv6 locator kind.

LOCATOR_KIND_TCPv4 4
TCP over IPv4 kind.

LOCATOR_KIND_TCPv6 8
TCP over IPv6 locator kind.

LOCATOR_KIND_SHM 16
Shared memory locator kind.

IsAddressDefined

inline bool eprosima::fastrtps::rtps::**IsAddressDefined**(const *Locator_t* &loc)
Auxiliary method to check that IP address is not invalid (0).

Parameters **loc** – Locator which IP address is going to be checked.

Returns true if IP address is defined (not 0).

Returns false otherwise.

IsLocatorValid

inline bool eprosima::fastrtps::rtps::**IsLocatorValid**(const *Locator_t* &loc)
Auxiliary method to check that locator kind is not LOCATOR_KIND_INVALID (-1).

Parameters **loc** – Locator to be checked.

Returns true if the locator kind is not LOCATOR_KIND_INVALID.

Returns false otherwise.

Locator_t

class **Locator_t**
Class *Locator_t*, uniquely identifies a communication channel for a particular transport. For example, an address + port combination in the case of UDP.

Subclassed by *eprosima::fastdds::rtps::LocatorWithMask*

Public Functions

inline **Locator_t**()
Default constructor.

inline **Locator_t**(*Locator_t* &&loc)
Move constructor.

inline **Locator_t**(const *Locator_t* &loc)
Copy constructor.

inline **Locator_t**(uint32_t portin)
Port constructor.

inline **Locator_t**(int32_t kindin, uint32_t portin)
Kind and port constructor.

inline *Locator_t* &**operator=**(const *Locator_t* &loc)
Copy assignment.

inline bool **set_address**(const *Locator_t* &other)
Set the locator IP address using another locator.

Parameters **other** – Locator which IP address is used to set this locator IP address.

Returns always true.

inline *octet* ***get_address**()
Getter for the locator IP address.

Returns IP address as octet pointer.

inline *octet* **get_address**(uint16_t field) const
Getter for a specific field of the locator IP address.

Parameters **field** – IP address element to be accessed.

Returns Octet value for the specific IP address element.

inline void **set_Invalid_Address**()
Automatic setter for setting locator IP address to invalid address (0).

Public Members

int32_t **kind**
Specifies the locator type. Valid values are:

LOCATOR_KIND_UDPv4

LOCATOR_KIND_UDPv6

LOCATOR_KIND_TCPv4

LOCATOR_KIND_TCPv6

LOCATOR_KIND_SHM

uint32_t **port**
Network port.

octet **address**[16]
IP address.

LocatorList

class **LocatorList**
Class *LocatorList*, a Locator vector that doesn't allow duplicates.

Public Functions

inline **LocatorList**()
Constructor.

inline **~LocatorList**()
Destructor.

inline **LocatorList**(const *LocatorList* &list)
Copy constructor.

inline **LocatorList**(*LocatorList* &&list)
Move constructor.

inline *LocatorList* &**operator=**(const *LocatorList* &list)
Copy assignment.

inline *LocatorList* &**operator=**(*LocatorList* &&list)
Move assignment.

inline bool **operator==**(const *LocatorList* &locator_list) const
Equal to operator.

inline *LocatorListIterator* **begin**()
Return an iterator to the beginning.

Returns LocatorListIterator iterator to the first locator.

inline *LocatorListIterator* **end**()
Return an iterator to the end.

Returns LocatorListIterator iterator to the element following the last element.

inline *LocatorListConstIterator* **begin**() const
Return a constant iterator to the beginning.

Returns LocatorListConstIterator iterator to the first locator.

inline *LocatorListConstIterator* **end**() const
Return a constant iterator to the end.

Returns LocatorListConstIterator iterator to the element following the last element.

inline size_t **size**() const
Return the number of locators.

Returns size_t The number of locators in the container.

inline *LocatorList* &**assign**(const *LocatorList* &list)
Replace the contents of the container.

Parameters **list** – New content to be saved into the container.

Returns *LocatorList*& reference to the container with the replaced content.

inline void **clear**()
Erase all locators from the container.

inline void **reserve**(size_t num)
Reserve storage increasing the capacity of the vector.

Parameters **num** – new capacity of the vector, in number of elements.

inline void **resize**(size_t num)
Resize the container to contain num locators. If the current size is greater than num, the container is reduced to its first num locators. If the current size is less than count, additional default-inserted locators are appended.

Parameters **num** – new size of the container.

inline void **push_back**(const Locator &loc)
Add locator to the end if not found within the list.

Parameters **loc** – locator to be appended.

inline void **push_back**(const *LocatorList* &locList)
Add several locators to the end if not already present within the list.

Parameters **locList** – *LocatorList* with the locators to be appended.

inline bool **empty**() const
Check that the container has no locators.

Returns true if the container is empty. False otherwise.

inline void **erase**(const Locator &loc)
Erase the specified locator from the container.

Parameters **loc** – Locator to be removed.

inline bool **isValid**() const
Check that every locator contained in the list is not LOCATOR_KIND_INVALID.

Returns true if all locators are valid. False otherwise.

inline void **swap**(*LocatorList* &locatorList)
exchange the content of the container.

Parameters **locatorList** – container to exchange the contents with.

LocatorListConstIterator

typedef std::vector<Locator>::const_iterator eprosima::fastdds::rtps::**LocatorListConstIterator**
Constant iterator to iterate over a vector of locators.

LocatorListIterator

typedef std::vector<Locator>::iterator eprosima::fastdds::rtps::**LocatorListIterator**
Iterator to iterate over a vector of locators.

LocatorsIterator

struct **LocatorsIterator**
Provides a Locator's iterator interface that can be used by different Locator's containers

Subclassed by *eprosima::fastdds::rtps::Locators*, *eprosima::fastrtps::rtps::LocatorSelector::iterator*

Public Functions

virtual *LocatorsIterator* &**operator++**() = 0
Increment operator.

Returns *LocatorsIterator*& reference to the next *LocatorsIterator*.

virtual bool **operator==**(const *LocatorsIterator* &other) const = 0
Equal to operator.

Parameters **other** – *LocatorsIterator* to compare.

Returns true if equal.

Returns false otherwise.

virtual bool **operator!=**(const *LocatorsIterator* &other) const = 0
Not equal to operator.

Parameters **other** – *LocatorsIterator* to compare.

Returns true if not equal.

Returns false otherwise.

virtual const Locator &**operator***() const = 0
Dereference operator.

Returns const Locator& Reference to the locator pointed by the *LocatorsIterator*.

Locators

class **Locators** : public eprosima::fastdds::rtps::*LocatorsIterator*
Adapter class that provides a *LocatorsIterator* interface from a LocatorListConstIterator

Public Functions

inline **Locators**(const *LocatorListConstIterator* &it)
Constructor.

inline **Locators**(const *Locators* &other)
Copy constructor.

inline virtual *LocatorsIterator* &**operator++**()
Increment operator.

Returns *LocatorsIterator*& reference to the next *LocatorsIterator*.

inline virtual bool **operator==**(const *LocatorsIterator* &other) const
Equal to operator.

Parameters **other** – *LocatorsIterator* to compare.

Returns true if equal.

Returns false otherwise.

inline virtual bool **operator!=**(const *LocatorsIterator* &other) const
Not equal to operator.

Parameters **other** – *LocatorsIterator* to compare.

Returns true if not equal.

Returns false otherwise.

inline virtual const Locator &**operator***() const
Dereference operator.

Returns const Locator& Reference to the locator pointed by the *LocatorsIterator*.

Locator Operators

inline bool eprosima::fastrtps::rtps::**operator<**(const *Locator_t* &loc1, const *Locator_t* &loc2)
Less than operator.

Parameters

- **loc1** – Left hand side locator being compared.
- **loc2** – Right hand side locator being compared.

Returns true if loc1 is less than loc2.

Returns false otherwise.

inline bool eprosima::fastrtps::rtps::**operator==**(const *Locator_t* &loc1, const *Locator_t* &loc2)
Equal to operator.

Parameters

- **loc1** – Left hand side locator being compared.
- **loc2** – Right hand side locator being compared.

Returns true if loc1 is equal to loc2.

Returns false otherwise.

inline bool eprosima::fastrtps::rtps::**operator!=**(const *Locator_t* &loc1, const *Locator_t* &loc2)
Not equal to operator.

Parameters

- **loc1** – Left hand side locator being compared.
- **loc2** – Right hand side locator being compared.

Returns true if loc1 is not equal to loc2.

Returns false otherwise.

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *Locator_t* &loc)
Insertion operator: serialize a locator The serialization format is kind:[address]:port kind must be one of the following:

- UDPv4
- UDPv6
- TCPv4
- TCPv6
- SHM `address` IP address unless `kind` is SHM `port` number

Parameters

- **output** – Output stream where the serialized locator is appended.
- **loc** – Locator to be serialized/inserted.

Returns `std::ostream&` Reference to the output stream with the serialized locator appended.

inline std::istream &eprosima::fastrtps::rtps::**operator>>**(std::istream &input, *Locator_t* &loc)

Extraction operator: deserialize a locator The deserialization format is kind:[address]:port `kind` must be one of the following:

- UDPv4
- UDPv6
- TCPv4
- TCPv6
- SHM `address` must be either a name which can be resolved by DNS or the IP address unless `kind` is SHM `port` number

Parameters

- **input** – Input stream where the locator to be deserialized is located.
- **loc** – Locator where the deserialized locator is saved.

Returns `std::istream&` Reference to the input stream after extracting the locator.

inline std::ostream &eprosima::fastdds::rtps::**operator<<**(std::ostream &output, const *LocatorList* &locList)

Insertion operator: serialize a locator list. The deserialization format is [locator1,locator2,...,locatorN]. Each individual locator within the list must follow the serialization format explained in the locator insertion operator.

Parameters

- **output** – Output stream where the serialized locator list is appended.
- **locList** – Locator list to be serialized/inserted.

Returns `std::ostream&` Reference to the output stream with the serialized locator list appended.

inline std::istream &eprosima::fastdds::rtps::**operator>>**(std::istream &input, *LocatorList* &locList)

Extraction operator: deserialize a list of locators. The serialization format is [locator1,locator2,...,locatorN]. Each individual locator within the list must follow the deserialization format explained in the locator extraction operator.

Parameters

- **input** – Input stream where the locator list to be deserialized is located.
- **locList** – Locator list where the deserialized locators are saved.

Returns `std::istream&` Reference to the input stream after extracting the locator list.

static inline bool `eprosima::fastrtps::rtps::operator==`(const ResourceLimitedVector<*Locator_t*> &lhs, const ResourceLimitedVector<*Locator_t*> &rhs)

Equal to operator to compare two locator lists.

Parameters

- **lhs** – Locator list to be compared.
- **rhs** – Other locator list to be compared.

Returns true if the list are equal.

Returns false otherwise.

LocatorSelectorEntry

struct **LocatorSelectorEntry**

An entry for the *LocatorSelector*.

This class holds the locators of a remote endpoint along with data required for the locator selection algorithm. Can be easily integrated inside other classes, such as *ReaderProxyData* and *WriterProxyData*.

Public Functions

inline **LocatorSelectorEntry**(size_t max_unicast_locators, size_t max_multicast_locators)

Construct a *LocatorSelectorEntry*.

Parameters

- **max_unicast_locators** – Maximum number of unicast locators to hold.
- **max_multicast_locators** – Maximum number of multicast locators to hold.

inline void **enable**(bool should_enable)

Set the enabled value.

Parameters **should_enable** – Whether this entry should be enabled.

inline void **reset**()

Reset the selections.

Public Members

GUID_t **remote_guid**

GUID of the remote entity.

ResourceLimitedVector<*Locator_t*> **unicast**

List of unicast locators to send data to the remote entity.

ResourceLimitedVector<*Locator_t*> **multicast**

List of multicast locators to send data to the remote entity.

EntryState **state**

State of the entry.

bool **enabled**
Indicates whether this entry should be taken into consideration.

bool **transport_should_process**
A temporary value for each transport to help optimizing some use cases.

struct **EntryState**
Holds the selection state of the locators held by a *LocatorSelectorEntry*

Public Functions

inline **EntryState**(size_t max_unicast_locators, size_t max_multicast_locators)
Construct an *EntryState* object.
Parameters
- **max_unicast_locators** – Maximum number of unicast locators to held by parent *LocatorSelectorEntry*.
- **max_multicast_locators** – Maximum number of multicast locators to held by parent *LocatorSelectorEntry*.

Public Members

ResourceLimitedVector<size_t> **unicast**
Unicast locators selection state.

ResourceLimitedVector<size_t> **multicast**
Multicast locators selection state.

LocatorSelector

class **LocatorSelector**
A class used for the efficient selection of locators when sending data to multiple entities.

Algorithm:

- Entries are added/removed with add_entry/remove_entry when matched/unmatched.
- When data is to be sent:
  - A reference to this object is passed to the message group
  - For each submessage:
    - A call to reset is performed
    - A call to enable is performed per desired destination
    - If *state_has_changed()* returns true:
      - the message group is flushed
      - selection_start is called
      - for each transport:
      - transport_starts is called

- transport handles the selection state of each entry
- select may be called

* Submessage is added to the message group

Public Functions

inline **LocatorSelector**(const ResourceLimitedContainerConfig &entries_allocation)
Construct a *LocatorSelector*.

Parameters **entries_allocation** – Allocation configuration regarding the number of remote entities.

inline void **clear**()
Clears all internal data.

inline bool **add_entry**(*LocatorSelectorEntry* *entry)
Add an entry to this selector.

Parameters **entry** – Pointer to the *LocatorSelectorEntry* to add.

inline bool **remove_entry**(const *GUID_t* &guid)
Remove an entry from this selector.

Parameters **guid** – Identifier of the entry to be removed.

inline void **reset**(bool enable_all)
Reset the enabling state of the selector.

Parameters **enable_all** – Indicates whether entries should be initially enabled.

inline void **enable**(const *GUID_t* &guid)
Enable an entry given its GUID.

Parameters **guid** – GUID of the entry to enable.

inline bool **state_has_changed**() const
Check if enabling state has changed.

Returns true if the enabling state has changed, false otherwise.

inline void **selection_start**()
Reset the selection state of the selector.

inline ResourceLimitedVector<*LocatorSelectorEntry**> &**transport_starts**()
Called when the selection algorithm starts for a specific transport.

Will set the temporary transport_should_process flag for all enabled entries.

Returns a reference to the entries collection.

inline void **select**(size_t index)
Marks an entry as selected.

Parameters **index** – The index of the entry to mark as selected.

inline size_t **selected_size**() const
Count the number of selected locators.

Returns the number of selected locators.

inline bool **is_selected**(const *Locator_t* locator) const
Check if a locator is present in the selections of this object.

Parameters `locator` – The locator to be checked.

Returns True if the locator has been selected, false otherwise.

template<class **UnaryPredicate**>
inline void **for_each**(*UnaryPredicate* action) const
Performs an action on each selected locator.

Parameters `action` – Unary function that accepts a locator as argument. The function shall not modify its argument. This can either be a function pointer or a function object.

class **iterator** : public eprosima::fastdds::rtps::*LocatorsIterator*

Public Functions

inline virtual *iterator* &**operator++**()
Increment operator.

Returns LocatorsIterator& reference to the next LocatorsIterator.

inline virtual bool **operator==**(const LocatorsIterator &other) const
Equal to operator.

Parameters `other` – LocatorsIterator to compare.

Returns true if equal.

Returns false otherwise.

inline virtual bool **operator!=**(const LocatorsIterator &other) const
Not equal to operator.

Parameters `other` – LocatorsIterator to compare.

Returns true if not equal.

Returns false otherwise.

inline virtual reference **operator***() const
Dereference operator.

Returns const Locator& Reference to the locator pointed by the LocatorsIterator.

struct **IteratorIndex**

LocatorWithMask

class **LocatorWithMask** : public eprosima::fastrtps::rtps::*Locator_t*
A Locator with a mask that defines the number of significant bits of its address.

Public Functions

uint8_t **mask**() const
Get the number of significant bits on the address of this locator.

Returns number of significant bits on the address of this locator.

void **mask**(uint8_t mask)
Set the number of significant bits on the address of this locator.

Parameters **mask** – number of significant bits on the address of this locator.

bool **matches**(const Locator &loc) const
Check whether the given locator is from the same network as this locator.

Parameters **loc** – locator to check if belonging to the same network as this locator.

Returns true if the two locators are from the same network, false otherwise.

LocatorWithMask &**operator=**(const Locator &loc)
Copy assignment.

MatchingInfo

MatchingInfo

class **MatchingInfo**
Class *MatchingInfo* contains information about the matching between two endpoints.

Public Functions

inline **MatchingInfo**()
Default constructor.

inline **MatchingInfo**(*MatchingStatus* stat, const *GUID_t* &guid)

Parameters

- **stat** – Status
- **guid** – GUID

Public Members

MatchingStatus **status**
Status.

GUID_t **remoteEndpointGuid**
Remote endpoint GUID.

MatchingStatus

enum eprosima::fastrtps::rtps::**MatchingStatus**
, indicates whether the matched publication/subscription method of the PublisherListener or SubscriberListener has been called for a matching or a removal of a remote endpoint.

Values:

enumerator **MATCHED_MATCHING**
MATCHED_MATCHING, new publisher/subscriber found.

enumerator **REMOVED_MATCHING**
REMOVED_MATCHING, publisher/subscriber removed.

PortParameters

class **PortParameters**
Class *PortParameters*, to define the port parameters and gains related with the RTPS protocol.

Public Functions

inline uint32_t **getMulticastPort**(uint32_t domainId) const
Get a multicast port based on the domain ID.

Parameters **domainId** – Domain ID.

Returns Multicast port

inline uint32_t **getUnicastPort**(uint32_t domainId, uint32_t RTPSParticipantID) const
Get a unicast port based on the domain ID and the participant ID.

Parameters

- **domainId** – Domain ID.
- **RTPSParticipantID** – Participant ID.

Returns Unicast port

Public Members

uint16_t **portBase**
PortBase, default value 7400.

uint16_t **domainIDGain**
DomainID gain, default value 250.

uint16_t **participantIDGain**
ParticipantID gain, default value 2.

uint16_t **offsetd0**
Offset d0, default value 0.

uint16_t **offsetd1**
: Offset d1, default value 10.

uint16_t **offsetd2**
: Offset d2, default value 1.

uint16_t **offsetd3**
: Offset d3, default value 11.

Property

Property

class **Property**

PropertyHelper

class **PropertyHelper**

PropertySeq

typedef std::vector<*Property*> `eprosima::fastrtps::rtps::`**PropertySeq**

RemoteLocators

RemoteLocators Operators

inline std::ostream &`eprosima::fastrtps::rtps::`**operator<<**(std::ostream &output, const *RemoteLocatorList* &remote_locators)

RemoteLocatorList

struct **RemoteLocatorList**
: Holds information about the locators of a remote entity.

Public Functions

inline **RemoteLocatorList**()
Default constructor of *RemoteLocatorList* for deserialize.

inline **RemoteLocatorList**(size_t max_unicast_locators, size_t max_multicast_locators)
Construct a *RemoteLocatorList*.

Parameters

- **max_unicast_locators** – Maximum number of unicast locators to hold.
- **max_multicast_locators** – Maximum number of multicast locators to hold.

inline **RemoteLocatorList**(const *RemoteLocatorList* &other)
Copy-construct a *RemoteLocatorList*.

Parameters **other** – *RemoteLocatorList* to copy data from.

inline *RemoteLocatorList* &**operator=**(const *RemoteLocatorList* &other)
Assign locator values from other *RemoteLocatorList*.

Remark

Using the assignment operator is different from copy-constructing as in the first case the configuration with the maximum number of locators is not copied. This means that, for two lists with different maximum number of locators, the expression `(a = b) == b` may not be true.

Parameters **other** – *RemoteLocatorList* to copy data from.

inline void **add_unicast_locator**(const *Locator_t* &locator)
Adds a locator to the unicast list.

If the locator already exists in the unicast list, or the maximum number of unicast locators has been reached, the new locator is silently discarded.

Parameters **locator** – Unicast locator to be added.

inline void **add_multicast_locator**(const *Locator_t* &locator)
Adds a locator to the multicast list.

If the locator already exists in the multicast list, or the maximum number of multicast locators has been reached, the new locator is silently discarded.

Parameters **locator** – Multicast locator to be added.

Public Members

ResourceLimitedVector<*Locator_t*> **unicast**
List of unicast locators.

ResourceLimitedVector<*Locator_t*> **multicast**
List of multicast locators.

SampleIdentity

class **SampleIdentity**
This class is used to specify a sample.

Public Functions

inline **SampleIdentity**()
Default constructor.

Constructs an unknown *SampleIdentity*.

inline **SampleIdentity**(const *SampleIdentity* &sample_id)
Copy constructor.

inline **SampleIdentity**(*SampleIdentity* &&sample_id)
Move constructor.

inline *SampleIdentity* &**operator=**(const *SampleIdentity* &sample_id)
Assignment operator.

inline *SampleIdentity* &**operator=**(*SampleIdentity* &&sample_id)
Move constructor.

inline bool **operator<**(const *SampleIdentity* &sample) const
To allow using *SampleIdentity* as map key.

Parameters **sample** –

Returns

SequenceNumber

c_SequenceNumber_Unknown

const *SequenceNumber_t* eprosima::fastrtps::rtps::**c_SequenceNumber_Unknown** = {-1, 0}

SequenceNumber_t Operators

inline bool eprosima::fastrtps::rtps::**operator==**(const *SequenceNumber_t* &sn1, const *SequenceNumber_t* &sn2) noexcept
Compares two *SequenceNumber_t*.

Parameters

- **sn1** – First *SequenceNumber_t* to compare
- **sn2** – Second *SequenceNumber_t* to compare

Returns True if equal

inline bool eprosima::fastrtps::rtps::**operator!=**(const *SequenceNumber_t* &sn1, const *SequenceNumber_t* &sn2) noexcept
Compares two *SequenceNumber_t*.

Parameters

- **sn1** – First *SequenceNumber_t* to compare
- **sn2** – Second *SequenceNumber_t* to compare

Returns True if not equal

inline bool eprosima::fastrtps::rtps::**operator>**(const *SequenceNumber_t* &seq1, const *SequenceNumber_t* &seq2) noexcept

Checks if a *SequenceNumber_t* is greater than other.

Parameters

- **seq1** – First *SequenceNumber_t* to compare
- **seq2** – Second *SequenceNumber_t* to compare

Returns True if the first *SequenceNumber_t* is greater than the second

inline bool eprosima::fastrtps::rtps::**operator<**(const *SequenceNumber_t* &seq1, const *SequenceNumber_t* &seq2) noexcept

Checks if a *SequenceNumber_t* is less than other.

Parameters

- **seq1** – First *SequenceNumber_t* to compare
- **seq2** – Second *SequenceNumber_t* to compare

Returns True if the first *SequenceNumber_t* is less than the second

inline bool eprosima::fastrtps::rtps::**operator>=**(const *SequenceNumber_t* &seq1, const *SequenceNumber_t* &seq2) noexcept

Checks if a *SequenceNumber_t* is greater or equal than other.

Parameters

- **seq1** – First *SequenceNumber_t* to compare
- **seq2** – Second *SequenceNumber_t* to compare

Returns True if the first *SequenceNumber_t* is greater or equal than the second

inline bool eprosima::fastrtps::rtps::**operator<=**(const *SequenceNumber_t* &seq1, const *SequenceNumber_t* &seq2) noexcept

Checks if a *SequenceNumber_t* is less or equal than other.

Parameters

- **seq1** – First *SequenceNumber_t* to compare
- **seq2** – Second *SequenceNumber_t* to compare

Returns True if the first *SequenceNumber_t* is less or equal than the second

inline *SequenceNumber_t* eprosima::fastrtps::rtps::**operator-**(const *SequenceNumber_t* &seq, const uint32_t inc) noexcept

Subtract one uint32_t from a *SequenceNumber_t*

Parameters

- **seq** – Base *SequenceNumber_t*
- **inc** – uint32_t to subtract

Returns Result of the subtraction

inline *SequenceNumber_t* eprosima::fastrtps::rtps::**operator+**(const *SequenceNumber_t* &seq, const uint32_t inc) noexcept

Add one uint32_t to a *SequenceNumber_t*

Parameters

- **seq** – **[in]** Base sequence number
- **inc** – value to add to the base

Returns Result of the addition

inline *SequenceNumber_t* eprosima::fastrtps::rtps::**operator-**(const *SequenceNumber_t* &minuend, const *SequenceNumber_t* &subtrahend) noexcept

Subtract one *SequenceNumber_t* to another

Parameters

- **minuend** – Minuend. Has to be greater than or equal to subtrahend.
- **subtrahend** – Subtrahend.

Returns Result of the subtraction

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *SequenceNumber_t* &seqNum)

Parameters

- **output** –
- **seqNum** –

Returns

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const std::vector<*SequenceNumber_t*> &seqNumSet)

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *SequenceNumberSet_t* &sns)

Prints a sequence Number set

Parameters

- **output** – Output Stream
- **sns** – SequenceNumber set

Returns OStream.

SequenceNumber_t

struct **SequenceNumber_t**

Structure *SequenceNumber_t*, different for each change in the same writer.

Public Functions

inline **SequenceNumber_t**() noexcept
Default constructor.

inline **SequenceNumber_t**(int32_t hi, uint32_t lo) noexcept

Parameters

- **hi** –
- **lo** –

inline explicit **SequenceNumber_t**(uint64_t u) noexcept

Parameters **u** –

inline uint64_t **to64long**() const noexcept
Convert the number to 64 bit.

Returns 64 bit representation of the SequenceNumber

inline *SequenceNumber_t* &**operator++**() noexcept
Increase SequenceNumber in 1.

inline *SequenceNumber_t* &**operator+=**(int inc) noexcept
Increase SequenceNumber.

Parameters **inc** – Number to add to the SequenceNumber

SequenceNumberDiff

struct **SequenceNumberDiff**

SequenceNumberHash

struct **SequenceNumberHash**
Defines the STL hash function for type *SequenceNumber_t*.

SequenceNumberSet_t

using eprosima::fastrtps::rtps::**SequenceNumberSet_t** = BitmapRange<*SequenceNumber_t*, *SequenceNumberDiff*, 256>
Structure SequenceNumberSet_t, contains a group of sequencenumbers.

sort_seqNum

inline bool eprosima::fastrtps::rtps::**sort_seqNum**(const *SequenceNumber_t* &s1, const *SequenceNumber_t* &s2) noexcept

Sorts two instances of *SequenceNumber_t*

Parameters

- **s1** – First *SequenceNumber_t* to compare
- **s2** – First *SequenceNumber_t* to compare

Returns True if s1 is less than s2

SerializedPayload

Macro definitions (#define)

CDR_BE 0x0000

CDR_LE 0x0001

PL_CDR_BE 0x0002

PL_CDR_LE 0x0003

SerializedPayload_t

struct **SerializedPayload_t**

Structure *SerializedPayload_t*.

Public Functions

inline **SerializedPayload_t**()

Default constructor.

inline explicit **SerializedPayload_t**(uint32_t len)

Parameters **len** – Maximum size of the payload

inline bool **copy**(const *SerializedPayload_t* *serData, bool with_limit = true)

Copy another structure (including allocating new space for the data.)

Parameters

- **serData** – **[in]** Pointer to the structure to copy
- **with_limit** – if true, the function will fail when providing a payload too big

Returns True if correct

inline bool **reserve_fragmented**(*SerializedPayload_t* *serData)

Allocate new space for fragmented data.

Parameters **serData** – **[in]** Pointer to the structure to copy

Returns True if correct

inline void **empty**()
Empty the payload.

Public Members

uint16_t **encapsulation**
Encapsulation of the data as suggested in the RTPS 2.1 specification chapter 10.

uint32_t **length**
Actual length of the data.

octet ***data**
Pointer to the data.

uint32_t **max_size**
Maximum size of the payload.

uint32_t **pos**
Position when reading.

Public Static Attributes

static constexpr size_t **representation_header_size** = 4u
Size in bytes of the representation header as specified in the RTPS 2.3 specification chapter 10.

Time_t

Const values

const *Time_t* `eprosima::fastrtps::`**c_TimeInfinite** = {*Time_t*::INFINITE_SECONDS, *Time_t*::INFINITE_NANOSECONDS}
Time_t (Duration_t) representing an infinite time. DONT USE IT IN CONSTRUCTORS.

const *Time_t* `eprosima::fastrtps::`**c_TimeZero** = {0, 0}
Time_t (Duration_t) representing a zero time. DONT USE IT IN CONSTRUCTORS.

const *Time_t* `eprosima::fastrtps::`**c_TimeInvalid** = {-1, *Time_t*::INFINITE_NANOSECONDS}
Time_t (Duration_t) representing an invalid time. DONT USE IT IN CONSTRUCTORS.

Macro definitions (#define)

TIME_T_INFINITE_SECONDS (eprosima::fastrtps::Time_t::INFINITE_SECONDS)

TIME_T_INFINITE_NANOSECONDS (eprosima::fastrtps::Time_t::INFINITE_NANOSECONDS)

eprosima::fastrtps::Duration_t

using `eprosima::fastrtps::`**`Duration_t`** = *Time_t*

eprosima::fastrtps::Time_t

struct **Time_t**
Structure *Time_t*, used to describe times.

Public Functions

Time_t()
Default constructor. Sets values to zero.

Time_t(int32_t sec, uint32_t nsec)

Parameters

- **sec** – Seconds
- **nsec** – Nanoseconds

Time_t(long double sec)

Parameters **sec** – Seconds. The fractional part is converted to nanoseconds.

int64_t **to_ns**() const
Returns stored time as nanoseconds (including seconds)

Public Static Functions

static void **now**(*Time_t* &ret)
Fills a *Time_t* struct with a representation of the current time.

Parameters **ret** – Reference to the structure to be filled in.

Time_t Operators

static inline bool eprosima::fastrtps::rtps::**operator==**(const *Time_t* &t1, const *Time_t* &t2)
Comparison assignment

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if equal

static inline bool eprosima::fastrtps::rtps::**operator!=**(const *Time_t* &t1, const *Time_t* &t2)
Comparison assignment

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if not equal

static inline bool eprosima::fastrtps::rtps::**operator<**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is less than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is less than the second

static inline bool eprosima::fastrtps::rtps::**operator>**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is greater than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is greater than the second

static inline bool eprosima::fastrtps::rtps::**operator<=**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is less or equal than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is less or equal than the second

static inline bool eprosima::fastrtps::rtps::**operator>=**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is greater or equal than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is greater or equal than the second

inline std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &output, const *Time_t* &t)

static inline *Time_t* eprosima::fastrtps::rtps::**operator+**(const *Time_t* &ta, const *Time_t* &tb)
Adds two *Time_t*.

Parameters

- **ta** – First *Time_t* to add
- **tb** – Second *Time_t* to add

Returns A new *Time_t* with the result.

static inline *Time_t* eprosima::fastrtps::rtps::**operator-**(const *Time_t* &ta, const *Time_t* &tb)
Subtracts two *Time_t*.

Parameters

- **ta** – First *Time_t* to subtract
- **tb** – Second *Time_t* to subtract

Returns A new *Time_t* with the result.

static inline bool eprosima::fastrtps::**operator==**(const *Time_t* &t1, const *Time_t* &t2)
Comparison assignment

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if equal

static inline bool eprosima::fastrtps::**operator!=**(const *Time_t* &t1, const *Time_t* &t2)
Comparison assignment

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if not equal

static inline bool eprosima::fastrtps::**operator<**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is less than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is less than the second

static inline bool eprosima::fastrtps::**operator>**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is greater than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is greater than the second

static inline bool eprosima::fastrtps::**operator<=**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is less or equal than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is less or equal than the second

static inline bool eprosima::fastrtps::**operator>=**(const *Time_t* &t1, const *Time_t* &t2)
Checks if a *Time_t* is greater or equal than other.

Parameters

- **t1** – First *Time_t* to compare
- **t2** – Second *Time_t* to compare

Returns True if the first *Time_t* is greater or equal than the second

inline std::ostream &eprosima::fastrtps::**operator<<**(std::ostream &output, const *Time_t* &t)

static inline *Time_t* eprosima::fastrtps::**operator+**(const *Time_t* &ta, const *Time_t* &tb)
Adds two *Time_t*.

Parameters

- **ta** – First *Time_t* to add
- **tb** – Second *Time_t* to add

Returns A new *Time_t* with the result.

static inline *Time_t* eprosima::fastrtps::**operator-**(const *Time_t* &ta, const *Time_t* &tb)
Subtracts two *Time_t*.

Parameters

- **ta** – First *Time_t* to subtract
- **tb** – Second *Time_t* to subtract

Returns A new *Time_t* with the result.

Time_t

class **Time_t**
Structure *Time_t*, used to describe times at RTPS protocol.

Public Functions

Time_t() = default
Default constructor. Sets values to zero.

Time_t(int32_t sec, uint32_t frac)

Parameters

- **sec** – Seconds
- **frac** – Fraction of second

Time_t(long double sec)

Parameters **sec** – Seconds. The fractional part is converted to nanoseconds.

Time_t(const eprosima::fastrtps::*Time_t* &time)

Parameters **time** – *fastrtps::Time_t*, aka. Duration_t.

int64_t **to_ns**() const
Returns stored time as nanoseconds (including seconds)

void **from_ns**(int64_t nanosecs)

Parameters **nanosecs** – Stores given time as nanoseconds (including seconds)

int32_t **seconds**() const
Retrieve the seconds field.

int32_t &**seconds**()
Retrieve the seconds field by ref.

void **seconds**(int32_t sec)
Sets seconds field.

uint32_t **nanosec**() const
Retrieve the nanosec field.

void **nanosec**(uint32_t nanos)
Sets nanoseconds field and updates the fraction.

uint32_t **fraction**() const
Retrieve the fraction field.

uint32_t &**fraction**()
Retrieve the fraction field by ref.

void **fraction**(uint32_t frac)
Sets fraction field and updates the nanoseconds.

Public Static Functions

static void **now**(*Time_t* &ret)
Fills a *Time_t* struct with a representation of the current time.

Parameters `ret` – Reference to the structure to be filled in.

Token

AuthenticatedPeerCredentialToken

typedef *Token* `eprosima::fastrtps::rtps::`**AuthenticatedPeerCredentialToken**

DataHolder

class **DataHolder**

DataHolderHelper

class **DataHolderHelper**

DataHolderSeq

typedef std::vector<*DataHolder*> `eprosima::fastrtps::rtps::`**DataHolderSeq**

IdentityStatusToken

typedef *Token* `eprosima::fastrtps::rtps::`**IdentityStatusToken**

IdentityToken

typedef *Token* `eprosima::fastrtps::rtps::`**IdentityToken**

PermissionsCredentialToken

typedef *Token* `eprosima::fastrtps::rtps::`**PermissionsCredentialToken**

PermissionsToken

typedef *Token* eprosima::fastrtps::rtps::**PermissionsToken**

Token

typedef *DataHolder* eprosima::fastrtps::rtps::**Token**

Types

BuiltinEndpointSet_t

using eprosima::fastrtps::rtps::**BuiltinEndpointSet_t** = uint32_t

Const values

const *ProtocolVersion_t* eprosima::fastrtps::rtps::**c_ProtocolVersion_2_0** = {2, 0}

const *ProtocolVersion_t* eprosima::fastrtps::rtps::**c_ProtocolVersion_2_1** = {2, 1}

const *ProtocolVersion_t* eprosima::fastrtps::rtps::**c_ProtocolVersion_2_2** = {2, 2}

const *ProtocolVersion_t* eprosima::fastrtps::rtps::**c_ProtocolVersion_2_3** = {2, 3}

const *ProtocolVersion_t* eprosima::fastrtps::rtps::**c_ProtocolVersion**

const *VendorId_t* eprosima::fastdds::rtps::**c_VendorId_Unknown** = {0x00, 0x00}

const *VendorId_t* eprosima::fastdds::rtps::**c_VendorId_eProsima** = {0x01, 0x0F}

Count_t

using eprosima::fastrtps::rtps::**Count_t** = uint32_t

Macro definitions (#define)

BIT0 0x01u

BIT1 0x02u

BIT2 0x04u

BIT3 0x08u

BIT4 0x10u

BIT5 0x20u

BIT6 0x40u

BIT7 0x80u

BIT(i) (1U << static_cast<unsigned>(i))

DurabilityKind_t

typedef enum eprosima::fastrtps::rtps::*DurabilityKind_t* `eprosima::fastrtps::rtps::`**DurabilityKind_t**
Durability kind

Endianness_t

enum `eprosima::fastrtps::rtps::`**Endianness_t**
This enumeration represents endianness types.

Values:

enumerator **BIGEND**
Big endianness.

enumerator **LITTLEEND**
Little endianness.

EndpointKind_t

typedef enum eprosima::fastrtps::rtps::*EndpointKind_t* `eprosima::fastrtps::rtps::`**EndpointKind_t**
Endpoint kind

octet

using `eprosima::fastrtps::rtps::`**octet** = unsigned char

ProtocolVersion_t

struct **ProtocolVersion_t**
Structure *ProtocolVersion_t*, contains the protocol version.

inline std::ostream &`eprosima::fastrtps::rtps::`**operator<<**(std::ostream &output, const *ProtocolVersion_t* &pv)
Prints a ProtocolVersion

Parameters

- **output** – Output Stream
- **pv** – ProtocolVersion

Returns OStream.

ReliabilityKind_t

typedef enum eprosima::fastrtps::rtps::*ReliabilityKind_t* **eprosima::fastrtps::rtps::ReliabilityKind_t**
Reliability enum used for internal purposes

SubmessageFlag

using **eprosima::fastrtps::rtps::SubmessageFlag** = unsigned char

TopicKind_t

typedef enum eprosima::fastrtps::rtps::*TopicKind_t* **eprosima::fastrtps::rtps::TopicKind_t**
Topic kind.

VendorId_t

using **eprosima::fastdds::rtps::VendorId_t** = std::array<uint8_t, 2>
Structure VendorId_t, specifying the vendor Id of the implementation.

WriteParams

class **WriteParams**
This class contains additional information of a CacheChange.

Public Functions

inline *WriteParams* &**sample_identity**(const *SampleIdentity* &sample_id)
Set the value of the sample_identity member.

Parameters **sample_id** – New value for the sample_identity member.

Returns Reference to the modified object in order to allow daisy chaining.

inline *WriteParams* &**sample_identity**(*SampleIdentity* &&sample_id)
Set the value of the sample_identity member.

Parameters **sample_id** – New value for the sample_identity member.

Returns Reference to the modified object in order to allow daisy chaining.

inline const *SampleIdentity* &**sample_identity**() const
Get the value of the sample_identity member.

Returns Constant reference to the sample_identity member.

inline *SampleIdentity* &**sample_identity**()
Set the value of the sample_identity member.

Returns Reference to the sample_identity member.

inline *WriteParams* &**related_sample_identity**(const *SampleIdentity* &sample_id)
Set the value of the related_sample_identity member of this class.

Parameters **sample_id** – New value for the related_sample_identity member.

Returns Reference to the modified object in order to allow daisy chaining.

inline *WriteParams* &**related_sample_identity**(*SampleIdentity* &&sample_id)
Set the related_sample_identity member of this class.

Parameters **sample_id** – New value for the related_sample_identity member.

Returns Reference to the modified object in order to allow daisy chaining.

inline const *SampleIdentity* &**related_sample_identity**() const
Get the value of the related_sample_identity member.

Returns Constant reference to the related_sample_identity member.

inline *SampleIdentity* &**related_sample_identity**()
Set the value of the related_sample_identity member.

Returns Reference to the related_sample_identity member.

inline *Time_t* **source_timestamp**() const
Get the value of the source_timestamp member.

Returns Current value of the source_timestamp member.

inline *Time_t* &**source_timestamp**()
Set the value of the source_timestamp member.

Returns Reference to the source_timestamp member.

inline *WriteParams* &**source_timestamp**(const *Time_t* &timestamp)
Set the source_timestamp member of this class.

Parameters **timestamp** – New value for the source_timestamp member.

Returns Reference to the modified object in order to allow daisy chaining.

inline *WriteParams* &**source_timestamp**(*Time_t* &&timestamp)
Set the source_timestamp member of this class.

Parameters **timestamp** – New value for the source_timestamp member.

Returns Reference to the modified object in order to allow daisy chaining.

Public Static Functions

static inline *WriteParams* **write_params_default**() noexcept
Default value for methods receiving a *WriteParams*.

Will contain the following values on its members:

- sample_identity: Invalid *SampleIdentity*
- related_sample_identity: Invalid *SampleIdentity*
- source_timestamp: Invalid *Time_t*

Note: This should not return a reference to the static value if this value is meant to be read and written from different threads.

Endpoint

class **Endpoint**

Class *Endpoint*, all entities of the RTPS network derive from this class. Although the *RTPSParticipant* is also defined as an endpoint in the RTPS specification, in this implementation the *RTPSParticipant* class **does not** inherit from the endpoint class. Each *Endpoint* object owns a pointer to the *RTPSParticipant* it belongs to.

Subclassed by *eprosima::fastrtps::rtps::RTPSReader*, *eprosima::fastrtps::rtps::RTPSWriter*

Public Functions

inline const *GUID_t* &**getGuid**() const

Get associated GUID

Returns Associated GUID

inline RecursiveTimedMutex &**getMutex**()

Get mutex

Returns Associated Mutex

inline *EndpointAttributes* &**getAttributes**()

Get associated attributes

Returns *Endpoint* attributes

Exceptions

Exception

class **Exception** : public exception

This abstract class is used to create exceptions.

Subclassed by eprosima::fastrtps::rtps::security::SecurityException

Public Functions

virtual **~Exception**()

Default destructor.

const int32_t &**minor**() const

This function returns the number associated with the system exception.

Returns The number associated with the system exception.

void **minor**(const int32_t &minor)

This function sets the number that will be associated with the system exception.

Parameters **minor** – The number that will be associated with the system exception.

virtual void **raise**() const = 0

This function throws the object as exception.

virtual const char ***what**() const

This function returns the error message.

Returns The error message.

Flow control

FlowControllerDescriptor

struct **FlowControllerDescriptor**
Configuration values for creating flow controllers.

This descriptor is used to define the configuration applied in the creation of a flow controller.

Since 2.4.0

Public Members

const char ***name** = nullptr
Name of the flow controller.

FlowControllerSchedulerPolicy **scheduler** = *FlowControllerSchedulerPolicy*::*FIFO*
Scheduler policy used by the flow controller.

Default value: FlowControllerScheduler::FIFO_SCHEDULER

int32_t **max_bytes_per_period** = 0
Maximum number of bytes to be sent to network per period.

Range of bytes: [1, 2147483647]; 0 value means no limit. Default value: 0

uint64_t **period_ms** = 100
Period time in milliseconds.

Period of time on which the flow controller is allowed to send max_bytes_per_period. Default value: 100ms.

ThreadSettings **sender_thread**
Thread settings for the sender thread.

const char *const `eprosima::fastdds::rtps::`**FASTDDS_FLOW_CONTROLLER_DEFAULT**
Name of the default flow controller.

const char *const `eprosima::fastdds::rtps::`**FASTDDS_STATISTICS_FLOW_CONTROLLER_DEFAULT**
Name of the default flow controller for statistics writers.

FlowControllerSchedulerPolicy

enum class `eprosima::fastdds::rtps::`**FlowControllerSchedulerPolicy** : int32_t
Supported scheduler policy by a flow controller.

A flow controller's scheduler policy takes the decision of which samples are the next ones to be sent to the network. Fast DDS flow controller supports several scheduler policies listed in this enumeration.

Values:

enumerator **FIFO**
FIFO scheduler policy: first written sample by user, first sample scheduled to be sent to network.

enumerator **ROUND_ROBIN**
Round Robin scheduler policy: schedules one sample of each DataWriter in circular order.

enumerator **HIGH_PRIORITY**
High priority scheduler policy: samples with highest priority are scheduled first to be sent to network.

enumerator **PRIORITY_WITH_RESERVATION**
Priority with reservation scheduler policy: guarantee each DataWriter's minimum reservation of throughput. Samples not fitting the reservation are scheduled by priority.

History

History

class **History**
Class *History*, container of the different CacheChanges and the methods to access them.

Subclassed by *eprosima::fastrtps::rtps::ReaderHistory*, *eprosima::fastrtps::rtps::WriterHistory*

Public Functions

inline bool **reserve_Cache**(*CacheChange_t* **change, const std::function<uint32_t()> &calculateSizeFunc)
Reserve a *CacheChange_t* from the CacheChange pool.

> **Warning:** This method has been deprecated and will be removed on v3.0.0

Parameters

- **change** – **[out]** Pointer to pointer to the *CacheChange_t* to reserve
- **calculateSizeFunc** – **[in]** Function to calculate the size of the payload.

Returns True if reserved

inline bool **reserve_Cache**(*CacheChange_t* **change, uint32_t dataSize)
Reserve a *CacheChange_t* from the CacheChange pool.

> **Warning:** This method has been deprecated and will be removed on v3.0.0

Parameters

- **change** – **[out]** Pointer to pointer to the *CacheChange_t* to reserve
- **dataSize** – **[in]** Required size for the payload.

Returns True if reserved

inline void **release_Cache**(*CacheChange_t* *ch)
release a previously reserved *CacheChange_t*.

> **Warning:** This method has been deprecated and will be removed on v3.0.0

Parameters **ch** – Pointer to the *CacheChange_t*.

inline bool **isFull**()
Check if the history is full

Returns true if the *History* is full.

inline size_t **getHistorySize**()
Get the *History* size.

Returns Size of the history.

const_iterator **find_change_nts**(*CacheChange_t* *ch)
Find a specific change in the history using the matches_change method criteria. No Thread Safe

Parameters **ch** – Pointer to the *CacheChange_t* to search for.

Returns an iterator if a suitable change is found

virtual iterator **remove_change_nts**(const_iterator removal, bool release = true)
Remove a specific change from the history. No Thread Safe

Parameters

- **removal** – iterator to the *CacheChange_t* to remove.
- **release** – defaults to true and hints if the *CacheChange_t* should return to the pool

Returns iterator to the next *CacheChange_t* or end iterator.

virtual iterator **remove_change_nts**(const_iterator removal, const
std::chrono::time_point<std::chrono::steady_clock>
&max_blocking_time, bool release = true)
Remove a specific change from the history. No Thread Safe

Parameters

- **removal** – iterator to the *CacheChange_t* to remove.
- **max_blocking_time** – **[in]** Maximum time this method has to complete the task.
- **release** – defaults to true and hints if the *CacheChange_t* should return to the pool

Returns iterator to the next *CacheChange_t* or end iterator.

bool **remove_all_changes**()
Remove all changes from the *History*

Returns True if everything was correctly removed.

bool **remove_change**(*CacheChange_t* *ch)
Remove a specific change from the history.

Parameters **ch** – Pointer to the *CacheChange_t*.

Returns True if removed.

bool **remove_change**(*CacheChange_t* *ch, const std::chrono::time_point<std::chrono::steady_clock> &max_blocking_time)

Remove a specific change from the history.

Parameters

- **ch** – Pointer to the *CacheChange_t*.
- **max_blocking_time** – **[in]** Maximum time this method has to complete the task.

Returns True if removed.

inline const_iterator **find_change**(*CacheChange_t* *ch)

Find a specific change in the history using the matches_change method criteria.

Parameters **ch** – Pointer to the *CacheChange_t* to search for.

Returns an iterator if a suitable change is found

virtual bool **matches_change**(const *CacheChange_t* *ch_inner, *CacheChange_t* *ch_outer)

Verifies if an element of the changes collection matches a given change Derived classes have more info on how to identify univocally a change and should override.

Parameters

- **ch_inner** – element of the collection to compare with the given change
- **ch_outer** – Pointer to the *CacheChange_t* to identify.

Returns true if the iterator identifies this change.

inline iterator **remove_change**(const_iterator removal, bool release = true)

Remove a specific change from the history.

Parameters

- **removal** – iterator to the *CacheChange_t* to remove.
- **release** – defaults to true and hints if the *CacheChange_t* should return to the pool

Returns iterator to the next *CacheChange_t* or end iterator.

inline iterator **changesBegin**()

Get the beginning of the changes history iterator.

Returns Iterator to the beginning of the vector.

inline iterator **changesEnd**()

Get the end of the changes history iterator.

Returns Iterator to the end of the vector.

bool **get_min_change**(*CacheChange_t* **min_change)

Get the minimum *CacheChange_t*.

Parameters **min_change** – Pointer to pointer to the minimum change.

Returns True if correct.

bool **get_max_change**(*CacheChange_t* **max_change)

Get the maximum *CacheChange_t*.

Parameters **max_change** – Pointer to pointer to the maximum change.

Returns True if correct.

inline uint32_t **getTypeMaxSerialized**()

Get the maximum serialized payload size

Returns Maximum serialized payload size

inline RecursiveTimedMutex ***getMutex**() const
Get the mutex.

Returns Mutex

bool **get_earliest_change**(*CacheChange_t* **change)
A method to get the change with the earliest timestamp.

Parameters **change** – Pointer to pointer to earliest change

Returns True on success

Public Members

HistoryAttributes **m_att**
Attributes of the *History*.

IChangePool

class **IChangePool**
An interface for classes responsible of cache changes allocation management.

Public Functions

virtual bool **reserve_cache**(*CacheChange_t* *&cache_change) = 0
Get a new cache change from the pool.

Parameters **cache_change** – **[out]** Pointer to the new cache change.

Returns whether the operation succeeded or not

Pre `cache_change` is `nullptr`

Post

- `cache_change` is not nullptr
- `*cache_change` equals `CacheChange_t()` except for the contents of `serializedPayload`

virtual bool **release_cache**(*CacheChange_t* *cache_change) = 0
Return a cache change to the pool.

Parameters **cache_change** – **[in]** Pointer to the cache change to release.

Returns whether the operation succeeded or not

Pre

- `cache_change` is not `nullptr`
- `cache_change` points to a cache change obtained from a call to `this->reserve_cache`

IPayloadPool

class **IPayloadPool**
An interface for classes responsible of serialized payload management.

Public Functions

virtual bool **get_payload**(uint32_t size, *CacheChange_t* &cache_change) = 0
Get a serialized payload for a new sample.

This method will usually be called in one of the following situations:

- When a writer creates a new cache change
- When a reader receives the first fragment of a cache change

In both cases, the received `size` will be for the whole serialized payload.

Parameters

- **size** – **[in]** Number of bytes required for the serialized payload.
- **cache_change** – **[inout]** Cache change to assign the payload to

Returns whether the operation succeeded or not

Pre Fields `writerGUID` and `sequenceNumber` of `cache_change` are either:

- Both equal to `unknown` (meaning a writer is creating a new change)
- Both different from `unknown` (meaning a reader has received the first fragment of a cache change)

Post

- Field `cache_change.payload_owner` equals this
- Field `serializedPayload.data` points to a buffer of at least `size` bytes
- Field `serializedPayload.max_size` is greater than or equal to `size`

virtual bool **get_payload**(*SerializedPayload_t* &data, *IPayloadPool* *&data_owner, *CacheChange_t* &cache_change) = 0
Assign a serialized payload to a new sample.

This method will usually be called when a reader receives a whole cache change.

Note: `data` and `data_owner` are received as references to accommodate the case where several readers receive the same payload. If the payload has no owner, it means it is allocated on the stack of a reception thread, and a copy should be performed. The pool may decide in that case to point `data.data` to the new copy and take ownership of the payload. In that case, when the reception thread is done with the payload (after all readers have been informed of the received data), method `release_payload` will be called to indicate that the reception thread is not using the payload anymore.

Warning: `data_owner` can only be changed from `nullptr` to `this`. If a value different from `nullptr` is received it should be left unchanged.

Warning: `data` fields can only be changed when `data_owner` is `nullptr`. If a value different from `nullptr` is received all fields in `data` should be left unchanged.

Parameters

- **data** – **[inout]** Serialized payload received
- **data_owner** – **[inout]** Payload pool owning incoming data
- **cache_change** – **[inout]** Cache change to assign the payload to

Returns whether the operation succeeded or not

Pre

- Field `cache_change.writerGUID` is not `unknown`
- Field `cache_change.sequenceNumber` is not `unknown`

Post

- Field `cache_change.payload_owner` equals this
- Field `cache_change.serializedPayload.data` points to a buffer of at least `data.length` bytes
- Field `cache_change.serializedPayload.length` is equal to `data.length`
- Field `cache_change.serializedPayload.max_size` is greater than or equal to `data.length`
- Content of `cache_change.serializedPayload.data` is the same as `data.data`

virtual bool **release_payload**(*CacheChange_t* &cache_change) = 0

Release a serialized payload from a sample.

This method will be called when a cache change is removed from a history.

Parameters **cache_change** – **[inout]** Cache change to assign the payload to

Returns whether the operation succeeded or not

Pre

- Field `payload_owner` of `cache_change` equals this

Post

- Field `payload_owner` of `cache_change` is `nullptr`

ReaderHistory

class **ReaderHistory** : public eprosima::fastrtps::rtps::*History*

Class *ReaderHistory*, container of the different CacheChanges of a reader

Public Functions

ReaderHistory(const *HistoryAttributes* &att)
Constructor of the *ReaderHistory*. It needs a *HistoryAttributes*.

virtual bool **can_change_be_added_nts**(const *GUID_t* &writer_guid, uint32_t total_payload_size, size_t unknown_missing_changes_up_to, bool &will_never_be_accepted) const
Check if a new change can be added to this history.

Parameters

- **writer_guid** – **[in]** GUID of the writer where the change came from.
- **total_payload_size** – **[in]** Total payload size of the incoming change.
- **unknown_missing_changes_up_to** – **[in]** The number of changes from the same writer with a lower sequence number that could potentially be received in the future.
- **will_never_be_accepted** – **[out]** When the method returns `false`, this parameter will inform whether the change could be accepted in the future or not.

Pre change should not be present in the history

Returns Whether a call to received_change will succeed when called with the same arguments.

virtual bool **received_change**(*CacheChange_t* *change, size_t unknown_missing_changes_up_to)
Virtual method that is called when a new change is received. In this implementation this method just calls add_change. The user can overload this method in case he needs to perform additional checks before adding the change.

Parameters

- **change** – Pointer to the change
- **unknown_missing_changes_up_to** – The number of changes from the same writer with a lower sequence number that could potentially be received in the future.

Returns True if added.

inline virtual bool **received_change**(*CacheChange_t* *change, size_t unknown_missing_changes_up_to, fastdds::dds::*SampleRejectedStatusKind* &rejection_reason)
Virtual method that is called when a new change is received. In this implementation this method just calls add_change. The user can overload this method in case he needs to perform additional checks before adding the change.

Parameters

- **change** – **[in]** Pointer to the change
- **unknown_missing_changes_up_to** – **[in]** The number of changes from the same writer with a lower sequence number that could potentially be received in the future.
- **rejection_reason** – **[out]** In case of been rejected the sample, it will contain the reason of the rejection.

Returns True if added.

inline virtual bool **completed_change**(rtps::*CacheChange_t* *change)
Called when a fragmented change is received completely by the Subscriber. Will find its instance and store it.

Parameters **change** – **[in]** The received change

Pre Change should be already present in the history.

Returns

inline virtual bool **completed_change**(*CacheChange_t* *change, size_t unknown_missing_changes_up_to, fastdds::dds::*SampleRejectedStatusKind* &rejection_reason)

Called when a fragmented change is received completely by the Subscriber. Will find its instance and store it.

Parameters

- **change** – **[in]** The received change
- **unknown_missing_changes_up_to** – **[in]** Number of missing changes before this one
- **rejection_reason** – **[out]** In case of been rejected the sample, it will contain the reason of the rejection.

Pre Change should be already present in the history.

Returns

bool **add_change**(*CacheChange_t* *a_change)

Add a *CacheChange_t* to the *ReaderHistory*.

Parameters **a_change** – Pointer to the CacheChange to add.

Returns True if added.

virtual iterator **remove_change_nts**(const_iterator removal, bool release = true) override

Remove a specific change from the history. No Thread Safe

Parameters

- **removal** – iterator to the change for removal
- **release** – specifies if the change must be returned to the pool

Returns iterator to the next change if any

virtual iterator **remove_change_nts**(const_iterator removal, const std::chrono::time_point<std::chrono::steady_clock> &max_blocking_time, bool release = true) override

Remove a specific change from the history. No Thread Safe

Parameters

- **removal** – iterator to the change for removal
- **max_blocking_time** – **[in]** Maximum time this method has to complete the task.
- **release** – specifies if the change must be returned to the pool

Returns iterator to the next change if any

virtual bool **matches_change**(const *CacheChange_t* *inner, *CacheChange_t* *outer) override

Criteria to search a specific *CacheChange_t* on history

Parameters

- **inner** – change to compare
- **outer** – change for comparison

Returns true if inner matches outer criteria

bool **remove_changes_with_guid**(const *GUID_t* &a_guid)

Remove all changes from the *History* that have a certain guid.

Parameters **a_guid** – Pointer to the target guid to search for.

Returns True if successful, even if no changes have been removed.

bool **remove_fragmented_changes_until**(const *SequenceNumber_t* &seq_num, const *GUID_t* &writer_guid)

Remove all fragmented changes from certain writer up to certain sequence number.

Parameters

- **seq_num** – First *SequenceNumber_t* not to be removed.
- **writer_guid** – GUID of the writer for which changes should be looked for.

Returns True if successful, even if no changes have been removed.

virtual void **writer_unmatched**(const *GUID_t* &writer_guid, const *SequenceNumber_t* &last_notified_seq)

Called when a writer is unmatched from the reader holding this history.

This method will remove all the changes on the history that came from the writer being unmatched and which have not yet been notified to the user.

Parameters

- **writer_guid** – GUID of the writer being unmatched.
- **last_notified_seq** – Last sequence number from the specified writer that was notified to the user.

inline virtual void **writer_update_its_ownership_strength_nts**(const *GUID_t* &writer_guid, const uint32_t ownership_strength)

This function should be called by reader if a writer updates its ownership strength.

Parameters

- **writer_guid** – **[in]** Guid of the writer which changes its ownership strength.
- **ownership_strength** – **[out]** New value of the writer's Ownership strength.

bool **remove_change**(*CacheChange_t* *ch)

Introduce base class method into scope.

bool **remove_change**(*CacheChange_t* *ch, const std::chrono::time_point<std::chrono::steady_clock> &max_blocking_time)

Introduce base class method into scope.

inline iterator **remove_change**(const_iterator removal, bool release = true)

Introduce base class method into scope.

WriterHistory

class **WriterHistory** : public eprosima::fastrtps::rtps::*History*

Class *WriterHistory*, container of the different CacheChanges of a writer

Public Functions

WriterHistory(const *HistoryAttributes* &att)
Constructor of the *WriterHistory*.

bool **add_change**(*CacheChange_t* *a_change)
Add a *CacheChange_t* to the *WriterHistory*.

Parameters **a_change** – Pointer to the *CacheChange_t* to be added.

Returns True if added.

bool **add_change**(*CacheChange_t* *a_change, *WriteParams* &wparams)
Add a *CacheChange_t* to the *WriterHistory*.

Parameters

- **a_change** – Pointer to the *CacheChange_t* to be added.
- **wparams** – Extra write parameters.

Returns True if added.

virtual iterator **remove_change_nts**(const_iterator removal, bool release = true) override
Remove a specific change from the history. No Thread Safe

Parameters

- **removal** – iterator to the change for removal
- **release** – specifies if the change should be return to the pool

Returns iterator to the next change if any

virtual iterator **remove_change_nts**(const_iterator removal, const std::chrono::time_point<std::chrono::steady_clock> &max_blocking_time, bool release = true) override
Remove a specific change from the history. No Thread Safe

Parameters

- **removal** – iterator to the change for removal
- **release** – specifies if the change should be return to the pool
- **max_blocking_time** – **[in]** Maximum time this method has to complete the task.

Returns iterator to the next change if any

virtual bool **matches_change**(const *CacheChange_t* *inner, *CacheChange_t* *outer) override
Criteria to search a specific *CacheChange_t* on history

Parameters

- **inner** – change to compare
- **outer** – change for comparison

Returns true if inner matches outer criteria

bool **remove_min_change**()
Remove the *CacheChange_t* with the minimum sequenceNumber.

Returns True if correctly removed.

bool **remove_min_change**(const std::chrono::time_point<std::chrono::steady_clock> &max_blocking_time)
Remove the *CacheChange_t* with the minimum sequenceNumber.

Parameters `max_blocking_time` – **[in]** Maximum time this method has to complete the task.

Returns True if correctly removed.

bool **remove_change**(*CacheChange_t* *ch)
Introduce base class method into scope.

bool **remove_change**(*CacheChange_t* *ch, const std::chrono::time_point<std::chrono::steady_clock> &max_blocking_time)
Introduce base class method into scope.

inline iterator **remove_change**(const_iterator removal, bool release = true)
Introduce base class method into scope.

RTPSParticipant

ParticipantDiscoveryInfo

ParticipantAuthenticationInfo

struct **ParticipantAuthenticationInfo**

Public Members

AUTHENTICATION_STATUS **status**
Status.

GUID_t **guid**
Associated GUID.

inline bool eprosima::fastrtps::rtps::**operator==**(const *ParticipantAuthenticationInfo* &l, const *ParticipantAuthenticationInfo* &r)

ParticipantDiscoveryInfo

struct **ParticipantDiscoveryInfo**
Class *ParticipantDiscoveryInfo* with discovery information of the Participant.

Public Types

enum **DISCOVERY_STATUS**
Enum DISCOVERY_STATUS, four different status for discovered participants.

Values:

enumerator **DISCOVERED_PARTICIPANT**

enumerator **CHANGED_QOS_PARTICIPANT**

enumerator **REMOVED_PARTICIPANT**

enumerator **DROPPED_PARTICIPANT**

enumerator **IGNORED_PARTICIPANT**

Public Members

DISCOVERY_STATUS **status**
Status.

const *ParticipantProxyData* &**info**
Participant discovery info.

Todo:
This is a reference to an object that could be deleted, thus it should not be a reference (intraprocess case -> BlackboxTests_DDS_PIM.DDSDiscovery.ParticipantProxyPhysicalData).

ParticipantProxyData

class **ParticipantProxyData**
ParticipantProxyData class is used to store and convert the information Participants send to each other during the PDP phase.

Public Functions

bool **updateData**(*ParticipantProxyData* &pdata)
Update the data.

Parameters **pdata** – Object to copy the data from

Returns True on success

uint32_t **get_serialized_size**(bool include_encapsulation) const
Get the size in bytes of the CDR serialization of this object.

Parameters **include_encapsulation** – Whether to include the size of the encapsulation info.

Returns size in bytes of the CDR serialization.

bool **writeToCDRMessage**(*CDRMessage_t* *msg, bool write_encapsulation)
Write as a parameter list on a *CDRMessage_t*

Returns True on success

bool **readFromCDRMessage**(*CDRMessage_t* *msg, bool use_encapsulation, const NetworkFactory &network, bool is_shm_transport_available, bool should_filter_locators, fastdds::rtps::*VendorId_t* source_vendor_id = c_VendorId_eProsima)
Read the parameter list from a received *CDRMessage_t*

Returns True on success

void **clear**()
Clear the data (restore to default state).

void **copy**(const *ParticipantProxyData* &pdata)
Copy the data from another object.

Parameters **pdata** – Object to copy the data from

void **set_persistence_guid**(const *GUID_t* &guid)
Set participant persistent *GUID_t*

Parameters **guid** – valid *GUID_t*

GUID_t **get_persistence_guid**() const
Retrieve participant persistent *GUID_t*

Returns guid persistent *GUID_t* or c_Guid_Unknown

void **set_sample_identity**(const *SampleIdentity* &sid)
Set participant client server sample identity

Parameters **sid** – valid *SampleIdentity*

SampleIdentity **get_sample_identity**() const
Retrieve participant *SampleIdentity*

Returns *SampleIdentity*

void **set_backup_stamp**(const *GUID_t* &guid)
Identifies the participant as client of the given server

Parameters **guid** – valid backup server GUID

GUID_t **get_backup_stamp**() const
Retrieves BACKUP server stamp. On deserialization hints if lease duration must be enforced

Returns GUID

Public Members

ProtocolVersion_t **m_protocolVersion**
Protocol version.

GUID_t **m_guid**
GUID.

fastdds::rtps::*VendorId_t* **m_VendorId**
Vendor ID.

bool **m_expectsInlineQos**
Expects Inline QOS.

BuiltinEndpointSet_t **m_availableBuiltinEndpoints**
Available builtin endpoints.

NetworkConfigSet_t **m_networkConfiguration**
Network configuration.

RemoteLocatorList **metatraffic_locators**
Metatraffic locators.

RemoteLocatorList **default_locators**
Default locators.

Count_t **m_manualLivelinessCount**
Manual liveliness count.

string_255 **m_participantName**
Participant name.

BUILTIN_PARTICIPANT_DATA_MAX_SIZE 100

TYPELOOKUP_DATA_MAX_SIZE 5000

DISC_BUILTIN_ENDPOINT_PARTICIPANT_ANNOUNCER (0x00000001 << 0)

DISC_BUILTIN_ENDPOINT_PARTICIPANT_DETECTOR (0x00000001 << 1)

DISC_BUILTIN_ENDPOINT_PUBLICATION_ANNOUNCER (0x00000001 << 2)

DISC_BUILTIN_ENDPOINT_PUBLICATION_DETECTOR (0x00000001 << 3)

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_ANNOUNCER (0x00000001 << 4)

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_DETECTOR (0x00000001 << 5)

DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_ANNOUNCER (0x00000001 << 6)

DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_DETECTOR (0x00000001 << 7)

DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_ANNOUNCER (0x00000001 << 8)

DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_DETECTOR (0x00000001 << 9)

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_WRITER (0x00000001 << 10)

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_READER (0x00000001 << 11)

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REQUEST_DATA_WRITER (0x00000001 << 12)

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REQUEST_DATA_READER (0x00000001 << 13)

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REPLY_DATA_WRITER (0x00000001 << 14)

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REPLY_DATA_READER (0x00000001 << 15)

DISC_BUILTIN_ENDPOINT_PUBLICATION_SECURE_ANNOUNCER (0x00000001 << 16)

DISC_BUILTIN_ENDPOINT_PUBLICATION_SECURE_DETECTOR (0x00000001 << 17)

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_SECURE_ANNOUNCER (0x00000001 << 18)

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_SECURE_DETECTOR (0x00000001 << 19)

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_SECURE_DATA_WRITER (0x00000001 << 20)

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_SECURE_DATA_READER (0x00000001 << 21)

DISC_BUILTIN_ENDPOINT_PARTICIPANT_SECURE_ANNOUNCER (0x00000001 << 26)

DISC_BUILTIN_ENDPOINT_PARTICIPANT_SECURE_DETECTOR (0x00000001 << 27)

ReaderDiscoveryInfo

struct **ReaderDiscoveryInfo**
Class *ReaderDiscoveryInfo* with discovery information of the reader.

Public Types

enum **DISCOVERY_STATUS**
Enum DISCOVERY_STATUS, four different status for discovered readers.

Values:

enumerator **DISCOVERED_READER**

enumerator **CHANGED_QOS_READER**

enumerator **REMOVED_READER**

enumerator **IGNORED_READER**

Public Members

DISCOVERY_STATUS **status**
Status.

const *ReaderProxyData* &**info**
Participant discovery info.

ReaderProxyData

class **ReaderProxyData**
Class *ReaderProxyData*, used to represent all the information on a Reader (both local and remote) with the purpose of implementing the discovery.

-

Public Functions

inline void **set_sample_identity**(const *SampleIdentity* &sid)
Set participant client server sample identity

Parameters **sid** – valid *SampleIdentity*

inline *SampleIdentity* **get_sample_identity**() const
Retrieve participant *SampleIdentity*

Returns *SampleIdentity*

uint32_t **get_serialized_size**(bool include_encapsulation) const
Get the size in bytes of the CDR serialization of this object.

Parameters **include_encapsulation** – Whether to include the size of the encapsulation info.

Returns size in bytes of the CDR serialization.

bool **writeToCDRMessage**(*CDRMessage_t* *msg, bool write_encapsulation) const
Write as a parameter list on a *CDRMessage_t*

Returns True on success

bool **readFromCDRMessage**(*CDRMessage_t* *msg, const NetworkFactory &network, bool is_shm_transport_available, bool should_filter_locators, fastdds::rtps::*VendorId_t* source_vendor_id = c_VendorId_eProsima)
Read the information from a *CDRMessage_t*. The position of the message must be in the beginning on the parameter list.

Parameters

- **msg** – Pointer to the message.
- **network** – Reference to network factory for locator validation and transformation
- **is_shm_transport_available** – Indicates whether the Reader is reachable by SHM.
- **should_filter_locators** – Whether to retrieve the locators before the external locators filtering
- **source_vendor_id** – VendorId of the source participant from which the message was received

Returns true on success

void **clear**()
Clear (put to default) the information.

bool **is_update_allowed**(const *ReaderProxyData* &rdata) const
Check if this object can be updated with the information on another object.

Parameters **rdata** – *ReaderProxyData* object to be checked.

Returns true if this object can be updated with the information on rdata.

void **update**(*ReaderProxyData* *rdata)
Update the information (only certain fields will be updated).

Parameters **rdata** – Pointer to the object from which we are going to update.

void **copy**(*ReaderProxyData* *rdata)
Copy ALL the information from another object.

Parameters **rdata** – Pointer to the object from where the information must be copied.

Public Members

ReaderQos **m_qos**
Reader Qos.

security::EndpointSecurityAttributesMask **security_attributes_**
EndpointSecurityInfo.endpoint_security_attributes.

security::PluginEndpointSecurityAttributesMask **plugin_security_attributes_**
EndpointSecurityInfo.plugin_endpoint_security_attributes.

WriterDiscoveryInfo

struct **WriterDiscoveryInfo**
Class *WriterDiscoveryInfo* with discovery information of the writer.

Public Types

enum **DISCOVERY_STATUS**
Enum DISCOVERY_STATUS, four different status for discovered writers.

Values:

enumerator **DISCOVERED_WRITER**

enumerator **CHANGED_QOS_WRITER**

enumerator **REMOVED_WRITER**

enumerator **IGNORED_WRITER**

Public Members

DISCOVERY_STATUS **status**
Status.

const *WriterProxyData* &**info**
Participant discovery info.

WriterProxyData

class **WriterProxyData**

Public Functions

inline void **set_sample_identity**(const *SampleIdentity* &sid)
Set participant client server sample identity

Parameters **sid** – valid *SampleIdentity*

inline *SampleIdentity* **get_sample_identity**() const
Retrieve participant *SampleIdentity*

Returns *SampleIdentity*

void **clear**()
Clear the information and return the object to the default state.

bool **is_update_allowed**(const *WriterProxyData* &wdata) const
Check if this object can be updated with the information on another object.

Parameters **wdata** – *WriterProxyData* object to be checked.

Returns true if this object can be updated with the information on wdata.

void **update**(*WriterProxyData* *wdata)
Update certain parameters from another object.

Parameters **wdata** – pointer to object with new information.

void **copy**(*WriterProxyData* *wdata)
Copy all information from another object.

uint32_t **get_serialized_size**(bool include_encapsulation) const
Get the size in bytes of the CDR serialization of this object.

Parameters **include_encapsulation** – Whether to include the size of the encapsulation info.

Returns size in bytes of the CDR serialization.

bool **writeToCDRMessage**(*CDRMessage_t* *msg, bool write_encapsulation) const
Write as a parameter list on a *CDRMessage_t*.

bool **readFromCDRMessage**(*CDRMessage_t* *msg, const NetworkFactory &network, bool is_shm_transport_possible, bool should_filter_locators, fastdds::rtps::*VendorId_t* source_vendor_id = c_VendorId_eProsima)
Read a parameter list from a *CDRMessage_t*.

Public Members

WriterQos **m_qos**
WriterQOS.

security::EndpointSecurityAttributesMask **security_attributes_**
EndpointSecurityInfo.endpoint_security_attributes.

security::PluginEndpointSecurityAttributesMask **plugin_security_attributes_**
EndpointSecurityInfo.plugin_endpoint_security_attributes.

RTPSParticipant

class **RTPSParticipant**
Class *RTPSParticipant*, contains the public API for a *RTPSParticipant*.

Public Functions

const *GUID_t* &**getGuid**() const
Get the *GUID_t* of the *RTPSParticipant*.

void **announceRTPSParticipantState**()
Force the announcement of the *RTPSParticipant* state.

void **stopRTPSParticipantAnnouncement**()
Stop the *RTPSParticipant* announcement period. //TODO remove this method because is only for testing.

void **resetRTPSParticipantAnnouncement**()
Reset the *RTPSParticipant* announcement period. //TODO remove this method because is only for testing.

bool **newRemoteWriterDiscovered**(const *GUID_t* &pguid, int16_t userDefinedId)
Indicate the Participant that you have discovered a new Remote Writer. This method can be used by the user to implements its own Static *Endpoint* Discovery Protocol

Parameters

- **pguid** – *GUID_t* of the discovered Writer.
- **userDefinedId** – ID of the discovered Writer.

Returns True if correctly added.

bool **newRemoteReaderDiscovered**(const *GUID_t* &pguid, int16_t userDefinedId)
Indicate the Participant that you have discovered a new Remote Reader. This method can be used by the user to implements its own Static *Endpoint* Discovery Protocol

Parameters

- **pguid** – *GUID_t* of the discovered Reader.
- **userDefinedId** – ID of the discovered Reader.

Returns True if correctly added.

uint32_t **getRTPSParticipantID**() const
Get the Participant ID.

Returns Participant ID.

bool **registerWriter**(*RTPSWriter* *Writer, const TopicAttributes &topicAtt, const WriterQos &wqos)
Register a *RTPSWriter* in the builtin Protocols.

Parameters

- **Writer** – Pointer to the *RTPSWriter*.
- **topicAtt** – Topic Attributes where you want to register it.
- **wqos** – WriterQos.

Returns True if correctly registered.

bool **registerReader**(*RTPSReader* *Reader, const TopicAttributes &topicAtt, const ReaderQos &rqos,
const fastdds::rtps::*ContentFilterProperty* *content_filter = nullptr)
Register a *RTPSReader* in the builtin Protocols.

Parameters

- **Reader** – Pointer to the *RTPSReader*.
- **topicAtt** – Topic Attributes where you want to register it.
- **rqos** – ReaderQos.
- **content_filter** – Optional content filtering information.

Returns True if correctly registered.

void **update_attributes**(const *RTPSParticipantAttributes* &patt)
Update participant attributes.

Parameters **patt** – New participant attributes.

bool **updateWriter**(*RTPSWriter* *Writer, const TopicAttributes &topicAtt, const WriterQos &wqos)
Update writer QOS

Parameters

- **Writer** – to update
- **topicAtt** – Topic Attributes where you want to register it.
- **wqos** – New writer QoS

Returns true on success

bool **updateReader**(*RTPSReader* *Reader, const TopicAttributes &topicAtt, const ReaderQos &rqos, const fastdds::rtps::*ContentFilterProperty* *content_filter = nullptr)

Update reader QOS

Parameters

- **Reader** – Pointer to the *RTPSReader* to update
- **topicAtt** – Topic Attributes where you want to register it.
- **rqos** – New reader QoS
- **content_filter** – Optional content filtering information.

Returns true on success

std::vector<std::string> **getParticipantNames**() const

Returns a list with the participant names.

Returns list of participant names.

const *RTPSParticipantAttributes* &**getRTPSParticipantAttributes**() const

Get a copy of the actual state of the RTPSParticipantParameters

Returns *RTPSParticipantAttributes* copy of the params.

uint32_t **getMaxMessageSize**() const

Retrieves the maximum message size.

uint32_t **getMaxDataSize**() const

Retrieves the maximum data size.

WLP ***wlp**() const

A method to retrieve the built-in writer liveliness protocol.

Returns Writer liveliness protocol

bool **get_new_entity_id**(*EntityId_t* &entityId)

Fills a new entityId if set to unknown, or checks if a entity already exists with that entityId in other case.

Parameters **entityId** – to check of fill. If filled, EntityKind will be "vendor-specific" (0x01)

Returns True if filled or the entityId is available.

void **set_check_type_function**(std::function<bool(const std::string&)> &&check_type)

Allows setting a function to check if a type is already known by the top level API participant.

fastdds::dds::builtin::TypeLookupManager ***typelookup_manager**() const

Retrieves the built-in typelookup service manager.

Returns

void **set_listener**(*RTPSParticipantListener* *listener)

Modifies the participant listener.

Parameters **listener** –

uint32_t **get_domain_id**() const

Retrieves the DomainId.

void **enable**()
This operation enables the RTPSParticipantImpl.

bool **ignore_participant**(const *GuidPrefix_t* &participant_guid)
Ignore all messages coming from the *RTPSParticipant*.

Parameters **participant_guid** – **[in]** *RTPSParticipant* GUID to be ignored

Returns True if correctly included into the ignore collection. False otherwise.

bool **ignore_writer**(const *GUID_t* &writer_guid)
Ignore all messages coming from the *RTPSWriter*.

Parameters **writer_guid** – **[in]** *RTPSWriter* GUID to be ignored

Returns True if correctly included into the ignore collection. False otherwise.

bool **ignore_reader**(const *GUID_t* &reader_guid)
Ignore all messages coming from the *RTPSReader*.

Parameters **reader_guid** – **[in]** *RTPSReader* GUID to be ignored

Returns True if correctly included into the ignore collection. False otherwise.

std::vector<fastdds::rtps::TransportNetmaskFilterInfo> **get_netmask_filter_info**() const
Returns registered transports' netmask filter information (transport's netmask filter kind and allowlist).

Returns A vector with all registered transports' netmask filter information.

bool **is_security_enabled_for_writer**(const *WriterAttributes* &writer_attributes)
Checks whether the writer has security attributes enabled.

Parameters **writer_attributes** – Attributes of the writer as given to the RTPSParticipantImpl::create_writer

bool **is_security_enabled_for_reader**(const *ReaderAttributes* &reader_attributes)
Checks whether the reader has security attributes enabled.

Parameters **reader_attributes** – Attributes of the reader as given to the RTPSParticipantImpl::create_reader

RTPSParticipantListener

class **RTPSParticipantListener**
Class *RTPSParticipantListener* with virtual method that the user can overload to respond to certain events.

Public Functions

inline virtual void **onParticipantDiscovery**(*RTPSParticipant* *participant, *ParticipantDiscoveryInfo* &&info)
This method is called when a new Participant is discovered, or a previously discovered participant changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote participant.
- **info** – **[out]** Remote participant information. User can take ownership of the object.

inline virtual void **onParticipantDiscovery**(*RTPSParticipant* *participant, *ParticipantDiscoveryInfo* &&info, bool &should_be_ignored)

This method is called when a new Participant is discovered, or a previously discovered participant changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote participant.
- **info** – **[out]** Remote participant information. User can take ownership of the object.
- **should_be_ignored** – **[out]** Flag to indicate the library to automatically ignore the discovered Participant.

inline virtual void **onReaderDiscovery**(*RTPSParticipant* *participant, *ReaderDiscoveryInfo* &&info)

This method is called when a new Reader is discovered, or a previously discovered reader changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote reader.
- **info** – **[out]** Remote reader information. User can take ownership of the object.

inline virtual void **onReaderDiscovery**(*RTPSParticipant* *participant, *ReaderDiscoveryInfo* &&info, bool &should_be_ignored)

This method is called when a new Reader is discovered, or a previously discovered reader changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote reader.
- **info** – **[out]** Remote reader information. User can take ownership of the object.
- **should_be_ignored** – **[out]** Flag to indicate the library to automatically ignore the discovered Reader.

inline virtual void **onWriterDiscovery**(*RTPSParticipant* *participant, *WriterDiscoveryInfo* &&info)

This method is called when a new Writer is discovered, or a previously discovered writer changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote writer.
- **info** – **[out]** Remote writer information. User can take ownership of the object.

inline virtual void **onWriterDiscovery**(*RTPSParticipant* *participant, *WriterDiscoveryInfo* &&info, bool &should_be_ignored)

This method is called when a new Writer is discovered, or a previously discovered writer changes its QOS or is removed.

Parameters

- **participant** – **[out]** Pointer to the Participant which discovered the remote writer.
- **info** – **[out]** Remote writer information. User can take ownership of the object.
- **should_be_ignored** – **[out]** Flag to indicate the library to automatically ignore the discovered Writer.

inline virtual void **on_type_discovery**(*RTPSParticipant* *participant, const *SampleIdentity* &request_sample_id, const string_255 &topic, const types::TypeIdentifier *identifier, const types::TypeObject *object, types::DynamicType_ptr dyn_type)

This method is called when a participant discovers a new Type The ownership of all object belongs to the caller so if needs to be used after the method ends, a full copy should be perform (except for dyn_type due to its shared_ptr nature.

The field "topic" it is only available if the type was discovered using "Discovery-Time Data Typing", in which case the field request_sample_id will contain INVALID_SAMPLE_IDENTITY. If the type was discovered using TypeLookup Service then "topic" will be empty, but will have the request_sample_id of the petition that caused the discovery. For example: fastrtps::types::TypeIdentifier new_type_id = *identifier;

inline virtual void **on_type_dependencies_reply**(*RTPSParticipant* *participant, const *SampleIdentity* &request_sample_id, const types::TypeIdentifierWithSizeSeq &dependencies)

This method is called when the typelookup client received a reply to a getTypeDependencies request.

The user may want to retrieve these new types using the getTypes request and create a new DynamicType using the retrieved TypeObject.

inline virtual void **on_type_information_received**(*RTPSParticipant* *participant, const string_255 &topic_name, const string_255 &type_name, const types::TypeInformation &type_information)

This method is called when a participant receives a TypeInformation while discovering another participant.

RTPSReader

ReaderListener

class **ReaderListener**

Class *ReaderListener*, to be used by the user to override some of is virtual method to program actions to certain events.

Subclassed by eprosima::fastdds::dds::builtin::TypeLookupReplyListener, eprosima::fastdds::dds::builtin::TypeLookupRequestLis eprosima::fastrtps::rtps::PDPListener, eprosima::fastrtps::rtps::WLPListener

Public Functions

inline virtual void **onReaderMatched**(*RTPSReader* *reader, *MatchingInfo* &info)

This method is invoked when a new reader matches

Parameters

- **reader** – Matching reader
- **info** – Matching information of the reader

inline virtual void **onReaderMatched**(*RTPSReader* *reader, const fastdds::dds::*SubscriptionMatchedStatus* &info)

This method is invoked when a new reader matches

Parameters

- **reader** – Matching reader
- **info** – Subscription matching information

inline virtual void **onNewCacheChangeAdded**(*RTPSReader* *reader, const *CacheChange_t* *const change)
This method is called when a new *CacheChange_t* is added to the *ReaderHistory*.

Parameters

- **reader** – Pointer to the reader.
- **change** – Pointer to the *CacheChange_t*. This is a const pointer to const data to indicate that the user should not dispose of this data himself. To remove the data call the remove_change method of the *ReaderHistory*. reader->getHistory()->remove_change((CacheChange_t*)change).

inline virtual void **on_liveliness_changed**(*RTPSReader* *reader, const eprosima::fastdds::dds::*LivelinessChangedStatus* &status)
Method called when the liveliness of a reader changes.

Parameters

- **reader** – The reader
- **status** – The liveliness changed status

inline virtual void **on_requested_incompatible_qos**(*RTPSReader* *reader, eprosima::fastdds::dds::PolicyMask qos)
This method is called when a new Writer is discovered, with a Topic that matches that of a local reader, but with an offered QoS that is incompatible with the one requested by the local reader

Parameters

- **reader** – Pointer to the *RTPSReader*.
- **qos** – A mask with the bits of all incompatible Qos activated.

inline virtual void **on_sample_lost**(*RTPSReader* *reader, int32_t sample_lost_since_last_update)
This method is called when the reader detects that one or more samples have been lost.

Parameters

- **reader** – Pointer to the *RTPSReader*.
- **sample_lost_since_last_update** – The number of samples that were lost since the last time this method was called for the same reader.

inline virtual void **on_writer_discovery**(*RTPSReader* *reader, *WriterDiscoveryInfo*::*DISCOVERY_STATUS* reason, const *GUID_t* &writer_guid, const *WriterProxyData* *writer_info)
Method called when the discovery information of a writer regarding a reader changes.

Parameters

- **reader** – The reader.
- **reason** – The reason motivating this method to be called.
- **writer_guid** – The GUID of the writer for which the discovery information changed.
- **writer_info** – Discovery information about the writer. Will be `nullptr` for reason `REMOVED_WRITER`.

inline virtual void **on_sample_rejected**(*RTPSReader* *reader, eprosima::fastdds::dds::*SampleRejectedStatusKind* reason, const *CacheChange_t* *const change)
This method is called when the reader rejects a samples.

Parameters

- **reader** – Pointer to the *RTPSReader*.
- **reason** – Indicates reason for sample rejection.
- **change** – Pointer to the *CacheChange_t*. This is a const pointer to const data to indicate that the user should not dispose of this data himself.

inline virtual void **on_data_available**(*RTPSReader* *reader, const *GUID_t* &writer_guid, const *SequenceNumber_t* &first_sequence, const *SequenceNumber_t* &last_sequence, bool &should_notify_individual_changes)

This method is called when new *CacheChange_t* objects are made available to the user.

Note: This method is currently never called. Implementation will be added in future releases.

Parameters

- **reader** – **[in]** Pointer to the reader performing the notification.
- **writer_guid** – **[in]** GUID of the writer from which the changes were received.
- **first_sequence** – **[in]** Sequence number of the first change made available.
- **last_sequence** – **[in]** Sequence number of the last change made available. It will always be greater or equal than `first_sequence`.
- **should_notify_individual_changes** – **[out]** Whether the individual changes should be notified by means of `onNewCacheChangeAdded`.

inline virtual void **on_incompatible_type**(*RTPSReader* *reader)

This method is called when a new Writer is discovered, with a Topic that matches the name of a local reader, but with an incompatible type

Parameters **reader** – Pointer to the *RTPSReader*.

RTPSReader

class **RTPSReader** : public eprosima::fastrtps::rtps::*Endpoint*, public eprosima::fastdds::statistics::StatisticsReaderImpl

Class *RTPSReader*, manages the reception of data from its matched writers.

Subclassed by eprosima::fastrtps::rtps::StatefulReader, eprosima::fastrtps::rtps::StatelessReader

Public Functions

virtual bool **matched_writer_add**(const *WriterProxyData* &wdata) = 0

Add a matched writer represented by its attributes.

Parameters **wdata** – Attributes of the writer to add.

Returns True if correctly added.

virtual bool **matched_writer_remove**(const *GUID_t* &writer_guid, bool removed_by_lease = false) = 0

Remove a writer represented by its attributes from the matched writers.

Parameters

- **writer_guid** – GUID of the writer to remove.

- **removed_by_lease** – Whether the writer is being unmatched due to a participant drop.

Returns True if correctly removed.

virtual bool **matched_writer_is_matched**(const *GUID_t* &writer_guid) = 0
Tells us if a specific Writer is matched against this reader.

Parameters **writer_guid** – GUID of the writer to check.

Returns True if it is matched.

virtual bool **processDataMsg**(*CacheChange_t* *change) = 0
Processes a new DATA message. Previously the message must have been accepted by function acceptMsgDirectedTo.

Parameters **change** – Pointer to the *CacheChange_t*.

Returns true if the reader accepts messages from the.

virtual bool **processDataFragMsg**(*CacheChange_t* *change, uint32_t sampleSize, uint32_t fragmentStartingNum, uint16_t fragmentsInSubmessage) = 0
Processes a new DATA FRAG message.

Parameters

- **change** – Pointer to the *CacheChange_t*.
- **sampleSize** – Size of the complete, assembled message.
- **fragmentStartingNum** – Starting number of this particular message.
- **fragmentsInSubmessage** – Number of fragments on this particular message.

Returns true if the reader accepts message.

virtual bool **processHeartbeatMsg**(const *GUID_t* &writerGUID, uint32_t hbCount, const *SequenceNumber_t* &firstSN, const *SequenceNumber_t* &lastSN, bool finalFlag, bool livelinessFlag, fastdds::rtps::*VendorId_t* origin_vendor_id = c_VendorId_Unknown) = 0
Processes a new HEARTBEAT message.

Parameters

- **writerGUID** –
- **hbCount** –
- **firstSN** –
- **lastSN** –
- **finalFlag** –
- **livelinessFlag** –
- **origin_vendor_id** –

Returns true if the reader accepts messages from the.

virtual bool **processGapMsg**(const *GUID_t* &writerGUID, const *SequenceNumber_t* &gapStart, const *SequenceNumberSet_t* &gapList, fastdds::rtps::*VendorId_t* origin_vendor_id = c_VendorId_Unknown) = 0
Processes a new GAP message.

Parameters

- **writerGUID** –
- **gapStart** –
- **gapList** –
- **origin_vendor_id** –

Returns true if the reader accepts messages from the.

virtual bool **change_removed_by_history**(*CacheChange_t* *change, WriterProxy *prox = nullptr) = 0
Method to indicate the reader that some change has been removed due to HistoryQos requirements.

Parameters

- **change** – Pointer to the *CacheChange_t*.
- **prox** – Pointer to the WriterProxy.

Returns True if correctly removed.

ReaderListener ***getListener**() const
Get the associated listener, secondary attached Listener in case it is of compound type

Returns Pointer to the associated reader listener.

bool **setListener**(*ReaderListener* *target)
Switch the *ReaderListener* kind for the Reader. If the *RTPSReader* does not belong to the built-in protocols it switches out the old one. If it belongs to the built-in protocols, it sets the new *ReaderListener* callbacks to be called after the built-in *ReaderListener* ones.

Parameters **target** – Pointed to ReaderLister to attach

Returns True is correctly set.

bool **reserveCache**(*CacheChange_t* **change, uint32_t dataCdrSerializedSize)
Reserve a *CacheChange_t*.

Parameters

- **change** – Pointer to pointer to the Cache.
- **dataCdrSerializedSize** – Size of the Cache.

Returns True if correctly reserved.

void **releaseCache**(*CacheChange_t* *change)
Release a cacheChange.

virtual bool **nextUnreadCache**(*CacheChange_t* **change, WriterProxy **wp) = 0
Read the next unread *CacheChange_t* from the history

Parameters

- **change** – Pointer to pointer of *CacheChange_t*
- **wp** – Pointer to pointer to the WriterProxy

Returns True if read.

virtual bool **nextUntakenCache**(*CacheChange_t* **change, WriterProxy **wp) = 0
Get the next *CacheChange_t* from the history to take.

Parameters

- **change** – Pointer to pointer of *CacheChange_t*.
- **wp** – Pointer to pointer to the WriterProxy.

Returns True if read.

inline bool **expectsInlineQos**()

Returns True if the reader expects Inline QOS.

inline *ReaderHistory* ***getHistory**()
Returns a pointer to the associated *History*.

inline eprosima::fastdds::rtps::IReaderDataFilter ***get_content_filter**() const

Returns The content filter associated to this reader.

inline void **set_content_filter**(eprosima::fastdds::rtps::IReaderDataFilter *filter)
Set the content filter associated to this reader.

Parameters **filter** – Pointer to the content filter to associate to this reader.

virtual bool **isInCleanState**() = 0
Returns there is a clean state with all Writers.

It occurs when the Reader received all samples sent by Writers. In other words, its WriterProxies are up to date.

Returns There is a clean state with all Writers.

virtual void **assert_writer_liveliness**(const *GUID_t* &writer) = 0
Assert the liveliness of a matched writer.

Parameters **writer** – GUID of the writer to assert.

virtual bool **begin_sample_access_nts**(*CacheChange_t* *change, WriterProxy *&wp, bool &is_future_change) = 0
Called just before a change is going to be deserialized.

Parameters

- **change** – **[in]** Pointer to the change being accessed.
- **wp** – **[out]** Writer proxy the change belongs to.
- **is_future_change** – **[out]** Whether the change is in the future (i.e. there are earlier unreceived changes from the same writer).

Returns Whether the change is still valid or not.

virtual void **end_sample_access_nts**(*CacheChange_t* *change, WriterProxy *&wp, bool mark_as_read) = 0
Called after the change has been deserialized.

Parameters

- **change** – **[in]** Pointer to the change being accessed.
- **wp** – **[in]** Writer proxy the change belongs to.
- **mark_as_read** – **[in]** Whether the change should be marked as read or not.

virtual void **change_read_by_user**(*CacheChange_t* *change, WriterProxy *writer, bool mark_as_read = true) = 0
Called when the user has retrieved a change from the history.

Parameters

- **change** – Pointer to the change to ACK

- **writer** – Writer proxy of the change.
- **mark_as_read** – Whether the change should be marked as read or not

bool **is_sample_valid**(const void *data, const *GUID_t* &writer, const *SequenceNumber_t* &sn) const
Checks whether the sample is still valid or is corrupted.

Parameters

- **data** – Pointer to the sample data to check. If it does not belong to the payload pool passed to the reader on construction, it yields undefined behavior.
- **writer** – GUID of the writer that sent data.
- **sn** – Sequence number related to data.

Returns true if the sample is valid

Public Members

LivelinessChangedStatus **liveliness_changed_status_**
The liveliness changed status struct as defined in the DDS.

Resources

MemoryManagementPolicy

enum eprosima::fastrtps::rtps::**MemoryManagementPolicy**
Enum MemoryuManagementPolicy_t, indicated the way memory is managed in terms of dealing with CacheChanges

Values:

enumerator **PREALLOCATED_MEMORY_MODE**
Preallocated memory.

Size set to the data type maximum. Largest memory footprint but smallest allocation count.

enumerator **PREALLOCATED_WITH_REALLOC_MEMORY_MODE**
Default size preallocated, requires reallocation when a bigger message arrives.

Smaller memory footprint at the cost of an increased allocation count.

enumerator **DYNAMIC_RESERVE_MEMORY_MODE**

enumerator **DYNAMIC_REUSABLE_MEMORY_MODE**

RTPSDomain

class **RTPSDomain**

Class *RTPSDomain*,it manages the creation and destruction of *RTPSParticipant* *RTPSWriter* and *RTPSReader*. It stores a list of all created *RTPSParticipant*. It has only static methods.

Public Static Functions

static void **set_filewatch_thread_config**(const fastdds::rtps::ThreadSettings &watch_thread, const fastdds::rtps::ThreadSettings &callback_thread)

Method to set the configuration of the threads created by the file watcher for the environment file. In order for these settings to take effect, this method must be called before the first call to *createParticipant*.

Parameters

- **watch_thread** – Settings for the thread watching the environment file.
- **callback_thread** – Settings for the thread executing the callback when the environment file changed.

static void **stopAll**()

Method to shut down all RTPSParticipants, readers, writers, etc. It must be called at the end of the process to avoid memory leaks. It also shut downs the DomainRTPSParticipant.

Post After this call, all the pointers to RTPS entities are invalidated and their use may result in undefined behaviour.

static *RTPSParticipant* ***createParticipant**(uint32_t domain_id, const *RTPSParticipantAttributes* &attrs, *RTPSParticipantListener* *plisten = nullptr)

Create a *RTPSParticipant*.

Warning: The returned pointer is invalidated after a call to *removeRTPSParticipant()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **domain_id** – DomainId to be used by the *RTPSParticipant* (80 by default).
- **attrs** – *RTPSParticipant* Attributes.
- **plisten** – Pointer to the ParticipantListener.

Returns Pointer to the *RTPSParticipant*.

static *RTPSParticipant* ***createParticipant**(uint32_t domain_id, bool enabled, const *RTPSParticipantAttributes* &attrs, *RTPSParticipantListener* *plisten = nullptr)

Create a *RTPSParticipant*.

Warning: The returned pointer is invalidated after a call to *removeRTPSParticipant()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **domain_id** – DomainId to be used by the *RTPSParticipant* (80 by default).

- **enabled** – True if the *RTPSParticipant* should be enabled on creation. False if it will be enabled later with *RTPSParticipant::enable()*
- **attrs** – *RTPSParticipant* Attributes.
- **plisten** – Pointer to the ParticipantListener.

Returns Pointer to the *RTPSParticipant*.

static *RTPSWriter* ***createRTPSWriter**(*RTPSParticipant* *p, *WriterAttributes* &watt, *WriterHistory* *hist, *WriterListener* *listen = nullptr)

Create a *RTPSWriter* in a participant.

Warning: The returned pointer is invalidated after a call to *removeRTPSWriter()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.
- **watt** – Writer Attributes.
- **hist** – Pointer to the *WriterHistory*.
- **listen** – Pointer to the *WriterListener*.

Returns Pointer to the created *RTPSWriter*.

static *RTPSWriter* ***createRTPSWriter**(*RTPSParticipant* *p, *WriterAttributes* &watt, const std::shared_ptr<*IPayloadPool*> &payload_pool, *WriterHistory* *hist, *WriterListener* *listen = nullptr)

Create a *RTPSWriter* in a participant using a custom payload pool.

Warning: The returned pointer is invalidated after a call to *removeRTPSWriter()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.
- **watt** – Writer Attributes.
- **payload_pool** – Shared pointer to the *IPayloadPool*
- **hist** – Pointer to the *WriterHistory*.
- **listen** – Pointer to the *WriterListener*.

Returns Pointer to the created *RTPSWriter*.

static *RTPSWriter* ***createRTPSWriter**(*RTPSParticipant* *p, *WriterAttributes* &watt, const std::shared_ptr<*IPayloadPool*> &payload_pool, const std::shared_ptr<*IChangePool*> &change_pool, *WriterHistory* *hist, *WriterListener* *listen = nullptr)

Create a *RTPSWriter* in a participant using a custom payload pool.

Warning: The returned pointer is invalidated after a call to *removeRTPSWriter()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.
- **watt** – Writer Attributes.
- **payload_pool** – Shared pointer to the *IPayloadPool*
- **change_pool** – Shared pointer to the *IChangePool*
- **hist** – Pointer to the *WriterHistory*.
- **listen** – Pointer to the *WriterListener*.

Returns Pointer to the created *RTPSWriter*.

static *RTPSWriter* ***createRTPSWriter**(*RTPSParticipant* *p, const *EntityId_t* &entity_id, *WriterAttributes* &watt, const std::shared_ptr<*IPayloadPool*> &payload_pool, const std::shared_ptr<*IChangePool*> &change_pool, *WriterHistory* *hist, *WriterListener* *listen = nullptr)

Create a *RTPSWriter* in a participant using a custom payload pool.

Warning: The returned pointer is invalidated after a call to *removeRTPSWriter()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.
- **entity_id** – Specific entity id to use for the created writer.
- **watt** – Writer Attributes.
- **payload_pool** – Shared pointer to the *IPayloadPool*
- **change_pool** – Shared pointer to the *IChangePool*
- **hist** – Pointer to the *WriterHistory*.
- **listen** – Pointer to the *WriterListener*.

Returns Pointer to the created *RTPSWriter*.

static *RTPSWriter* ***createRTPSWriter**(*RTPSParticipant* *p, const *EntityId_t* &entity_id, *WriterAttributes* &watt, const std::shared_ptr<*IPayloadPool*> &payload_pool, *WriterHistory* *hist, *WriterListener* *listen = nullptr)

Create a *RTPSWriter* in a participant.

Warning: The returned pointer is invalidated after a call to *removeRTPSWriter()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.

- **entity_id** – Specific entity id to use for the created writer.
- **watt** – Writer Attributes.
- **payload_pool** – Shared pointer to the *IPayloadPool*
- **hist** – Pointer to the *WriterHistory*.
- **listen** – Pointer to the *WriterListener*.

Returns Pointer to the created *RTPSWriter*.

static bool **removeRTPSWriter**(*RTPSWriter* *writer)
Remove a *RTPSWriter*.

Parameters **writer** – Pointer to the writer you want to remove.

Returns True if correctly removed.

static *RTPSReader* ***createRTPSReader**(*RTPSParticipant* *p, *ReaderAttributes* &ratt, *ReaderHistory* *hist, *ReaderListener* *listen = nullptr)
Create a *RTPSReader* in a participant.

Warning: The returned pointer is invalidated after a call to *removeRTPSReader()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.
- **ratt** – Reader Attributes.
- **hist** – Pointer to the *ReaderHistory*.
- **listen** – Pointer to the *ReaderListener*.

Returns Pointer to the created *RTPSReader*.

static *RTPSReader* ***createRTPSReader**(*RTPSParticipant* *p, *ReaderAttributes* &ratt, const std::shared_ptr<*IPayloadPool*> &payload_pool, *ReaderHistory* *hist, *ReaderListener* *listen = nullptr)
Create a RTPReader in a participant using a custom payload pool.

Warning: The returned pointer is invalidated after a call to *removeRTPSReader()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.
- **ratt** – Reader Attributes.
- **payload_pool** – Shared pointer to the *IPayloadPool*
- **hist** – Pointer to the *ReaderHistory*.
- **listen** – Pointer to the *ReaderListener*.

Returns Pointer to the created *RTPSReader*.

static *RTPSReader* ***createRTPSReader**(*RTPSParticipant* *p, const *EntityId_t* &entity_id, *ReaderAttributes* &ratt, const std::shared_ptr<*IPayloadPool*> &payload_pool, *ReaderHistory* *hist, *ReaderListener* *listen = nullptr)

Create a *RTPSReader* in a participant using a custom payload pool.

Warning: The returned pointer is invalidated after a call to *removeRTPSReader()* or *stopAll()*, so its use may result in undefined behaviour.

Parameters

- **p** – Pointer to the *RTPSParticipant*.
- **entity_id** – Specific entity id to use for the created reader.
- **ratt** – Reader Attributes.
- **payload_pool** – Shared pointer to the *IPayloadPool*
- **hist** – Pointer to the *ReaderHistory*.
- **listen** – Pointer to the *ReaderListener*.

Returns Pointer to the created *RTPSReader*.

static bool **removeRTPSReader**(*RTPSReader* *reader)

Remove a *RTPSReader*.

Parameters **reader** – Pointer to the reader you want to remove.

Returns True if correctly removed.

static bool **removeRTPSParticipant**(*RTPSParticipant* *p)

Remove a *RTPSParticipant* and delete all its associated Writers, Readers, resources, etc.

Parameters **p** – **[in]** Pointer to the *RTPSParticipant*;

Returns True if correct.

RTPSWriter

LivelinessData

struct **LivelinessData**

A struct keeping relevant liveliness information of a writer.

Public Functions

inline **LivelinessData**(*GUID_t* guid_in, LivelinessQosPolicyKind kind_in, *Duration_t* lease_duration_in)

Constructor.

Parameters

- **guid_in** – GUID of the writer
- **kind_in** – Liveliness kind
- **lease_duration_in** – Liveliness lease duration

inline bool **operator==**(const *LivelinessData* &other) const
Equality operator.

Parameters **other** – Liveliness data to compare to

Returns True if equal

inline bool **operator!=**(const *LivelinessData* &other) const
Inequality operator.

Parameters **other** – Liveliness data to compare to

Returns True if different

Public Members

GUID_t **guid**
GUID of the writer.

LivelinessQosPolicyKind **kind**
Writer liveliness kind.

Duration_t **lease_duration**
The lease duration.

unsigned int **count** = 1
The number of times the writer is being counted.

WriterStatus **status**
The writer status.

std::chrono::steady_clock::time_point **time**
The time when the writer will lose liveliness.

RTPSWriter

class **RTPSWriter** : public eprosima::fastrtps::rtps::*Endpoint*, public eprosima::fastdds::statistics::StatisticsWriterImpl
Class *RTPSWriter*, manages the sending of data to the readers. Is always associated with a HistoryCache.

Subclassed by eprosima::fastrtps::rtps::StatefulWriter, eprosima::fastrtps::rtps::StatelessWriter

Public Functions

template<typename **T**>
inline *CacheChange_t* ***new_change**(*T* &data, *ChangeKind_t* changeKind, *InstanceHandle_t* handle = *c_InstanceHandle_Unknown*)
Create a new change based with the provided changeKind.

Parameters

- **data** – Data of the change.
- **changeKind** – The type of change.

- **handle** – InstanceHandle to assign.

Returns Pointer to the CacheChange or nullptr if incorrect.

bool **release_change**(*CacheChange_t* *change)
Release a change when it is not being used anymore.

Parameters **change** – Pointer to the cache change to be released.

Returns whether the operation succeeded or not

Pre

- change is not nullptr
- change points to a cache change obtained from a call to this->new_change

Post memory pointed to by change is not accessed

virtual bool **matched_reader_add**(const *ReaderProxyData* &data) = 0
Add a matched reader.

Parameters **data** – Pointer to the *ReaderProxyData* object added.

Returns True if added.

virtual bool **matched_reader_remove**(const *GUID_t* &reader_guid) = 0
Remove a matched reader.

Parameters **reader_guid** – GUID of the reader to remove.

Returns True if removed.

virtual bool **matched_reader_is_matched**(const *GUID_t* &reader_guid) = 0
Tells us if a specific Reader is matched against this writer.

Parameters **reader_guid** – GUID of the reader to check.

Returns True if it was matched.

virtual void **reader_data_filter**(fastdds::rtps::IReaderDataFilter *filter) = 0
Set a content filter to perform content filtering on this writer.

This method sets a content filter that will be used to check whether a cache change is relevant for a reader or not.

Parameters **filter** – The content filter to use on this writer. May be nullptr to remove the content filter (i.e. treat all samples as relevant).

virtual const fastdds::rtps::IReaderDataFilter ***reader_data_filter**() const = 0
Get the content filter used to perform content filtering on this writer.

Returns The content filter used on this writer.

inline virtual bool **has_been_fully_delivered**(const *SequenceNumber_t* &seq_num) const
Check if a specific change has been delivered to the transport layer of every matched remote *RTPSReader* at least once.

Parameters **seq_num** – Sequence number of the change to check.

Returns true if delivered. False otherwise.

inline virtual bool **is_acked_by_all**(const *CacheChange_t**) const
Check if a specific change has been acknowledged by all Readers. Is only useful in reliable Writer. In BE Writers returns false when pending to be sent.

Returns True if acknowledged by all.

inline virtual bool **wait_for_all_acked**(const *Duration_t*&)
Waits until all changes were acknowledged or max_wait.

Returns True if all were acknowledged.

virtual void **updateAttributes**(const *WriterAttributes* &att) = 0
Update the Attributes of the Writer.

Parameters **att** – New attributes

SequenceNumber_t **get_seq_num_min**()
Get Min Seq Num in *History*.

Returns Minimum sequence number in history

SequenceNumber_t **get_seq_num_max**()
Get Max Seq Num in *History*.

Returns Maximum sequence number in history

uint32_t **getTypeMaxSerialized**()
Get maximum size of the serialized type

Returns Maximum size of the serialized type

uint32_t **getMaxDataSize**()
Get maximum size of the data.

uint32_t **calculateMaxDataSize**(uint32_t length)
Calculates the maximum size of the data.

inline *WriterListener* ***getListener**()
Get listener

Returns Listener

inline bool **isAsync**() const
Get the publication mode

Returns publication mode

bool **remove_older_changes**(unsigned int max = 0)
Remove an specified max number of changes

Parameters **max** – Maximum number of changes to remove.

Returns at least one change has been removed

inline virtual bool **get_disable_positive_acks**() const
Returns if disable positive ACKs QoS is enabled.

Returns Best effort writers always return false. Reliable writers override this method.

virtual bool **try_remove_change**(const std::chrono::steady_clock::time_point &max_blocking_time_point,
std::unique_lock<RecursiveTimedMutex> &lock) = 0
Tries to remove a change waiting a maximum of the provided microseconds.

Parameters

- **max_blocking_time_point** – Maximum time to wait for.
- **lock** – Lock of the Change list.

Returns at least one change has been removed

virtual bool **wait_for_acknowledgement**(const *SequenceNumber_t* &seq, const std::chrono::steady_clock::time_point &max_blocking_time_point, std::unique_lock<RecursiveTimedMutex> &lock) = 0

Waits till a change has been acknowledged.

Parameters

- **seq** – Sequence number to wait for acknowledgement.
- **max_blocking_time_point** – Maximum time to wait for.
- **lock** – Lock of the Change list.

Returns true when change was acknowledged, false when timeout is reached.

inline RTPSParticipantImpl ***getRTPSParticipant**() const

Get RTPS participant

Returns RTPS participant

inline void **set_separate_sending**(bool enable)

Enable or disable sending data to readers separately NOTE: This will only work for synchronous writers

Parameters **enable** – If separate sending should be enabled

inline bool **get_separate_sending**() const

Inform if data is sent to readers separately

Returns true if separate sending is enabled

inline virtual bool **process_acknack**(const *GUID_t* &writer_guid, const *GUID_t* &reader_guid, uint32_t ack_count, const *SequenceNumberSet_t* &sn_set, bool final_flag, bool &result, fastdds::rtps::*VendorId_t* origin_vendor_id = c_VendorId_Unknown)

Process an incoming ACKNACK submessage.

Parameters

- **writer_guid** – **[in]** GUID of the writer the submessage is directed to.
- **reader_guid** – **[in]** GUID of the reader originating the submessage.
- **ack_count** – **[in]** Count field of the submessage.
- **sn_set** – **[in]** Sequence number bitmap field of the submessage.
- **final_flag** – **[in]** Final flag field of the submessage.
- **result** – **[out]** true if the writer could process the submessage. Only valid when returned value is true.
- **origin_vendor_id** – **[in]** VendorId of the source participant from which the message was received

Returns true when the submessage was destinated to this writer, false otherwise.

inline virtual bool **process_nack_frag**(const *GUID_t* &writer_guid, const *GUID_t* &reader_guid, uint32_t ack_count, const *SequenceNumber_t* &seq_num, const *FragmentNumberSet_t* fragments_state, bool &result, fastdds::rtps::*VendorId_t* origin_vendor_id = c_VendorId_Unknown)

Process an incoming NACKFRAG submessage.

Parameters

- **writer_guid** – **[in]** GUID of the writer the submessage is directed to.
- **reader_guid** – **[in]** GUID of the reader originating the submessage.
- **ack_count** – **[in]** Count field of the submessage.
- **seq_num** – **[in]** Sequence number field of the submessage.
- **fragments_state** – **[in]** Fragment number bitmap field of the submessage.
- **result** – **[out]** true if the writer could process the submessage. Only valid when returned value is true.
- **origin_vendor_id** – **[in]** VendorId of the source participant from which the message was received

Returns true when the submessage was destinated to this writer, false otherwise.

const LivelinessQosPolicyKind &**get_liveliness_kind**() const
A method to retrieve the liveliness kind.

Returns Liveliness kind

const *Duration_t* &**get_liveliness_lease_duration**() const
A method to retrieve the liveliness lease duration.

Returns Lease duration

const *Duration_t* &**get_liveliness_announcement_period**() const
A method to return the liveliness announcement period.

Returns The announcement period

bool **is_datasharing_compatible**() const

Returns Whether the writer is data sharing compatible or not

virtual DeliveryRetCode **deliver_sample_nts**(*CacheChange_t* *cache_change, RTPSMessageGroup &group, LocatorSelectorSender &locator_selector, const std::chrono::time_point<std::chrono::steady_clock> &max_blocking_time) = 0
Tells writer the sample can be sent to the network.

This function should be used by a fastdds::rtps::FlowController.

Note: Must be non-thread safe.

Parameters

- **cache_change** – Pointer to the *CacheChange_t* that represents the sample which can be sent.
- **group** – RTPSMessageGroup reference uses for generating the RTPS message.
- **locator_selector** – RTPSMessageSenderInterface reference uses for selecting locators. The reference has to be a member of this *RTPSWriter* object.
- **max_blocking_time** – Future timepoint where blocking send should end.

Returns Return code.

virtual bool **send_nts**(*CDRMessage_t* *message, const LocatorSelectorSender &locator_selector, std::chrono::steady_clock::time_point &max_blocking_time_point) const

Send a message through this interface.

Parameters

- **message** – Pointer to the buffer with the message already serialized.
- **locator_selector** – RTPSMessageSenderInterface reference uses for selecting locators. The reference has to be a member of this *RTPSWriter* object.
- **max_blocking_time_point** – Future timepoint where blocking send should end.

Public Members

LivelinessLostStatus **liveliness_lost_status_**

Liveliness lost status of this writer.

WriterListener

class **WriterListener**

Class *WriterListener* with virtual method so the user can implement callbacks to certain events.

Subclassed by eprosima::fastdds::dds::builtin::TypeLookupReplyListener, eprosima::fastdds::dds::builtin::TypeLookupRequestLis

Public Functions

inline virtual void **onWriterMatched**(*RTPSWriter* *writer, *MatchingInfo* &info)

This method is called when a new Reader is matched with this Writer by the builtin protocols

Parameters

- **writer** – Pointer to the *RTPSWriter*.
- **info** – Matching Information.

inline virtual void **onWriterMatched**(*RTPSWriter* *writer, const eprosima::fastdds::dds::*PublicationMatchedStatus* &info)

This method is called when a new Reader is matched with this Writer by the builtin protocols

Parameters

- **writer** – Pointer to the *RTPSWriter*.
- **info** – Publication matching information.

inline virtual void **on_offered_incompatible_qos**(*RTPSWriter* *writer, eprosima::fastdds::dds::PolicyMask qos)

This method is called when a new Reader is discovered, with a Topic that matches that of a local writer, but with a requested QoS that is incompatible with the one offered by the local writer

Parameters

- **writer** – Pointer to the *RTPSWriter*.
- **qos** – A mask with the bits of all incompatible Qos activated.

inline virtual void **onWriterChangeReceivedByAll**(*RTPSWriter* *writer, *CacheChange_t* *change)
This method is called when all the readers matched with this Writer acknowledge that a cache change has been received.

Parameters

- **writer** – Pointer to the *RTPSWriter*.
- **change** – Pointer to the affected *CacheChange_t*.

inline virtual void **on_liveliness_lost**(*RTPSWriter* *writer, const LivelinessLostStatus &status)
Method called when the liveliness of a writer is lost.

Parameters

- **writer** – The writer
- **status** – The liveliness lost status

inline virtual void **on_reader_discovery**(*RTPSWriter* *writer, *ReaderDiscoveryInfo*::*DISCOVERY_STATUS* reason, const *GUID_t* &reader_guid, const *ReaderProxyData* *reader_info)
Method called when the discovery information of a reader regarding a writer changes.

Parameters

- **writer** – The writer.
- **reason** – The reason motivating this method to be called.
- **reader_guid** – The GUID of the reader for which the discovery information changed.
- **reader_info** – Discovery information about the reader. Will be `nullptr` for reason `REMOVED_READER`.

inline virtual void **on_incompatible_type**(*RTPSWriter* *writer)
This method is called when a new Reader is discovered, with a Topic that matches the name of a local writer, but with an incompatible type

Parameters **writer** – Pointer to the *RTPSWriter*.

6.32.3 Transport

eProsima Fast DDS Transport Layer API.

Transport Generic Interfaces

TransportDescriptorInterface

struct **TransportDescriptorInterface**
Virtual base class for the data type used to define transport configuration. It acts as a builder for a given transport meaning that it allows to configure the transport, and then a new Transport can be built according to this configuration using its *create_transport()* factory member function.

- maxMessageSize: maximum size of a single message in the transport.
- maxInitialPeersRange: number of channels opened with each initial remote peer.

Subclassed by *eprosima::fastdds::rtps::ChainingTransportDescriptor*, eprosima::fastdds::rtps::PortBasedTransportDescriptor

Public Functions

inline **TransportDescriptorInterface**(uint32_t maximumMessageSize, uint32_t maximumInitialPeersRange)
Constructor.

inline **TransportDescriptorInterface**(const *TransportDescriptorInterface* &t)
Copy constructor.

inline *TransportDescriptorInterface* &**operator=**(const *TransportDescriptorInterface* &t)
Copy assignment.

virtual **~TransportDescriptorInterface**() = default
Destructor.

virtual *TransportInterface* ***create_transport**() const = 0
Factory method pattern. It will create and return a *TransportInterface* corresponding to this descriptor. This provides an interface to the NetworkFactory to create the transports without the need to know about their type

virtual uint32_t **min_send_buffer_size**() const = 0
Returns the minimum size required for a send operation.

inline virtual uint32_t **max_message_size**() const
Returns the maximum size expected for received messages.

inline virtual uint32_t **max_initial_peers_range**() const
Returns the maximum number of opened channels for each initial remote peer (maximum number of guessed initial peers to try to connect)

inline bool **operator==**(const *TransportDescriptorInterface* &t) const
Comparison operator.

inline void **lock**()
Lock internal mutex (for Fast-DDS internal use)

inline void **unlock**()
Unlock internal mutex (for Fast-DDS internal use)

Public Members

uint32_t **maxMessageSize**
Maximum size of a single message in the transport.

uint32_t **maxInitialPeersRange**
Number of channels opened with each initial remote peer.

TransportInterface

class **TransportInterface**

Interface against which to implement a transport layer, decoupled from FastRTPS internals. *TransportInterface* expects the user to implement a logical equivalence between *Locators* and protocol-specific "channels". This equivalence can be narrowing: For instance in UDP/IP, a port would take the role of channel, and several different locators can map to the same port, and hence the same channel.

Subclassed by *eprosima::fastdds::rtps::ChainingTransport*

Public Functions

virtual **~TransportInterface**() = default

Aside from the API defined here, an user-defined Transport must define a descriptor data type and a constructor that expects a constant reference to such descriptor. e.g:

class MyTransport: public: typedef struct { ... } MyTransportDescriptor; MyTransport(const MyTransportDescriptor&); ...

TransportInterface(const *TransportInterface* &t) = delete

Copy constructor.

TransportInterface &**operator=**(const *TransportInterface* &t) = delete

Copy assignment.

TransportInterface(*TransportInterface* &&t) = delete

Move constructor.

TransportInterface &**operator=**(*TransportInterface* &&t) = delete

Move assignment.

virtual bool **init**(const fastrtps::rtps::*PropertyPolicy* *properties = nullptr, const uint32_t &max_msg_size_no_frag = 0) = 0

Initialize this transport. This method will prepare all the internals of the transport.

Parameters

- **properties** – Optional policy to specify additional parameters of the created transport.
- **max_msg_size_no_frag** – Optional maximum message size to avoid 65500 KB fragmentation limit.

Returns True when the transport was correctly initialized.

virtual bool **IsInputChannelOpen**(const Locator&) const = 0

Must report whether the input channel associated to this locator is open. Channels must either be fully closed or fully open, so that "open" and "close" operations are whole and definitive.

virtual bool **IsLocatorSupported**(const Locator&) const = 0

Must report whether the given locator is supported by this transport (typically inspecting its "kind" value).

virtual bool **is_locator_allowed**(const Locator&) const = 0

Must report whether the given locator is allowed by this transport.

virtual Locator **RemoteToMainLocal**(const Locator &remote) const = 0

Returns the locator describing the main (most general) channel that can write to the provided remote locator.

inline virtual bool **transform_remote_locator**(const Locator &remote_locator, Locator &result_locator) const

Transforms a remote locator into a locator optimized for local communications.

If the remote locator corresponds to one of the local interfaces, it is converted to the corresponding local address.

Parameters

- **remote_locator** – **[in]** Locator to be converted.
- **result_locator** – **[out]** Converted locator.

Returns false if the input locator is not supported/allowed by this transport, true otherwise.

virtual bool **OpenOutputChannel**(*SendResourceList* &sender_resource_list, const Locator&) = 0

Must open the channel that maps to/from the given locator. This method must allocate, reserve and mark any resources that are needed for said channel.

virtual bool **OpenOutputChannels**(*SendResourceList* &sender_resource_list, const fastrtps::rtps::*LocatorSelectorEntry* &locator_selector_entry)

Must open the channel that maps to/from the given locator selector entry. This method must allocate, reserve and mark any resources that are needed for said channel.

Parameters

- **sender_resource_list** – Participant's send resource list.
- **locator_selector_entry** – Locator selector entry with the remote entity locators.

Returns true if the channel was correctly opened or if finding an already opened one.

virtual void **CloseOutputChannels**(*SendResourceList* &sender_resource_list, const fastrtps::rtps::*LocatorSelectorEntry* &locator_selector_entry)

Close the channel that maps to/from the given locator selector entry.

Parameters

- **sender_resource_list** – Participant's send resource list.
- **locator_selector_entry** – Locator selector entry with the remote entity locators.

virtual bool **OpenInputChannel**(const Locator&, *TransportReceiverInterface**, uint32_t) = 0

Opens an input channel to receive incoming connections. If there is an existing channel it registers the receiver interface.

virtual bool **CloseInputChannel**(const Locator&) = 0

Must close the channel that maps to/from the given locator. IMPORTANT: It MUST be safe to call this method even during a Receive operation on another thread. You must implement any necessary mutual exclusion and timeout mechanisms to make sure the channel can be closed without damage.

virtual bool **DoInputLocatorsMatch**(const Locator&, const Locator&) const = 0

Must report whether two locators map to the same internal channel.

virtual *LocatorList* **NormalizeLocator**(const Locator &locator) = 0

Performs locator normalization (assign valid IP if not defined by user)

virtual void **select_locators**(fastrtps::rtps::*LocatorSelector* &selector) const = 0

Performs the locator selection algorithm for this transport.

It basically consists of the following steps

- selector.transport_starts is called
- transport handles the selection state of each locator

- if a locator from an entry is selected, selector.select is called for that entry

Parameters `selector` – **[inout]** Locator selector.

virtual bool **is_local_locator**(const Locator &locator) const = 0
Must report whether the given locator is from the local host.

virtual *TransportDescriptorInterface* ***get_configuration**() = 0
Return the transport configuration (Transport Descriptor)

virtual void **AddDefaultOutputLocator**(*LocatorList* &defaultList) = 0
Add default output locator to the locator list.

virtual bool **getDefaultMetatrafficMulticastLocators**(*LocatorList* &locators, uint32_t metatraffic_multicast_port) const = 0
Add metatraffic multicast locator with the given port.

virtual bool **getDefaultMetatrafficUnicastLocators**(*LocatorList* &locators, uint32_t metatraffic_unicast_port) const = 0
Add metatraffic unicast locator with the given port.

virtual bool **getDefaultUnicastLocators**(*LocatorList* &locators, uint32_t unicast_port) const = 0
Add unicast locator with the given port.

virtual bool **fillMetatrafficMulticastLocator**(Locator &locator, uint32_t metatraffic_multicast_port) const = 0
Assign port to the given metatraffic multicast locator if not already defined.

virtual bool **fillMetatrafficUnicastLocator**(Locator &locator, uint32_t metatraffic_unicast_port) const = 0
Assign port to the given metatraffic unicast locator if not already defined.

virtual bool **configureInitialPeerLocator**(Locator &locator, const fastrtps::rtps::*PortParameters* &port_params, uint32_t domainId, *LocatorList* &list) const = 0
Configure the initial peer locators list.

virtual bool **fillUnicastLocator**(Locator &locator, uint32_t well_known_port) const = 0
Assign port to the given unicast locator if not already defined.

virtual uint32_t **max_recv_buffer_size**() const = 0

Returns The maximum datagram size for reception supported by the transport

inline virtual void **shutdown**()
Shutdown method to close the connections of the transports.

inline virtual void **update_network_interfaces**()
Update network interfaces.

inline int32_t **kind**() const
Return the transport kind.

inline virtual bool **transform_remote_locator**(const Locator &remote_locator, Locator &result_locator, bool allowed_remote_localhost, bool allowed_local_localhost) const
Transforms a remote locator into a locator optimized for local communications.

If the remote locator corresponds to one of the local interfaces, it is converted to the corresponding local address if allowed by both local and remote transports.

Parameters

- **remote_locator** – **[in]** Locator to be converted.
- **result_locator** – **[out]** Converted locator.
- **allowed_remote_localhost** – **[in]** Whether localhost is allowed (and hence used) in the remote transport.
- **allowed_local_localhost** – **[in]** Whether localhost is allowed locally (by this or other transport).

Returns false if the input locator is not supported/allowed by this transport, true otherwise.

inline virtual bool **is_localhost_allowed**() const
Must report whether localhost locator is allowed.

inline virtual NetmaskFilterInfo **netmask_filter_info**() const
Returns netmask filter information (transport's netmask filter kind and allowlist)

constexpr uint32_t eprosima::fastdds::rtps::**s_maximumMessageSize** = 65500
Default maximum message size.

constexpr uint32_t eprosima::fastdds::rtps::**s_maximumInitialPeersRange** = 4
Default maximum initial peers range.

constexpr uint32_t eprosima::fastdds::rtps::**s_minimumSocketBuffer** = 65536
Default minimum socket buffer.

static const std::string eprosima::fastdds::rtps::**s_IPv4AddressAny** = "0.0.0.0"
Default IPv4 address.

static const std::string eprosima::fastdds::rtps::**s_IPv6AddressAny** = "::"
Default IPv6 address.

using eprosima::fastdds::rtps::**SendResourceList** =
std::vector<std::unique_ptr<fastrtps::rtps::SenderResource>>

TransportReceiverInterface

class **TransportReceiverInterface**
Interface against which to implement a data receiver, decoupled from transport internals.

Public Functions

virtual **~TransportReceiverInterface**() = default
Destructor.

virtual void **OnDataReceived**(const fastrtps::rtps::*octet* *data, const uint32_t size, const Locator &local_locator, const Locator &remote_locator) = 0
Method to be called by the transport when receiving data.

Parameters

- **data** – Pointer to the received data.

- **size** – Number of bytes received.
- **local_locator** – Locator identifying the local endpoint.
- **remote_locator** – Locator identifying the remote endpoint.

SocketTransportDescriptor

struct **SocketTransportDescriptor** : public eprosima::fastdds::rtps::PortBasedTransportDescriptor
Virtual base class for the data type used to define configuration of transports using sockets.

- `sendBufferSize:` size of the sending buffer of the socket (in octets).
- `receiveBufferSize:` size of the receiving buffer of the socket (in octets).
- `interfaceWhiteList:` list of allowed interfaces.
- `TTL:` time to live, in number of hops.

Subclassed by *eprosima::fastdds::rtps::TCPTransportDescriptor*, *eprosima::fastdds::rtps::test_UDPv4TransportDescriptor*, *eprosima::fastdds::rtps::UDPTransportDescriptor*

Public Functions

inline **SocketTransportDescriptor**(uint32_t maximumMessageSize, uint32_t maximumInitialPeersRange)
Constructor.

SocketTransportDescriptor(const *SocketTransportDescriptor* &t) = default
Copy constructor.

SocketTransportDescriptor &**operator=**(const *SocketTransportDescriptor* &t) = default
Copy assignment.

virtual **~SocketTransportDescriptor**() = default
Destructor.

inline virtual uint32_t **min_send_buffer_size**() const override
Returns the minimum size required for a send operation.

inline bool **operator==**(const *SocketTransportDescriptor* &t) const
Comparison operator.

Public Members

uint32_t **sendBufferSize**
Length of the send buffer.

uint32_t **receiveBufferSize**
Length of the receive buffer.

std::vector<std::string> **interfaceWhiteList**
Allowed interfaces in an IP or device name string format.

NetmaskFilterKind **netmask_filter**
Transport's netmask filter configuration.

std::vector<AllowedNetworkInterface> **interface_allowlist**
Allowed interfaces in an IP or device name string format, each with a specific netmask filter configuration.

std::vector<BlockedNetworkInterface> **interface_blocklist**
Blocked interfaces in an IP or device name string format.

uint8_t **TTL**
Specified time to live (8bit - 255 max TTL)

constexpr uint8_t `eprosima::fastdds::rtps::`**s_defaultTTL** = 1
Default time to live (TTL)

Chaining of transports

ChainingTransportDescriptor

struct **ChainingTransportDescriptor** : public eprosima::fastdds::rtps::*TransportDescriptorInterface*
Base class for the descriptors of chaining transports. A chaining transport allows for the manipulation of data before sending or after receiving from another transport.

Transport configuration:

- low_level_descriptor: Descriptor for lower level transport.

Public Functions

inline virtual uint32_t **min_send_buffer_size**() const override
Returns the minimum size required for a send operation.

inline virtual uint32_t **max_message_size**() const override
Returns the maximum size expected for received messages.

Public Members

std::shared_ptr<*TransportDescriptorInterface*> **low_level_descriptor**
Descriptor for lower level transport.

ChainingTransport

class **ChainingTransport** : public eprosima::fastdds::rtps::*TransportInterface*
This is the base class for chaining adapter transports.

- Directly proxies all operations except Send and Receive
- Has a pointer to the low level transport

Public Functions

inline **ChainingTransport**(const *ChainingTransportDescriptor* &t)
Constructor.

virtual **~ChainingTransport**() = default
Destructor.

inline virtual bool **init**(const fastrtps::rtps::*PropertyPolicy* *properties = nullptr, const uint32_t &max_msg_size_no_frag = 0) override
Initialize the low-level transport.

This method will prepare all the internals of the transport.

Parameters

- **properties** – Optional policy to specify additional parameters of the created transport.
- **max_msg_size_no_frag** – Optional maximum message size to avoid 65500 KB fragmentation limit.

Returns True when the transport was correctly initialized.

inline virtual bool **IsInputChannelOpen**(const fastrtps::rtps::*Locator_t* &loc) const override
Call the low-level transport *IsInputChannelOpen()*.

Must report whether the input channel associated to this locator is open. Channels must either be fully closed or fully open, so that "open" and "close" operations are whole and definitive.

inline virtual bool **IsLocatorSupported**(const fastrtps::rtps::*Locator_t* &loc) const override
Call the low-level transport *IsLocatorSupported()*.

Must report whether the given locator is supported by this transport (typically inspecting its "kind" value).

inline virtual fastrtps::rtps::*Locator_t* **RemoteToMainLocal**(const fastrtps::rtps::*Locator_t* &loc) const override
Call the low-level transport *RemoteToMainLocal()*.

Returns the locator describing the main (most general) channel that can write to the provided remote locator.

virtual bool **OpenInputChannel**(const fastrtps::rtps::*Locator_t* &loc, *TransportReceiverInterface* *receiver_interface, uint32_t max_message_size) override
Call the low-level transport *OpenInputChannel()*.

Opens an input channel to receive incoming connections. If there is an existing channel it registers the receiver interface.

virtual bool **OpenOutputChannel**(*SendResourceList* &sender_resource_list, const fastrtps::rtps::*Locator_t* &loc) override
Call the low-level transport *OpenOutputChannel()*.

Must open the channel that maps to/from the given locator. This method must allocate, reserve and mark any resources that are needed for said channel.

inline virtual bool **CloseInputChannel**(const fastrtps::rtps::*Locator_t* &loc) override
Call the low-level transport *CloseInputChannel()*.

Must close the channel that maps to/from the given locator. IMPORTANT: It MUST be safe to call this method even during a Receive operation on another thread. You must implement any necessary mutual exclusion and timeout mechanisms to make sure the channel can be closed without damage.

inline virtual fastrtps::rtps::LocatorList_t **NormalizeLocator**(const fastrtps::rtps::*Locator_t* &locator) override
Call the low-level transport *NormalizeLocator()*.

Performs locator normalization (assign valid IP if not defined by user)

inline virtual bool **is_local_locator**(const fastrtps::rtps::*Locator_t* &locator) const override
Call the low-level transport *is_local_locator()*.

Must report whether the given locator is from the local host

inline virtual bool **is_localhost_allowed**() const override
Call the low-level transport *is_localhost_allowed()*.

Must report whether localhost locator is allowed

inline virtual NetmaskFilterInfo **netmask_filter_info**() const override
Call the low-level transport *netmask_filter_info()*.

Returns netmask filter information (transport's netmask filter kind and allowlist)

inline virtual bool **DoInputLocatorsMatch**(const fastrtps::rtps::*Locator_t* &locator_1, const fastrtps::rtps::*Locator_t* &locator_2) const override
Call the low-level transport *DoInputLocatorsMatch()*.

Must report whether two locators map to the same internal channel.

inline virtual void **select_locators**(fastrtps::rtps::*LocatorSelector* &selector) const override
Call the low-level transport *select_locators()*.

Performs the locator selection algorithm for this transport.

inline virtual void **AddDefaultOutputLocator**(fastrtps::rtps::LocatorList_t &defaultList) override
Call the low-level transport *AddDefaultOutputLocator()*.

Add default output locator to the locator list

inline virtual bool **getDefaultMetatrafficMulticastLocators**(fastrtps::rtps::LocatorList_t &locators, uint32_t metatraffic_multicast_port) const override
Call the low-level transport *getDefaultMetatrafficMulticastLocators()*.

Add metatraffic multicast locator with the given port

inline virtual bool **getDefaultMetatrafficUnicastLocators**(fastrtps::rtps::LocatorList_t &locators, uint32_t metatraffic_unicast_port) const override
Call the low-level transport *getDefaultMetatrafficUnicastLocators()*.

Add metatraffic unicast locator with the given port

inline virtual bool **getDefaultUnicastLocators**(fastrtps::rtps::LocatorList_t &locators, uint32_t unicast_port) const override
Call the low-level transport *getDefaultUnicastLocators()*.

Add unicast locator with the given port

inline virtual bool **fillMetatrafficMulticastLocator**(fastrtps::rtps::*Locator_t* &locator, uint32_t metatraffic_multicast_port) const override

Call the low-level transport *fillMetatrafficMulticastLocator()*.

Assign port to the given metatraffic multicast locator if not already defined

inline virtual bool **fillMetatrafficUnicastLocator**(fastrtps::rtps::*Locator_t* &locator, uint32_t metatraffic_unicast_port) const override

Call the low-level transport *fillMetatrafficUnicastLocator()*.

Assign port to the given metatraffic unicast locator if not already defined

inline virtual bool **configureInitialPeerLocator**(fastrtps::rtps::*Locator_t* &locator, const fastrtps::rtps::*PortParameters* &port_params, uint32_t domainId, fastrtps::rtps::LocatorList_t &list) const override

Call the low-level transport *configureInitialPeerLocator()*.

Configure the initial peer locators list

inline virtual bool **fillUnicastLocator**(fastrtps::rtps::*Locator_t* &locator, uint32_t well_known_port) const override

Call the low-level transport *fillUnicastLocator()*.

Assign port to the given unicast locator if not already defined

inline virtual bool **transform_remote_locator**(const fastrtps::rtps::*Locator_t* &remote_locator, fastrtps::rtps::*Locator_t* &result_locator) const override

Call the low-level transport *transform_remote_locator()*. Transforms a remote locator into a locator optimized for local communications.

inline virtual uint32_t **max_recv_buffer_size**() const override

Call the low-level transport *max_recv_buffer_size()*.

Returns The maximum datagram size for reception supported by the transport

virtual bool **send**(fastrtps::rtps::SenderResource *low_sender_resource, const fastrtps::rtps::*octet* *send_buffer, uint32_t send_buffer_size, fastrtps::rtps::LocatorsIterator *destination_locators_begin, fastrtps::rtps::LocatorsIterator *destination_locators_end, const std::chrono::steady_clock::time_point &timeout) = 0

Blocking Send through the specified channel. It may perform operations on the output buffer. At the end the function must call to the low-level transport's *send()* function.

```
// Example of calling the low-level transport `send()` function.
return low_sender_resource->send(send_buffer, send_buffer_size, destination_
↪locators_begin,
            destination_locators_end, timeout);
```

Parameters

- **low_sender_resource** – SenderResource generated by the lower transport.
- **send_buffer** – Slice into the raw data to send.
- **send_buffer_size** – Size of the raw data. It will be used as a bounds check for the previous argument. It must not exceed the `sendBufferSize` fed to this class during construction.
- **destination_locators_begin** – First iterator of the list of *Locators* describing the remote destinations we're sending to.

- **destination_locators_end** – End iterator of the list of *Locators* describing the remote destinations we're sending to.
- **timeout** – Maximum blocking time.

virtual void **receive**(*TransportReceiverInterface* *next_receiver, const fastrtps::rtps::*octet* *receive_buffer, uint32_t receive_buffer_size, const fastrtps::rtps::*Locator_t* &local_locator, const fastrtps::rtps::*Locator_t* &remote_locator) = 0

Blocking Receive from the specified channel.

It may perform operations on the input buffer. At the end the function must call to the `next_receiver`'s `OnDataReceived` function.

```
// Example of calling the `next_receiver`'s `OnDataReceived` function.
next_receiver->OnDataReceived(receive_buffer, receive_buffer_size, local_
↪locator, remote_locator);
```

Parameters

- **next_receiver** – Next resource receiver to be called.
- **receive_buffer** – vector with enough capacity (not size) to accommodate a full receive buffer. That capacity must not be less than the `receiveBufferSize` supplied to this class during construction.
- **receive_buffer_size** – Size of the raw data. It will be used as bounds check for the previous argument. It must not exceed the `receiveBufferSize` fed to this class during construction.
- **local_locator** – Locator mapping to the local channel we're listening to.
- **remote_locator** – **[out]** Locator describing the remote destination we received a packet from.

inline virtual void **update_network_interfaces**() override

Update network interfaces.

inline virtual bool **transform_remote_locator**(const fastrtps::rtps::*Locator_t* &remote_locator, fastrtps::rtps::*Locator_t* &result_locator, bool allowed_remote_localhost, bool allowed_local_localhost) const override

Call the low-level transport *transform_remote_locator()*. Transforms a remote locator into a locator optimized for local communications, if allowed by both local and remote transports.

inline virtual bool **is_locator_allowed**(const fastrtps::rtps::*Locator_t* &locator) const override

Call the low-level transport *is_locator_allowed()*.

Must report whether the given locator is allowed by this transport.

UDP Transport

UDPTransportDescriptor

struct **UDPTransportDescriptor** : public eprosima::fastdds::rtps::*SocketTransportDescriptor*
UDP Transport configuration

- `m_output_udp_socket:` source port to use for outgoing datagrams.
- `non_blocking_send:` do not block on send operations. When it is set to true, send operations will return immediately if the buffer is full, but no error will be returned to the upper layer. This means that the application will behave as if the datagram is sent and lost.

Subclassed by *eprosima::fastdds::rtps::UDPv4TransportDescriptor*, *eprosima::fastdds::rtps::UDPv6TransportDescriptor*

Public Functions

virtual **~UDPTransportDescriptor**() = default
Destructor.

UDPTransportDescriptor()
Constructor.

UDPTransportDescriptor(const *UDPTransportDescriptor* &t) = default
Copy constructor.

UDPTransportDescriptor &**operator=**(const *UDPTransportDescriptor* &t) = default
Copy assignment.

bool **operator==**(const *UDPTransportDescriptor* &t) const
Comparison operator.

Public Members

uint16_t **m_output_udp_socket**
Source port to use for outgoing datagrams.

bool **non_blocking_send** = false
Whether to use non-blocking calls to send_to().

When set to true, calls to send_to() will return immediately if the buffer is full, but no error will be returned to the upper layer. This means that the application will behave as if the datagram is sent but lost (i.e. throughput may be reduced). This value is specially useful on high-frequency best-effort writers.

When set to false, calls to send_to() will block until the network buffer has space for the datagram. This may hinder performance on high-frequency writers.

UDPv4TransportDescriptor

struct **UDPv4TransportDescriptor** : public eprosima::fastdds::rtps::*UDPTransportDescriptor*

UDPv4 Transport configuration The kind value for *UDPv4TransportDescriptor* is given by `eprosima::fastrtps::rtps::LOCATOR_KIND_UDPv4`.

Public Functions

virtual **~UDPv4TransportDescriptor**() = default

Destructor.

virtual *TransportInterface* ***create_transport**() const override

Factory method pattern. It will create and return a *TransportInterface* corresponding to this descriptor. This provides an interface to the NetworkFactory to create the transports without the need to know about their type

UDPv4TransportDescriptor()

Constructor.

UDPv4TransportDescriptor(const *UDPv4TransportDescriptor* &t) = default

Copy constructor.

UDPv4TransportDescriptor &**operator=**(const *UDPv4TransportDescriptor* &t) = default

Copy assignment.

UDPv6TransportDescriptor

struct **UDPv6TransportDescriptor** : public eprosima::fastdds::rtps::*UDPTransportDescriptor*

UDPv6 Transport configuration The kind value for *UDPv6TransportDescriptor* is given by `eprosima::fastrtps::rtps::LOCATOR_KIND_UDPv6`.

Public Functions

virtual **~UDPv6TransportDescriptor**() = default

Destructor.

virtual *TransportInterface* ***create_transport**() const override

Factory method pattern. It will create and return a *TransportInterface* corresponding to this descriptor. This provides an interface to the NetworkFactory to create the transports without the need to know about their type

UDPv6TransportDescriptor()

Constructor.

UDPv6TransportDescriptor(const *UDPv6TransportDescriptor* &t) = default

Copy constructor.

UDPv6TransportDescriptor &**operator=**(const *UDPv6TransportDescriptor* &t) = default

Copy assignment.

test_UDPv4TransportDescriptor

struct **test_UDPv4TransportDescriptor** : public eprosima::fastdds::rtps::*SocketTransportDescriptor*
UDP v4 Test Transport configuration

Public Types

typedef std::function<bool(fastrtps::rtps::*CDRMessage_t* &msg)> **filter**
Custom message filtering functions.

typedef std::function<bool(const Locator &destination)> **DestinationLocatorFilter**
Locator filtering function.

Public Functions

test_UDPv4TransportDescriptor()
Constructor.

virtual **~test_UDPv4TransportDescriptor**() = default
Destructor.

virtual *TransportInterface* ***create_transport**() const override
Create transport using the parameters defined within the Descriptor.

test_UDPv4TransportDescriptor(const *test_UDPv4TransportDescriptor* &t) = delete
Copy constructor.

test_UDPv4TransportDescriptor &**operator=**(const *test_UDPv4TransportDescriptor* &t) = delete
Copy assignment.

test_UDPv4TransportDescriptor(*test_UDPv4TransportDescriptor* &&t) = delete
Move constructor.

test_UDPv4TransportDescriptor &**operator=**(*test_UDPv4TransportDescriptor* &&t) = delete
Move assignment.

bool **operator==**(const *test_UDPv4TransportDescriptor* &t) const
Comparison operator.

Public Members

mutable std::atomic<uint8_t> **dropDataMessagesPercentage**
Test shim parameters Percentage of data messages being dropped

filter **drop_data_messages_filter_**
Filtering function for dropping data messages.

bool **dropParticipantBuiltinTopicData**
Flag to enable dropping of discovery Participant DATA(P) messages.

bool **dropPublicationBuiltinTopicData**
Flag to enable dropping of discovery Writer DATA(W) messages.

bool **dropSubscriptionBuiltinTopicData**
Flag to enable dropping of discovery Reader DATA(R) messages.

mutable std::atomic<uint8_t> **dropDataFragMessagesPercentage**
Percentage of data fragments being dropped.

filter **drop_data_frag_messages_filter_**
Filtering function for dropping data fragments messages.

mutable std::atomic<uint8_t> **dropHeartbeatMessagesPercentage**
Percentage of heartbeats being dropped.

filter **drop_heartbeat_messages_filter_**
Filtering function for dropping heartbeat messages.

mutable std::atomic<uint8_t> **dropAckNackMessagesPercentage**
Percentage of AckNacks being dropped.

filter **drop_ack_nack_messages_filter_**
Filtering function for dropping AckNacks.

mutable std::atomic<uint8_t> **dropGapMessagesPercentage**
Percentage of gap messages being dropped.

filter **drop_gap_messages_filter_**
Filtering function for dropping gap messages.

DestinationLocatorFilter **locator_filter_**
Filtering function for dropping messages to specific destinations.

std::vector<fastrtps::rtps::*SequenceNumber_t*> **sequenceNumberDataMessagesToDrop**
Vector containing the message's sequence numbers being dropped.

uint32_t **dropLogLength**
Log dropped packets.

TCP Transport

TCPTransportDescriptor

struct **TCPTransportDescriptor** : public eprosima::fastdds::rtps::*SocketTransportDescriptor*
TCP Transport configuration

- `listening_ports:` list of ports to listen as server.
- `keep_alive_frequency_ms:` frequency of RTCP keep alive requests (in ms).
- `keep_alive_timeout_ms:` time since sending the last keep alive request to consider a connection as broken (in ms).

- `max_logical_port:` maximum number of logical ports to try during RTCP negotiation.
- `logical_port_range:` maximum number of logical ports per request to try during RTCP negotiation.
- `logical_port_increment:` increment between logical ports to try during RTCP negotiation.
- `enable_tcp_nodelay:` enables the TCP_NODELAY socket option.
- `calculate_crc:` true to calculate and send CRC on message headers.
- `check_crc:` true to check the CRC of incoming message headers.
- `apply_security:` true to use TLS (Transport Layer Security).
- `tls_config:` Configuration for TLS.
- `non_blocking_send:` do not block on send operations. When it is set to true, send operations will return immediately if the buffer might get full, but no error will be returned to the upper layer. This means that the application will behave as if the datagram is sent and lost.
- `tcp_negotiation_timeout:` time to wait for logical port negotiation (in ms).

Subclassed by *eprosima::fastdds::rtps::TCPv4TransportDescriptor*, *eprosima::fastdds::rtps::TCPv6TransportDescriptor*

Public Functions

inline void **add_listener_port**(uint16_t port)
Add listener port to the listening_ports list.

TCPTransportDescriptor()
Constructor.

TCPTransportDescriptor(const *TCPTransportDescriptor* &t)
Copy constructor.

TCPTransportDescriptor &**operator=**(const *TCPTransportDescriptor* &t)
Copy assignment.

virtual **~TCPTransportDescriptor**() = default
Destructor.

bool **operator==**(const *TCPTransportDescriptor* &t) const
Comparison operator.

Public Members

std::vector<uint16_t> **listening_ports**
List of ports to listen as server.

uint32_t **keep_alive_frequency_ms**
Frequency of RTCP keep alive requests (ms)

uint32_t **keep_alive_timeout_ms**
Time since sending the last keep alive request to consider a connection as broken (ms)

uint16_t **max_logical_port**
Maximum number of logical ports to try during RTCP negotiation.

uint16_t **logical_port_range**
: Maximum number of logical ports per request to try during RTCP negotiation.

uint16_t **logical_port_increment**
: Increment between logical ports to try during RTCP negotiation.

uint32_t **tcp_negotiation_timeout**
: Time to wait for logical port negotiation (ms). If a logical port is under negotiation, it waits for the negotiation to finish up to this timeout before trying to send a message to that port. Zero value means no waiting (default).

bool **enable_tcp_nodelay**
: Enables the TCP_NODELAY socket option.

bool **calculate_crc**
: Enables the calculation and sending of CRC on message headers.

bool **check_crc**
: Enables checking the CRC of incoming message headers.

bool **apply_security**
: Enables the use of TLS (Transport Layer Security)

TLSConfig **tls_config**
: Configuration of the TLS (Transport Layer Security)

ThreadSettings **keep_alive_thread**
: Thread settings for keep alive thread.

ThreadSettings **accept_thread**
: Thread settings for the accept connections thread.

bool **non_blocking_send**
: Whether to use non-blocking calls to send().

  When set to true, calls to send() will return immediately if the send buffer might get full. This may happen when receive buffer on reader's side is full. No error will be returned to the upper layer. This means that the application will behave as if the datagram is sent but lost (i.e. throughput may be reduced). This value is specially useful on high-frequency writers.

  When set to false, which is the default, calls to send() will block until the send buffer has space for the datagram. This may cause application lock.

struct **TLSConfig**
: TLS Configuration

  - `password:` password of the `private_key_file` or `rsa_private_key_file`.
  - `private_key_file:` path to the private key certificate file.
  - `rsa_private_key_file:` path to the private key RSA certificate file.
  - `cert_chain_file:` path to the public certificate chain file.

- `tmp_dh_file:` path to the Diffie-Hellman parameters file.
- `verify_file:` path to the CA (Certification-Authority) file.
- `verify_mode:` establishes the verification mode mask.
- `options:` establishes the SSL Context options mask.
- `verify_paths:` paths where the system will look for verification files.
- `default_verify_path:` look for verification files on the default paths.
- `handshake_role:` role that the transport will take on handshaking.
- `server_name:` server name or host name required in case Server Name Indication (SNI) is used.

Public Types

enum **TLSOptions**

Supported TLS features. Several options can be combined in the same TransportDescriptor using the *add_option()* member function.

- DEFAULT_WORKAROUNDS: implement various bug workarounds.
- NO_COMPRESSION: disable compression.
- NO_SSLV2: disable SSL v2.
- NO_SSLV3: disable SSL v3.
- NO_TLSV1: disable TLS v1.
- NO_TLSV1_1: disable TLS v1.1.
- NO_TLSV1_2: disable TLS v1.2.
- NO_TLSV1_3: disable TLS v1.3.
- SINGLE_DH_USE: always create a new key using Diffie-Hellman parameters.

Values:

enumerator **NONE**

enumerator **DEFAULT_WORKAROUNDS**

enumerator **NO_COMPRESSION**

enumerator **NO_SSLV2**

enumerator **NO_SSLV3**

enumerator **NO_TLSV1**

enumerator **NO_TLSV1_1**

enumerator **NO_TLSV1_2**

enumerator **NO_TLSV1_3**

enumerator **SINGLE_DH_USE**

enum **TLSVerifyMode**
Peer node verification options. Several verification options can be combined in the same TransportDescriptor using the *add_verify_mode()* member function.

- VERIFY_NONE: perform no verification.
- VERIFY_PEER: perform verification of the peer.
- VERIFY_FAIL_IF_NO_PEER_CERT: fail verification if the peer has no certificate. Ignored unless VERIFY_PEER is also set.
- VERIFY_CLIENT_ONCE: do not request client certificate on renegotiation. Ignored unless VERIFY_PEER is also set.

Values:

enumerator **UNUSED**

enumerator **VERIFY_NONE**

enumerator **VERIFY_PEER**

enumerator **VERIFY_FAIL_IF_NO_PEER_CERT**

enumerator **VERIFY_CLIENT_ONCE**

enum **TLSHandShakeRole**
Role that the transport will take on handshaking.

- DEFAULT: configured as client if connector, and as server if acceptor.
- CLIENT: configured as client.
- SERVER: configured as server.

Values:

enumerator **DEFAULT**

enumerator **CLIENT**

enumerator **SERVER**

Public Functions

inline void **add_verify_mode**(const *TLSVerifyMode* verify)
Add verification modes to the verification mode mask.

inline bool **get_verify_mode**(const *TLSVerifyMode* verify) const
Get the verification mode mask.

inline void **add_option**(const *TLSOptions* option)
Add TLS features to the SSL Context options mask.

inline bool **get_option**(const *TLSOptions* option) const
Get the SSL Context options mask.

inline bool **operator==**(const *TLSConfig* &t) const
Comparison operator.

Public Members

std::string **password**
Password of the `private_key_file` or `rsa_private_key_file`.

uint32_t **options** = *TLSOptions*::*NONE*
SSL context options mask.

std::string **cert_chain_file**
Path to the public certificate chain file.

std::string **private_key_file**
Path to the private key certificate file.

std::string **tmp_dh_file**
Path to the Diffie-Hellman parameters file.

std::string **verify_file**
Path to the CA (Certification-Authority) file.

uint8_t **verify_mode** = *TLSVerifyMode*::*UNUSED*
Verification mode mask.

std::vector<std::string> **verify_paths**
Paths where the system will look for verification files.

bool **default_verify_path** = false
Look for verification files on the default paths.

int32_t **verify_depth** = -1
Maximum allowed depth for verifying intermediate certificates. Do not override.

std::string **rsa_private_key_file**
Path to the private key RSA certificate file.

TLSHandShakeRole **handshake_role** = *TLSHandShakeRole*::*DEFAULT*
Role that the transport will take on handshaking.

std::string **server_name**
Server name or host name required in case Server Name Indication (SNI) is used.

TCPv4TransportDescriptor

struct **TCPv4TransportDescriptor** : public eprosima::fastdds::rtps::*TCPTransportDescriptor*

TCPv4 Transport configuration. The kind value for *TCPv4TransportDescriptor* is given by `eprosima::fastrtps::rtps::LOCATOR_KIND_TCPv4`.

- `wan_addr:` Public IP address. Peers on a different LAN will use this IP for communications with this host.

Public Functions

virtual **~TCPv4TransportDescriptor**() = default

Destructor.

virtual *TransportInterface* ***create_transport**() const override

Factory method pattern. It will create and return a *TransportInterface* corresponding to this descriptor. This provides an interface to the NetworkFactory to create the transports without the need to know about their type

inline void **set_WAN_address**(fastrtps::rtps::*octet* o1, fastrtps::rtps::*octet* o2, fastrtps::rtps::*octet* o3, fastrtps::rtps::*octet* o4)

Set the public IP address.

inline void **set_WAN_address**(const std::string &in_address)

Set the public IP address.

inline std::string **get_WAN_address**()

Get the public IP address.

TCPv4TransportDescriptor()

Constructor.

TCPv4TransportDescriptor(const *TCPv4TransportDescriptor* &t)

Copy constructor.

TCPv4TransportDescriptor &**operator=**(const *TCPv4TransportDescriptor* &t)

Copy assignment.

bool **operator==**(const *TCPv4TransportDescriptor* &t) const

Comparison operator.

Public Members

fastrtps::rtps::*octet* **wan_addr**[4]

Public IP address.

TCPv6TransportDescriptor

struct **TCPv6TransportDescriptor** : public eprosima::fastdds::rtps::*TCPTransportDescriptor*
TCPv6 Transport configuration The kind value for *TCPv6TransportDescriptor* is given by `eprosima::fastrtps::rtps::LOCATOR_KIND_TCPv6`.

Public Functions

virtual **~TCPv6TransportDescriptor**() = default
Destructor.

virtual *TransportInterface* ***create_transport**() const override
Factory method pattern. It will create and return a *TransportInterface* corresponding to this descriptor. This provides an interface to the NetworkFactory to create the transports without the need to know about their type

TCPv6TransportDescriptor()
Constructor.

TCPv6TransportDescriptor(const *TCPv6TransportDescriptor* &t)
Copy constructor.

TCPv6TransportDescriptor &**operator=**(const *TCPv6TransportDescriptor* &t) = default
Copy assignment.

bool **operator==**(const *TCPv6TransportDescriptor* &t) const
Comparison operator.

Shared Memory Transport

SharedMemTransportDescriptor

struct **SharedMemTransportDescriptor** : public eprosima::fastdds::rtps::PortBasedTransportDescriptor
Shared memory transport configuration. The kind value for *SharedMemTransportDescriptor* is given by eprosima::fastrtps::rtps::LOCATOR_KIND_SHM.

- segment_size_: size of the shared memory segment (in octets).
- port_queue_capacity_: size of the listening port (in messages).
- healthy_check_timeout_ms_: timeout for the health check of ports (ms).
- rtps_dump_file_: full path of the protocol dump file.

Public Functions

virtual **~SharedMemTransportDescriptor**() = default
Destructor.

virtual *TransportInterface* ***create_transport**() const override
Factory method pattern. It will create and return a *TransportInterface* corresponding to this descriptor. This provides an interface to the NetworkFactory to create the transports without the need to know about their type

inline virtual uint32_t **min_send_buffer_size**() const override
Minimum size of the send buffer.

SharedMemTransportDescriptor()
Constructor.

SharedMemTransportDescriptor(const *SharedMemTransportDescriptor* &t) = default
Copy constructor.

SharedMemTransportDescriptor &**operator=**(const *SharedMemTransportDescriptor* &t) = default
Copy assignment.

inline uint32_t **segment_size**() const
Return the size of the shared memory segment.

inline void **segment_size**(uint32_t segment_size)
Set the size of the shared memory segment.

inline virtual uint32_t **max_message_size**() const override
Return the maximum size of a single message in the transport (in octets)

inline void **max_message_size**(uint32_t max_message_size)
Set the maximum size of a single message in the transport (in octets)

inline uint32_t **port_queue_capacity**() const
Return the size of the listening port (in messages)

inline void **port_queue_capacity**(uint32_t port_queue_capacity)
Set the size of the listening port (in messages)

inline uint32_t **healthy_check_timeout_ms**() const
Return the timeout for the health check of ports (ms)

inline void **healthy_check_timeout_ms**(uint32_t healthy_check_timeout_ms)
Set the timeout for the health check of ports (ms)

inline std::string **rtps_dump_file**() const
Return the full path of the protocol dump file.

inline void **rtps_dump_file**(const std::string &rtps_dump_file)
Set the full path of the protocol dump file.

inline ThreadSettings **dump_thread**() const
Return the thread settings for the transport dump thread.

inline void **dump_thread**(const ThreadSettings &dump_thread)
Set the thread settings for the transport dump thread.

bool **operator==**(const *SharedMemTransportDescriptor* &t) const
Comparison operator.

6.32.4 LOG

eProsima Fast DDS Logging Module API

Colors

A collection of macros for ease the stream coloring.

Color Blue

C_BLUE "\033[34m"

Color Bright

C_BRIGHT "\033[1m"

Color Bright Blue

C_B_BLUE "\033[34;1m"

Color Bright Cyan

C_B_CYAN "\033[36;1m"

Color Bright Green

C_B_GREEN "\033[32;1m"

Color Bright Magenta

C_B_MAGENTA "\033[35;1m"

Color Bright Red

C_B_RED "\033[31;1m"

Color Bright White

C_B_WHITE "\033[37;1m"

Color Bright Yellow

C_B_YELLOW "\033[33;1m"

Color Cyan

C_CYAN "\033[36m"

Color Def

C_DEF "\033[m"

Color Green

C_GREEN "\033[32m"

Color Magenta

C_MAGENTA "\033[35m"

Color Red

C_RED "\033[31m"

Color White

C_WHITE "\033[37m"

Color Yellow

C_YELLOW "\033[33m"

FileConsumer

class **FileConsumer** : public eprosima::fastdds::dds::*OStreamConsumer*

Public Functions

FileConsumer()
Default constructor: filename = "output.log", append = false.

FileConsumer(const std::string &filename, bool append = false)
Constructor with parameters.

Parameters

- **filename** – path of the output file where the log will be wrote.
- **append** – indicates if the consumer must append the content in the filename.

Log

class **Log**
Logging utilities. Logging is accessed through the three macros above, and configuration on the log output can be achieved through static methods on the class. Logging at various levels can be disabled dynamically (through the Verbosity level) or statically (through the LOG_NO_[VERB] macros) for maximum performance.

Public Types

enum **Kind**
Types of log entry.

- Error: Maximum priority. Can only be disabled statically through LOG_NO_ERROR.
- Warning: Medium priority. Can be disabled statically and dynamically.
- Info: Low priority. Useful for debugging. Disabled by default on release branches.

Values:

enumerator **Error**

enumerator **Warning**

enumerator **Info**

Public Static Functions

static void **RegisterConsumer**(std::unique_ptr<*LogConsumer*> &&consumer)
Registers an user defined consumer to route log output. There is a default stdout consumer active as default.

Parameters **consumer** – r-value to a consumer unique_ptr. It will be invalidated after the call.

static void **ClearConsumers**()
Removes all registered consumers, including the default stdout.

static void **ReportFilenames**(bool)
Enables the reporting of filenames in log entries. Disabled by default.

static void **ReportFunctions**(bool)
Enables the reporting of function names in log entries. Enabled by default when supported.

static void **SetVerbosity**(*Log*::*Kind*)
Sets the verbosity level, allowing for messages equal or under that priority to be logged.

static *Log*::*Kind* **GetVerbosity**()
Returns the current verbosity level.

static void **SetCategoryFilter**(const std::regex&)
Sets a filter that will pattern-match against log categories, dropping any unmatched categories.

static void **SetFilenameFilter**(const std::regex&)
Sets a filter that will pattern-match against filenames, dropping any unmatched categories.

static void **SetErrorStringFilter**(const std::regex&)
Sets a filter that will pattern-match against the provided error string, dropping any unmatched categories.

static void **SetThreadConfig**(const rtps::ThreadSettings&)
Sets thread configuration for the logging thread.

static void **Reset**()
Returns the logging engine to configuration defaults.

static void **Flush**()
Waits until all info logged up to the call time is consumed.

static void **KillThread**()
Stops the logging thread. It will re-launch on the next call to a successful log macro.

static void **QueueLog**(const std::string &message, const *Log*::*Context*&, *Log*::*Kind*)
Not recommended to call this method directly! Use the following macros:

- *EPROSIMA_LOG_INFO(cat, msg)*;
- *EPROSIMA_LOG_WARNING(cat, msg)*;
- *EPROSIMA_LOG_ERROR(cat, msg)*;

Todo:
this method takes 2 mutexes (same mutex) internally. This is a very high sensible point of the code and it should be refactored to be as efficient as possible.

struct **Context**

struct **Entry**

LogConsumer

class **LogConsumer**
Consumes a log entry to output it somewhere.

Subclassed by *eprosima::fastdds::dds::OStreamConsumer*

EPROSIMA_LOG_ERROR

EPROSIMA_LOG_ERROR(cat, msg) EPROSIMA_LOG_ERROR_IMPL_(cat, msg)
Logs an error. Disable reporting through define LOG_NO_ERROR.

EPROSIMA_LOG_INFO

EPROSIMA_LOG_INFO(cat, msg) EPROSIMA_LOG_INFO_IMPL_(cat, msg)
Logs an info message. Disable it through Log::SetVerbosity, define LOG_NO_INFO, or being in a release branch.

eProsima log layer. Logging categories and verbosity can be specified dynamically at runtime. However, even on a category not covered by the current verbosity level, there is some overhead on calling a log macro. For maximum performance, you can opt out of logging any particular level by defining the following symbols:

- define LOG_NO_ERROR
- define LOG_NO_WARNING
- define LOG_NO_INFO

Additionally. the lowest level (Info) is disabled by default on release branches.

EPROSIMA_LOG_WARNING

EPROSIMA_LOG_WARNING(cat, msg) EPROSIMA_LOG_WARNING_IMPL_(cat, msg)
Logs a warning. Disable reporting through Log::SetVerbosity or define LOG_NO_WARNING.

OStreamConsumer

class **OStreamConsumer** : public eprosima::fastdds::dds::*LogConsumer*
Subclassed by *eprosima::fastdds::dds::FileConsumer*, *eprosima::fastdds::dds::StdoutConsumer*, *eprosima::fastdds::dds::StdoutErrConsumer*

StdoutConsumer

class **StdoutConsumer** : public eprosima::fastdds::dds::*OStreamConsumer*

StdoutErrConsumer

class **StdoutErrConsumer** : public eprosima::fastdds::dds::*OStreamConsumer*

Public Functions

virtual void **stderr_threshold**(const *Log*::*Kind* &kind)
Set the stderr_threshold to a *Log::Kind*. This threshold decides which log messages are output on STDOUT, and which are output to STDERR. *Log* messages with a *Log::Kind* equal to or more severe than the stderr_threshold are output to STDERR using std::cerr. *Log* messages with a *Log::Kind* less severe than the stderr_threshold are output to STDOUT using std::cout.

Parameters **kind** – The *Log::Kind* to which stderr_threshold is set.

virtual *Log*::*Kind* **stderr_threshold**() const
Retrieve the stderr_threshold.

Returns The *Log::Kind* to which stderr_threshold is set.

Public Static Attributes

static constexpr *Log*::*Kind* **STDERR_THRESHOLD_DEFAULT** = *Log*::*Kind*::*Warning*
Default value of stderr_threshold.

6.32.5 Statistics

eProsima Fast DDS Statistics Module extension API.

DomainParticipant

class **DomainParticipant** : public eprosima::fastdds::dds::*DomainParticipant*
Class *DomainParticipant*: extends standard DDS *DomainParticipant* class to include specific methods for the Statistics module

Public Functions

ReturnCode_t **enable_statistics_datawriter**(const std::string &topic_name, const eprosima::fastdds::dds::*DataWriterQos* &dwqos)
This operation enables a Statistics DataWriter.

Parameters

- **topic_name** – **[in]** Name of the topic associated to the Statistics DataWriter
- **dwqos** – **[in]** *DataWriterQos* to be set

Returns RETCODE_UNSUPPORTED if the FASTDDS_STATISTICS CMake option has not been set, RETCODE_BAD_PARAMETER if the topic name provided does not correspond to any Statistics DataWriter, RETCODE_INCONSISTENT_POLICY if the *DataWriterQos* provided are inconsistent, RETCODE_OK if the DataWriter has been created or if it has been created previously, and RETCODE_ERROR otherwise

ReturnCode_t **enable_statistics_datawriter_with_profile**(const std::string &profile_name, const std::string &topic_name)

This operation enables a Statistics DataWriter from a given profile.

Parameters

- **profile_name** – **[in]** DataWriter QoS profile name
- **topic_name** – **[in]** Name of the statistics topic to be enabled.

Returns RETCODE_UNSUPPORTED if the FASTDDS_STATISTICS CMake option has not been set, RETCODE_BAD_PARAMETER if the topic name provided does not correspond to any Statistics DataWriter, RETCODE_INCONSISTENT_POLICY if the *DataWriterQos* provided in profile are inconsistent, RETCODE_OK if the DataWriter has been created or if it has been created previously, and RETCODE_ERROR otherwise

ReturnCode_t **disable_statistics_datawriter**(const std::string &topic_name)

This operation disables a Statistics DataWriter.

Parameters **topic_name** – Name of the topic associated to the Statistics DataWriter

Returns RETCODE_UNSUPPORTED if the FASTDDS_STATISTICS CMake option has not been set, RETCODE_BAD_PARAMETER if the topic name provided does not correspond to any Statistics DataWriter, RETCODE_OK if the DataWriter has been correctly deleted or does not exist, and RETCODE_ERROR otherwise

ReturnCode_t **enable_monitor_service**()

Enables the monitor service in the *DomainParticipant*.

Note: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK if the monitor service could be correctly enabled.

Returns RETCODE_ERROR if the monitor service could not be enabled properly.

Returns RETCODE_UNSUPPORTED if FASTDDS_STATISTICS is not enabled.

ReturnCode_t **disable_monitor_service**()

Disables the monitor service in this *DomainParticipant*. Does nothing if the service was not enabled before.

Note: Not supported yet. Currently returns RETCODE_UNSUPPORTED

Returns RETCODE_OK if the monitor service could be correctly disabled.

Returns RETCODE_NOT_ENABLED if the monitor service was not previously enabled.

Returns RETCODE_ERROR if the service could not be properly disabled.

Returns RETCODE_UNSUPPORTED if FASTDDS_STATISTICS is not enabled.

ReturnCode_t **fill_discovery_data_from_cdr_message**(fastrtps::rtps::*ParticipantProxyData* &data, statistics::MonitorServiceStatusData &msg)

fills in the ParticipantProxyData from a MonitorService Message

Parameters

- **data** – **[out]** Proxy to fill
- **msg** – **[in]** MonitorService Message to get the proxy information from.

Returns RETCODE_OK if the operation succeeds.

Returns RETCODE_ERROR if the operation fails.

ReturnCode_t **fill_discovery_data_from_cdr_message**(fastrtps::rtps::*WriterProxyData* &data, statistics::MonitorServiceStatusData &msg)

fills in the WriterProxyData from a MonitorService Message

Parameters

- **data** – **[out]** Proxy to fill.
- **msg** – **[in]** MonitorService Message to get the proxy information from.

Returns RETCODE_OK if the operation succeeds.

Returns RETCODE_ERROR if the operation fails.

ReturnCode_t **fill_discovery_data_from_cdr_message**(fastrtps::rtps::*ReaderProxyData* &data, statistics::MonitorServiceStatusData &msg)

fills in the ReaderProxyData from a MonitorService Message

Parameters

- **data** – **[out]** Proxy to fill.
- **msg** – **[in]** MonitorService Message to get the proxy information from.

Returns RETCODE_OK if the operation succeeds.

Returns RETCODE_ERROR if the operation fails.

ReturnCode_t **fill_discovery_data_from_cdr_message**(fastrtps::rtps::*ParticipantProxyData* &data, const statistics::MonitorServiceStatusData &msg)

fills in the ParticipantProxyData from a MonitorService Message

Parameters

- **data** – **[out]** Proxy to fill
- **msg** – **[in]** MonitorService Message to get the proxy information from.

Returns RETCODE_OK if the operation succeeds.

Returns RETCODE_ERROR if the operation fails.

ReturnCode_t **fill_discovery_data_from_cdr_message**(fastrtps::rtps::*WriterProxyData* &data, const statistics::MonitorServiceStatusData &msg)

fills in the WriterProxyData from a MonitorService Message

Parameters

- **data** – **[out]** Proxy to fill.
- **msg** – **[in]** MonitorService Message to get the proxy information from.

Returns RETCODE_OK if the operation succeeds.

Returns RETCODE_ERROR if the operation fails.

ReturnCode_t **fill_discovery_data_from_cdr_message**(fastrtps::rtps::*ReaderProxyData* &data, const statistics::MonitorServiceStatusData &msg)

fills in the ReaderProxyData from a MonitorService Message

Parameters

- **data** – **[out]** Proxy to fill.
- **msg** – **[in]** MonitorService Message to get the proxy information from.

Returns RETCODE_OK if the operation succeeds.

Returns RETCODE_ERROR if the operation fails.

Public Static Functions

static *DomainParticipant* ***narrow**(eprosima::fastdds::dds::*DomainParticipant* *domain_participant)

This operation narrows the DDS *DomainParticipant* to the Statistics *DomainParticipant*.

Parameters **domain_participant** – Reference to the DDS *DomainParticipant*

Returns Reference to the Statistics *DomainParticipant* if successful. nullptr otherwise.

static const *DomainParticipant* ***narrow**(const eprosima::fastdds::dds::*DomainParticipant* *domain_participant)

This operation narrows the DDS *DomainParticipant* to the Statistics *DomainParticipant*.

Parameters **domain_participant** – Constant reference to the DDS *DomainParticipant*

Returns Constant reference to the Statistics *DomainParticipant* if successful. nullptr otherwise.

DataWriterQos

class **DataWriterQos** : public eprosima::fastdds::dds::*DataWriterQos*

Class *DataWriterQos*: extends standard DDS *DataWriterQos* class to include specific default constructor for the recommended *DataWriterQos* profile.

Public Functions

DataWriterQos()

Constructor.

const eprosima::fastdds::statistics::dds::*DataWriterQos*
eprosima::fastdds::statistics::dds::STATISTICS_DATAWRITER_QOS

Constant to access default Statistics DataWriter Qos.

DataReaderQos

class **DataReaderQos** : public eprosima::fastdds::dds::*DataReaderQos*

Class *DataReaderQos*: extends standard DDS *DataReaderQos* class to include specific default constructor for the recommended *DataReaderQos* profile.

Public Functions

DataReaderQos()

Constructor.

const eprosima::fastdds::statistics::dds::*DataReaderQos*
eprosima::fastdds::statistics::dds::STATISTICS_DATAREADER_QOS

Constant to access default Statistics DataReader Qos.

Topic names

constexpr const char *__eprosima::fastdds::statistics::__**HISTORY_LATENCY_TOPIC** = "_fastdds_statistics_history2history_latency"

Statistic topic that reports the write-to-notification latency between any two pairs of matched DataWriter-DataReader histories

constexpr const char *__eprosima::fastdds::statistics::__**NETWORK_LATENCY_TOPIC** = "_fastdds_statistics_network_latency"

Statistics topic that reports the network latency (message group to message receiver) between any two communicating locators

constexpr const char *__eprosima::fastdds::statistics::__**PUBLICATION_THROUGHPUT_TOPIC** = "_fastdds_statistics_publication_throughput"

Statistic topic that reports the publication's throughput (amount of data sent) for every DataWriter.

constexpr const char *__eprosima::fastdds::statistics::__**SUBSCRIPTION_THROUGHPUT_TOPIC** = "_fastdds_statistics_subscription_throughput"

Statistics topic that reports the subscription's throughput (amount of data received) for every DataReader.

constexpr const char *__eprosima::fastdds::statistics::__**RTPS_SENT_TOPIC** = "_fastdds_statistics_rtps_sent"

Statistics topic that reports the number of RTPS packets and bytes sent to each locator.

constexpr const char *__eprosima::fastdds::statistics::__**RTPS_LOST_TOPIC** = "_fastdds_statistics_rtps_lost"

Statistics topic that reports the number of RTPS packets and bytes that have been lost in the network.

constexpr const char *__eprosima::fastdds::statistics::__**RESENT_DATAS_TOPIC** = "_fastdds_statistics_resent_datas"

Statistics topic that reports the number of DATA/DATAFRAG sub-messages resent.

constexpr const char *__eprosima::fastdds::statistics::__**HEARTBEAT_COUNT_TOPIC** = "_fastdds_statistics_heartbeat_count"

Statistics topic that reports the number of HEARTBEATs that each non discovery DataWriter sends.

constexpr const char *eprosima::fastdds::statistics::**ACKNACK_COUNT_TOPIC** = "_fastdds_statistics_acknack_count"

Statistics topic that reports the number of ACKNACKs that each non discovery DataReader sends.

constexpr const char *eprosima::fastdds::statistics::**NACKFRAG_COUNT_TOPIC** = "_fastdds_statistics_nackfrag_count"

Statistics topic that reports the number of NACKFRAGs that each non discovery DataReader sends.

constexpr const char *eprosima::fastdds::statistics::**GAP_COUNT_TOPIC** = "_fastdds_statistics_gap_count"

Statistics topic that reports the number of GAPs that each non discovery DataWriter sends.

constexpr const char *eprosima::fastdds::statistics::**DATA_COUNT_TOPIC** = "_fastdds_statistics_data_count"

Statistics topic that reports the number of DATA/DATAFRAG sub-messages that each non discovery DataWriter sends.

constexpr const char *eprosima::fastdds::statistics::**PDP_PACKETS_TOPIC** = "_fastdds_statistics_pdp_packets"

Statistics topic that reports the number of PDP discovery traffic RTPS packets transmitted by each DDS participant.

constexpr const char *eprosima::fastdds::statistics::**EDP_PACKETS_TOPIC** = "_fastdds_statistics_edp_packets"

Statistics topic that reports the number of EDP discovery traffic RTPS packets transmitted by each DDS participant.

constexpr const char *eprosima::fastdds::statistics::**DISCOVERY_TOPIC** = "_fastdds_statistics_discovered_entity"

Statistics topic that reports when new entities are discovered.

constexpr const char *eprosima::fastdds::statistics::**SAMPLE_DATAS_TOPIC** = "_fastdds_statistics_sample_datas"

Statistics topic that reports the number of DATA/DATAFRAG sub-messages needed to send a single sample.

constexpr const char *eprosima::fastdds::statistics::**PHYSICAL_DATA_TOPIC** = "_fastdds_statistics_physical_data"

Statistics topic that reports the host, user and process where the module is running.

6.33 Python API Reference

This section presents the most commonly used Python APIs provided by *Fast DDS*.

6.33.1 DDS DCPS PIM

Data Distribution Service (DDS) Data-Centric Publish-Subscribe (DCPS) Platform Independent Model (PIM) API

Core

Entity

class fastdds.**Entity**(**args*, ***kwargs*)

The Entity class is the abstract base class for all the objects that support QoS policies, a listener and a status condition.

close()

This operation disables the Entity before closing it

enable()

This operation enables the Entity

Return type ReturnCode_t

Returns RETCODE_OK

get_instance_handle()

Retrieves the instance handler that represents the Entity

Return type InstanceHandle_t

Returns Reference to the InstanceHandle

get_status_changes()

Retrieves the set of triggered statuses in the Entity

Triggered statuses are the ones whose value has changed since the last time the application read the status. When the entity is first created or if the entity is not enabled, all communication statuses are in the non-triggered state, so the list returned by the get_status_changes operation will be empty. The list of statuses returned by the get_status_changes operation refers to the status that are triggered on the Entity itself and does not include statuses that apply to contained entities.

Return type *StatusMask*

Returns const reference to the StatusMask with the triggered statuses set to 1

get_status_mask()

Retrieves the set of relevant statuses for the Entity

Return type *StatusMask*

Returns Reference to the StatusMask with the relevant statuses set to 1

get_statuscondition()

Allows access to the StatusCondition associated with the Entity

Return type StatusCondition

Returns Reference to StatusCondition object

is_enabled()
Checks if the Entity is enabled

Return type boolean

Returns true if enabled, false if not

property thisown
The membership flag

DomainEntity

class fastdds.DomainEntity(*args, **kwargs)
The DomainEntity class is a subclass of Entity created in order to differentiate between DomainParticipants and the rest of Entities

property thisown
The membership flag

Policy

DataRepresentationId

class fastdds.XCDR_DATA_REPRESENTATION(*args: Any, **kwargs: Any)

class fastdds.XML_DATA_REPRESENTATION(*args: Any, **kwargs: Any)

class fastdds.XCDR2_DATA_REPRESENTATION(*args: Any, **kwargs: Any)

class fastdds.DEFAULT_DATA_REPRESENTATION(*args: Any, **kwargs: Any)

DataRepresentationQosPolicy

class fastdds.DataRepresentationQosPolicy
With multiple standard data Representations available, and vendor-specific extensions possible, DataWriters and DataReaders must be able to negotiate which data representation(s) to use. This negotiation shall occur based on DataRepresentationQosPolicy.

Warning: If a writer's offered representation is contained within a reader's sequence, the offer satisfies the request and the policies are compatible. Otherwise, they are incompatible.

Notes: Immutable Qos Policy

clear()
Clears the QosPolicy object

property m_value

List of 'DataRepresentationId'. By default, empty list.

property thisown
The membership flag

DataSharingQosPolicy

class fastdds.**DataSharingQosPolicy**(**args*)

Qos Policy to configure the data sharing

Notes: Immutable Qos Policy

add_domain_id(**args*)

automatic(**args*)

Overload 1:

Configures the DataSharing in automatic mode

The default shared memory directory of the OS is used. A default domain ID is automatically computed.

Overload 2:

Configures the DataSharing in automatic mode

The default shared memory directory of the OS is used.

Parameters **domain_ids** (*std::vector< uint16_t,std::allocator< uint16_t > >*) – the user configured DataSharing domain IDs (16 bits).

Overload 3:

Configures the DataSharing in automatic mode

A default domain ID is automatically computed.

Parameters **directory** (*string*) – The shared memory directory to use.

Overload 4:

Configures the DataSharing in automatic mode

Parameters

- **directory** (*string*) – The shared memory directory to use.
- **domain_ids** (*std::vector< uint16_t,std::allocator< uint16_t > >*) – the user configured DataSharing domain IDs (16 bits).

clear()

Clears the QosPolicy object

data_sharing_listener_thread(**args*)

Overload 1:

Getter for DataSharing listener thread ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 2:

Getter for DataSharing listener thread ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 3:

Setter for the DataSharing listener thread ThreadSettings

Parameters **value** (ThreadSettings) – New ThreadSettings to be set

domain_ids()

Gets the set of DataSharing domain IDs.

Each domain ID is 64 bit long. However, user-defined domain IDs are only 16 bit long, while the rest of the 48 bits are used for the automatically generated domain ID (if any).

- Automatic domain IDs use the 48 MSB and leave the 16 LSB as zero.
- User defined domain IDs use the 16 LSB and leave the 48 MSB as zero.

Return type std::vector< uint64_t,std::allocator< uint64_t > >

Returns the current DataSharing domain IDs

kind()

Return type int

Returns the current DataSharing configuration mode

max_domains()

Return type int

Returns the current configured maximum number of domain IDs

off()

Configures the DataSharing in disabled mode

on(**args*)

Overload 1:

Configures the DataSharing in active mode

A default domain ID is automatically computed.

Parameters **directory** (*string*) – The shared memory directory to use. It is mandatory to provide a non-empty name or the creation of endpoints will fail.

Overload 2:

Configures the DataSharing in active mode

Parameters

- **directory** (*string*) – The shared memory directory to use. It is mandatory to provide a non-empty name or the creation of endpoints will fail.
- **domain_ids** (*std::vector< uint16_t,std::allocator< uint16_t > >*) – the user configured DataSharing domain IDs (16 bits).

set_max_domains(*size*)

Parameters **size** (*int*) – the new maximum number of domain IDs

shm_directory()

Return type string

Returns the current DataSharing shared memory directory

property thisown
The membership flag

DataSharingKind

class fastdds.**AUTO**(**args: Any*, ***kwargs: Any*)

class fastdds.**ON**(**args: Any*, ***kwargs: Any*)

class fastdds.**OFF**(**args: Any*, ***kwargs: Any*)

DeadlineQosPolicy

class fastdds.**DeadlineQosPolicy**
DataReader expects a new sample updating the value of each instance at least once every deadline period. DataWriter indicates that the application commits to write a new value (using the DataWriter) for each instance managed by the DataWriter at least once every deadline period.

Notes: Mutable Qos Policy

clear()
Clears the QosPolicy object

property period
Maximum time expected between samples. It is inconsistent for a DataReader to have a DEADLINE period less than its TimeBasedFilterQosPolicy minimum_separation.

By default, c_TimeInifinite.

property thisown
The membership flag

DestinationOrderQosPolicy

class fastdds.DestinationOrderQosPolicy
Controls the criteria used to determine the logical order among changes made by Publisher entities to the same instance of data (i.e., matching Topic and key).

Warning: This QosPolicy can be defined and is transmitted to the rest of the network but is not implemented in this version.

Notes: Immutable Qos Policy

clear()
Clears the QosPolicy object

property kind

DestinationOrderQosPolicyKind. By default, BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS.

property thisown
The membership flag

DestinationOrderQosPolicyKind

class fastdds.BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS(*args: Any, **kwargs: Any)

class fastdds.BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS(*args: Any, **kwargs: Any)

DisablePositiveACKsQosPolicy

class fastdds.DisablePositiveACKsQosPolicy
Class DisablePositiveACKsQosPolicy to disable sending of positive ACKs

Notes: Immutable Qos Policy

clear()
Clears the QosPolicy object

property duration
The duration to keep samples for (not serialized as not needed by reader). By default, c_TimeInfinite

property enabled
True if this QoS is enabled. By default, false

property thisown
The membership flag

DurabilityQosPolicy

class fastdds.**DurabilityQosPolicy**

This policy expresses if the data should 'outlive' their writing time.

Notes: Immutable Qos Policy

clear()

Clears the QosPolicy object

durabilityKind(**args*)

Overload 1:

Translates kind to rtps layer equivalent

Return type int

Returns fastrtps::rtps::DurabilityKind_t

Overload 2:

Set kind passing the rtps layer equivalent kind

Parameters **new_kind** (*int*) – fastrtps::rtps::DurabilityKind_t

property kind

DurabilityQosPolicyKind.

By default the value for DataReaders: VOLATILE_DURABILITY_QOS, for DataWriters TRANSIENT_LOCAL_DURABILITY_QOS

property thisown

The membership flag

DurabilityQosPolicyKind

class fastdds.**VOLATILE_DURABILITY_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**TRANSIENT_LOCAL_DURABILITY_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**TRANSIENT_DURABILITY_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**PERSISTENT_DURABILITY_QOS**(**args: Any*, ***kwargs: Any*)

DurabilityServiceQosPolicy

class fastdds.**DurabilityServiceQosPolicy**

Specifies the configuration of the durability service. That is, the service that implements the DurabilityQosPolicy kind of TRANSIENT and PERSISTENT.

Warning: This QosPolicy can be defined and is transmitted to the rest of the network but is not implemented in this version.

Notes: Immutable Qos Policy

clear()
Clears the QosPolicy object

property history_depth
Number of most recent values that should be maintained on the History. It only have effect if the history_kind is KEEP_LAST_HISTORY_QOS.

By default, 1.

property history_kind
Controls the HistoryQosPolicy of the fictitious DataReader that stores the data within the durability service.

By default, KEEP_LAST_HISTORY_QOS.

property max_instances
Control the ResourceLimitsQos of the implied DataReader that stores the data within the durability service. Represents the maximum number of instances DataWriter (or DataReader) can manage.

By default, LENGTH_UNLIMITED.

property max_samples
Control the ResourceLimitsQos of the implied DataReader that stores the data within the durability service. Specifies the maximum number of data-samples the DataWriter (or DataReader) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one DataWriter (or DataReader). It is inconsistent for this value to be less than max_samples_per_instance.

By default, LENGTH_UNLIMITED.

property max_samples_per_instance
Control the ResourceLimitsQos of the implied DataReader that stores the data within the durability service. Represents the maximum number of samples of any one instance a DataWriter(or DataReader) can manage. It is inconsistent for this value to be greater than max_samples.

By default, LENGTH_UNLIMITED.

property service_cleanup_delay
Control when the service is able to remove all information regarding a data-instance.

By default, c_TimeZero.

property thisown
The membership flag

EntityFactoryQosPolicy

class fastdds.**EntityFactoryQosPolicy**(**args*)
Controls the behavior of the entity when acting as a factory for other entities. In other words, configures the side-effects of the create_* and delete_* operations.

Notes: Mutable Qos Policy

property autoenable_created_entities
Specifies whether the entity acting as a factory automatically enables the instances it creates. If True the factory will automatically enable each created Entity otherwise it will not.

By default, True.

clear()

property thisown
The membership flag

GenericDataQosPolicy

class fastdds.**GenericDataQosPolicy**(**args*)

Class GenericDataQosPolicy, base class to transmit user data during the discovery phase.

clear()

Clears the QosPolicy object

dataVec()

Return type eprosima::fastrtps::ResourceLimitedVector< unsigned char >::collection_type

Returns const reference to the internal raw data.

data_vec(**args*)

Overload 1:

Returns raw data vector.

Return type eprosima::fastrtps::ResourceLimitedVector< unsigned char >::collection_type

Returns raw data as vector of octets.

Overload 2:

Returns raw data vector.

Return type eprosima::fastrtps::ResourceLimitedVector< unsigned char >::collection_type

Returns raw data as vector of octets.

Overload 3:

Sets raw data vector.

Parameters **vec** (*eprosima::fastrtps::ResourceLimitedVector< unsigned char >::collection_type*) – raw data to set.

getValue()

Returns raw data vector.

Return type eprosima::fastrtps::ResourceLimitedVector< unsigned char >::collection_type

Returns raw data as vector of octets.

resize(*new_size*)

setValue(*vec*)

Sets raw data vector.

Parameters **vec** (*eprosima::fastrtps::ResourceLimitedVector< unsigned char >::collection_type*) – raw data to set.

set_max_size(*size*)

Set the maximum size of the user data and reserves memory for that much.

Parameters **size** (*int*) – new maximum size of the user data. Zero for unlimited size

property thisown
The membership flag

GroupDataQosPolicy

class fastdds.**GroupDataQosPolicy**(**args*)

property thisown
The membership flag

HistoryQosPolicy

class fastdds.**HistoryQosPolicy**
Specifies the behavior of the Service in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers. This QoS policy controls whether the Service should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between. On the publishing side this policy controls the samples that should be maintained by the DataWriter on behalf of existing DataReader entities. The behavior with regards to a DataReaderentities discovered after a sample is written is controlled by the DURABILITY QoS policy. On the subscribing side it controls the samples that should be maintained until the application "takes" them from the Service.

Notes: Immutable Qos Policy

clear()
Clears the QosPolicy object

property depth
History depth. By default, 1. If a value other than 1 is specified, it should be consistent with the settings of the ResourceLimitsQosPolicy.

Warning: Only takes effect if the kind is KEEP_LAST_HISTORY_QOS.

property kind

HistoryQosPolicyKind. By default, KEEP_LAST_HISTORY_QOS.

property thisown
The membership flag

HistoryQosPolicyKind

class fastdds.**KEEP_LAST_HISTORY_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**KEEP_ALL_HISTORY_QOS**(**args: Any*, ***kwargs: Any*)

LatencyBudgetQosPolicy

class fastdds.LatencyBudgetQosPolicy

Specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver's application-cache and the receiving application is notified of the fact.This policy is a hint to the Service, not something that must be monitored or enforced. The Service is not required to track or alert the user of any violation.

Warning: This QosPolicy can be defined and is transmitted to the rest of the network but is not implemented in this version.

Notes: Mutable Qos Policy

clear()

Clears the QosPolicy object

property duration

Maximum acceptable delay from the time data is written until it is received. By default, c_TimeZero.

property thisown

The membership flag

LifespanQosPolicy

class fastdds.LifespanQosPolicy

Specifies the maximum duration of validity of the data written by the DataWriter.

Notes: Mutable Qos Policy

clear()

Clears the QosPolicy object

property duration

Period of validity. By default, c_TimeInfinite.

property thisown

The membership flag

LivelinessQosPolicy

class fastdds.LivelinessQosPolicy

Determines the mechanism and parameters used by the application to determine whether an Entity is "active" (alive). The "liveliness" status of an Entity is used to maintain instance ownership in combination with the setting of the OwnershipQosPolicy. The application is also informed via listener when an Entity is no longer alive.

The DataReader requests that liveliness of the writers is maintained by the requested means and loss of liveliness is detected with delay not to exceed the lease_duration.

The DataWriter commits to signaling its liveliness using the stated means at intervals not to exceed the lease_duration. Listeners are used to notify the DataReaderof loss of liveliness and DataWriter of violations to the liveliness contract.

property announcement_period

The period for automatic assertion of liveliness. Only used for DataWriters with AUTOMATIC liveliness. By default, c_TimeInfinite.

Warning: When not infinite, must be < lease_duration, and it is advisable to be less than 0.7*lease_duration.

clear()
Clears the QosPolicy object

property kind
Liveliness kind By default, AUTOMATIC_LIVELINESS.

property lease_duration
Period within which liveliness should be asserted. On a DataWriter it represents the period it commits to signal its liveliness. On a DataReader it represents the period without assertion after which a DataWriter is considered inactive. By default, c_TimeInfinite.

property thisown
The membership flag

LivelinessQosPolicyKind

class fastdds.**AUTOMATIC_LIVELINESS_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**MANUAL_BY_PARTICIPANT_LIVELINESS_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**MANUAL_BY_TOPIC_LIVELINESS_QOS**(**args: Any*, ***kwargs: Any*)

OwnershipQosPolicy

class fastdds.**OwnershipQosPolicy**
Specifies whether it is allowed for multiple DataWriters to write the same instance of the data and if so, how these modifications should be arbitrated

Notes: Immutable Qos Policy

clear()
Clears the QosPolicy object

property kind
OwnershipQosPolicyKind

property thisown
The membership flag

OwnershipQosPolicyKind

class fastdds.**SHARED_OWNERSHIP_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**EXCLUSIVE_OWNERSHIP_QOS**(**args: Any*, ***kwargs: Any*)

OwnershipStrengthQosPolicy

class fastdds.**OwnershipStrengthQosPolicy**

Specifies the value of the "strength" used to arbitrate among multiple DataWriter objects that attempt to modify the same instance of a data-object (identified by Topic + key).This policy only applies if the OWNERSHIP QoS policy is of kind EXCLUSIVE.

Notes: Mutable Qos Policy

clear()

Clears the QosPolicy object

property thisown

The membership flag

property value

Strength By default, 0.

ParticipantResourceLimitsQos

Partition_t

class fastdds.**Partition_t**(*ptr*)

name()

Getter for the partition name

Return type string

Returns name

size()

Getter for the size

Return type int

Returns uint32_t with the size

property thisown

The membership flag

PartitionQosPolicy

class fastdds.**PartitionQosPolicy**(**args*)

Set of strings that introduces a logical partition among the topics visible by the Publisher and Subscriber. A DataWriter within a Publisher only communicates with a DataReader in a Subscriber if (in addition to matching the Topic and having compatible QoS) the Publisher and Subscriber have a common partition name string.

The empty string ("") is considered a valid partition that is matched with other partition names using the same rules of string matching and regular-expression matching used for any other partition name.

Notes: Mutable Qos Policy

begin()

Getter for the first position of the partition list

Return type eprosima::fastdds::dds::PartitionQosPolicy::const_iterator

Returns const_iterator

clear()
Clears list of partition names

empty()
Check if the set is empty

Return type int

Returns true if it is empty, false otherwise

end()
Getter for the end of the partition list

Return type eprosima::fastdds::dds::PartitionQosPolicy::const_iterator

Returns const_iterator

getNames()
Returns partition names.

Return type std::vector< std::string,std::allocator< std::string > >

Returns Vector of partition name strings.

max_size()
Getter for the maximum size (in bytes)

Return type int

Returns uint32_t with the maximum size

names(**args*)
Overload 1:

Returns partition names.

Return type std::vector< std::string,std::allocator< std::string > >

Returns Vector of partition name strings.

Overload 2:

Overrides partition names

Parameters **nam** (*std::vector< std::string,std::allocator< std::string > >*) – Vector of partition name strings.

push_back(*name*)
Appends a name to the list of partition names.

Parameters **name** (*string*) – Name to append.

setNames(*nam*)
Overrides partition names

Parameters **nam** (*std::vector< std::string,std::allocator< std::string > >*) – Vector of partition name strings.

set_max_size(*size*)
Setter for the maximum size reserved for partitions (in bytes)

Parameters **size** (*int*) – Size to be set

size()
Getter for the number of partitions

Return type int

Returns uint32_t with the size

property thisown
The membership flag

PresentationQosPolicy

class fastdds.**PresentationQosPolicy**
Specifies how the samples representing changes to data instances are presented to the subscribing application. This policy affects the application's ability to specify and receive coherent changes and to see the relative order of changes.access_scope determines the largest scope spanning the entities for which the order and coherency of changes can be preserved. The two booleans control whether coherent access and ordered access are supported within the scope access_scope.

Warning: This QosPolicy can be defined and is transmitted to the rest of the network but is not implemented in this version.

Notes: Immutable Qos Policy

property access_scope

Access Scope Kind By default, INSTANCE_PRESENTATION_QOS.

clear()
Clears the QosPolicy object

property coherent_access
Specifies support coherent access. That is, the ability to group a set of changes as a unit on the publishing end such that they are received as a unit at the subscribing end. by default, false.

property ordered_access
Specifies support for ordered access to the samples received at the subscription end. That is, the ability of the subscriber to see changes in the same order as they occurred on the publishing end. By default, false.

property thisown
The membership flag

PresentationQosPolicyAccessScopeKind

class fastdds.**INSTANCE_PRESENTATION_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**TOPIC_PRESENTATION_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**GROUP_PRESENTATION_QOS**(**args: Any*, ***kwargs: Any*)

PropertyPolicyQos

PublishModeQosPolicy

class fastdds.**PublishModeQosPolicy**
Class PublishModeQosPolicy, defines the publication mode for a specific writer.

clear()
Clears the QosPolicy object

property flow_controller_name

Name of the flow controller used when publish mode kind is ASYNCHRONOUS_PUBLISH_MODE.

Since: 2.4.0

property kind

PublishModeQosPolicyKind By default, SYNCHRONOUS_PUBLISH_MODE.

property thisown
The membership flag

PublishModeQosPolicyKind

class fastdds.**SYNCHRONOUS_PUBLISH_MODE**(**args: Any*, ***kwargs: Any*)

class fastdds.**ASYNCHRONOUS_PUBLISH_MODE**(**args: Any*, ***kwargs: Any*)

QosPolicy

class fastdds.**QosPolicy**(**args*, ***kwargs*)
Class QosPolicy, base for all QoS policies defined for Writers and Readers.

clear()
Clears the QosPolicy object

property hasChanged
Boolean that indicates if the Qos has been changed with respect to the default Qos.

send_always()
Whether it should always be sent.

Return type boolean

Returns True if it should always be sent.

property thisown
The membership flag

QosPolicyId_t

class fastdds.**INVALID_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**USERDATA_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**DURABILITY_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**PRESENTATION_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**DEADLINE_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**LATENCYBUDGET_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**OWNERSHIP_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**OWNERSHIPSTRENGTH_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**LIVELINESS_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**TIMEBASEDFILTER_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**PARTITION_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**RELIABILITY_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**DESTINATIONORDER_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**HISTORY_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**RESOURCELIMITS_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**ENTITYFACTORY_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**WRITERDATALIFECYCLE_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**READERDATALIFECYCLE_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**TOPICDATA_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**GROUPDATA_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**TRANSPORTPRIORITY_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**LIFESPAN_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**DURABILITYSERVICE_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**DATAREPRESENTATION_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**TYPECONSISTENCYENFORCEMENT_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**DISABLEPOSITIVEACKS_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**PARTICIPANTRESOURCELIMITS_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**PROPERTYPOLICY_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**PUBLISHMODE_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**READERRESOURCELIMITS_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**RTPSENDPOINT_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**RTPSRELIABLEREADER_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**RTPSRELIABLEWRITER_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**TRANSPORTCONFIG_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**TYPECONSISTENCY_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**WIREPROTOCOLCONFIG_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**WRITERRESOURCELIMITS_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

class fastdds.**NEXT_QOS_POLICY_ID**(**args: Any*, ***kwargs: Any*)

ReaderDataLifecycleQosPolicy

class fastdds.**ReaderDataLifecycleQosPolicy**

Specifies the behavior of the DataReader with regards to the lifecycle of the data-instances it manages. Warning: This Qos Policy will be implemented in future releases. Notes: Mutable Qos Policy

property autopurge_disposed_samples_delay

Indicates the duration the DataReader must retain information regarding instances that have the instance_state NOT_ALIVE_DISPOSED.

By default, c_TimeInfinite.

property autopurge_no_writer_samples_delay

Indicates the duration the DataReader must retain information regarding instances that have the instance_state NOT_ALIVE_NO_WRITERS.

By default, c_TimeInfinite.

clear()

property thisown

The membership flag

ReliabilityQosPolicy

class fastdds.**ReliabilityQosPolicy**

Indicates the reliability of the endpoint.

Notes: Immutable Qos Policy

clear()

Clears the QosPolicy object

property kind

Defines the reliability kind of the endpoint.

By default, BEST_EFFORT_RELIABILITY_QOS for DataReaders and RELIABLE_RELIABILITY_QOS for DataWriters.

property max_blocking_time

Defines the maximum period of time certain methods will be blocked.

Methods affected by this property are: - DataWriter::write - DataReader::takeNextData - DataReader::readNextData

By default, 100 ms.

property thisown

The membership flag

ReliabilityQosPolicyKind

class fastdds.**BEST_EFFORT_RELIABILITY_QOS**(**args: Any*, ***kwargs: Any*)

class fastdds.**RELIABLE_RELIABILITY_QOS**(**args: Any*, ***kwargs: Any*)

ResourceLimitsQosPolicy

class fastdds.**ResourceLimitsQosPolicy**

Specifies the resources that the Service can consume in order to meet the requested QoS

Notes: Immutable Qos Policy

property allocated_samples

Number of samples currently allocated.

By default, 100.

clear()

Clears the QosPolicy object

property extra_samples

Represents the extra number of samples available once the max_samples have been reached in the history. This makes it possible, for example, to loan samples even with a full history. By default, 1.

property max_instances

Represents the maximum number of instances DataWriter (or DataReader) can manage.

Value 0 means infinite resources. By default, 10.

Warning: It is inconsistent if *(max_instances * max_samples_per_instance)* > *max_samples*.

property max_samples

Specifies the maximum number of data-samples the DataWriter (or DataReader) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one DataWriter (or DataReader).

Value 0 means infinite resources. By default, 5000.

Warning: It is inconsistent if *max_samples* < *(max_instances * max_samples_per_instance)*.

property max_samples_per_instance

Represents the maximum number of samples of any one instance a DataWriter(or DataReader) can manage.

Value 0 means infinite resources. By default, 400.

Warning: It is inconsistent if *(max_instances * max_samples_per_instance)* > *max_samples*.

property thisown

The membership flag

RTPSEndpointQos

class fastdds.**RTPSEndpointQos**

Qos Policy to configure the endpoint

property entity_id
Entity ID, if the user wants to specify the EntityID of the endpoint. By default, -1.

property external_unicast_locators
The collection of external locators to use for communication.

property history_memory_policy
Underlying History memory policy. By default, PREALLOCATED_WITH_REALLOC_MEMORY_MODE.

property ignore_non_matching_locators
Whether locators that don't match with the announced locators should be kept.

property multicast_locator_list
Multicast locator list

property remote_locator_list
Remote locator list

property thisown
The membership flag

property unicast_locator_list
Unicast locator list

property user_defined_id
User Defined ID, used for StaticEndpointDiscovery. By default, -1.

TimeBasedFilterQosPolicy

class fastdds.**TimeBasedFilterQosPolicy**

Filter that allows a DataReader to specify that it is interested only in (potentially) a subset of the values of the data. The filter states that the DataReader does not want to receive more than one value each minimum_separation, regardless of how fast the changes occur. It is inconsistent for a DataReader to have a minimum_separation longer than its Deadline period.

Warning: This QosPolicy can be defined and is transmitted to the rest of the network but is not implemented in this version.

Notes: Mutable Qos Policy

clear()
Clears the QosPolicy object

property minimum_separation
Minimum interval between samples. By default, c_TimeZero (the DataReader is interested in all values)

property thisown
The membership flag

TopicDataQosPolicy

class fastdds.**TopicDataQosPolicy**(**args*)

property **thisown**
The membership flag

TransportConfigQos

class fastdds.**TransportConfigQos**
Qos Policy to configure the transport layer

property **builtin_transports_reception_threads_**
Thread settings for the builtin transports reception threads

clear()
Clears the QosPolicy object

property **listen_socket_buffer_size**

Listen socket buffer for all listen resources. Zero value indicates to use default system buffer size.

By default, 0.

property **max_msg_size_no_frag**

Maximum message size used to avoid fragmentation, set ONLY in LARGE_DATA. If this value is

not zero, the network factory will allow the initialization of UDP transports with maxMessageSize higher than 65500K.

property **netmask_filter**
Netmask filter configuration

property **send_socket_buffer_size**
Send socket buffer size for the send resource. Zero value indicates to use default system buffer size.

By default, 0.

property **thisown**
The membership flag

property **use_builtin_transports**

Set as false to disable the default UDPv4 implementation. By default, true.

property **user_transports**
User defined transports to use alongside or in place of builtins.

TransportPriorityQosPolicy

class fastdds.**TransportPriorityQosPolicy**

This policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data.

Warning: This QosPolicy can be defined and is transmitted to the rest of the network but is not implemented in this version.

Notes: Mutable Qos Policy

clear()

Clears the QosPolicy object

property thisown

The membership flag

property value

Priority By default, 0.

TypeConsistencyEnforcementQosPolicy

class fastdds.**TypeConsistencyEnforcementQosPolicy**

The TypeConsistencyEnforcementQosPolicy defines the rules for determining whether the type used to publish a given data stream is consistent with that used to subscribe to it. It applies to DataReaders.

Notes: Immutable Qos Policy

clear()

Clears the QosPolicy object

property m_force_type_validation

This option requires type information to be available in order to complete matching between a DataWriter and DataReader when set to TRUE, otherwise matching can occur without complete type information when set to FALSE.

By default, false.

property m_ignore_member_names

This option controls whether member names are taken into consideration for type assignability. If the option is set to TRUE, member names are considered as part of assignability in addition to member IDs (so that members with the same ID also have the same name). If the option is set to FALSE, then member names are not ignored.

By default, false.

property m_ignore_sequence_bounds

This option controls whether sequence bounds are taken into consideration for type assignability. If the option is set to TRUE, sequence bounds (maximum lengths) are not considered as part of the type assignability. This means that a T2 sequence type with maximum length L2 would be assignable to a T1 sequence type with maximum length L1, even if L2 is greater than L1. If the option is set to false, then sequence bounds are taken into consideration for type assignability and in order for T1 to be assignable from T2 it is required that L1>= L2.

By default, true.

property m_ignore_string_bounds

This option controls whether string bounds are taken into consideration for type assignability. If the option is set to TRUE, string bounds (maximum lengths) are not considered as part of the type assignability.

This means that a T2 string type with maximum length L2 would be assignable to a T1 string type with maximum length L1, even if L2 is greater than L1. If the option is set to false, then string bounds are taken into consideration for type assignability and in order for T1 to be assignable from T2 it is required that L1>= L2.

By default, true.

property m_kind

TypeConsistencyKind. By default, ALLOW_TYPE_COERCION.

property m_prevent_type_widening

This option controls whether type widening is allowed. If the option is set to FALSE, type widening is permitted. If the option is set to TRUE,it shall cause a wider type to not be assignable to a narrower type.

By default, false.

property thisown

The membership flag

TypeConsistencyKind

class fastdds.**DISALLOW_TYPE_COERCION**(**args: Any*, ***kwargs: Any*)

class fastdds.**ALLOW_TYPE_COERCION**(**args: Any*, ***kwargs: Any*)

UserDataQosPolicy

class fastdds.**UserDataQosPolicy**(**args*)

Class TClassName, base template for data qos policies. Data not known by the middleware, but distributed by means of built-in topics. By default, zero-sized sequence.

Notes: Mutable Qos Policy

property thisown

The membership flag

WireProtocolConfigQos

class fastdds.**WireProtocolConfigQos**

Qos Policy that configures the wire protocol

property builtin

Builtin parameters.

clear()

Clears the QosPolicy object

property default_external_unicast_locators

The collection of external locators to use for communication on user created topics.

property default_multicast_locator_list

Default list of Multicast Locators to be used for any Endpoint defined inside this RTPSParticipant in the case that it was defined with NO MulticastLocators. This is usually left empty.

property default_unicast_locator_list

Default list of Unicast Locators to be used for any Endpoint defined inside this RTPSParticipant in the case that it was defined with NO UnicastLocators. At least ONE locator should be included in this list.

property ignore_non_matching_locators
Whether locators that don't match with the announced locators should be kept.

property participant_id
Participant ID By default, -1.

property port
Port Parameters

property prefix
Optionally allows user to define the GuidPrefix_t

property thisown
The membership flag

WriterDataLifecycleQosPolicy

class fastdds.**WriterDataLifecycleQosPolicy**
Specifies the behavior of the DataWriter with regards to the lifecycle of the data-instances it manages. Warning: This Qos Policy will be implemented in future releases. Notes: Mutable Qos Policy

property autodispose_unregistered_instances
Controls whether a DataWriter will automatically dispose instances each time they are unregistered. The setting autodispose_unregistered_instances = TRUE indicates that unregistered instances will also be considered disposed.

By default, true.

property thisown
The membership flag

WriterResourceLimitsQos

class fastdds.**WriterResourceLimitsQos**
Qos Policy to configure the limit of the writer resources

property matched_subscriber_allocation
Matched subscribers allocation limits.

property reader_filters_allocation
Reader filters allocation limits.

property thisown
The membership flag

Status

BaseStatus

class fastdds.**BaseStatus**
A struct storing the base status

property thisown
The membership flag

property total_count
Total cumulative count

property total_count_change
Increment since the last time the status was read

DeadlineMissedStatus

class fastdds.DeadlineMissedStatus
A struct storing the deadline status

property last_instance_handle
Handle to the last instance missing the deadline

property thisown
The membership flag

property total_count
Total cumulative number of offered deadline periods elapsed during which a writer failed to provide data Missed deadlines accumulate, that is, each deadline period the total_count will be incremented by 1

property total_count_change
The change in total_count since the last time the listener was called or the status was read

IncompatibleQosStatus

class fastdds.IncompatibleQosStatus
A struct storing the requested incompatible QoS status

property last_policy_id
The id of the policy that was found to be incompatible the last time an incompatibility is detected

property policies
A list of QosPolicyCount

property thisown
The membership flag

property total_count
Total cumulative number of times the concerned writer discovered a reader for the same topic The requested QoS is incompatible with the one offered by the writer

property total_count_change
The change in total_count since the last time the listener was called or the status was read

InconsistentTopicStatus

LivelinessChangedStatus

class fastdds.LivelinessChangedStatus
A struct storing the liveliness changed status

property alive_count
The total number of currently active publishers that write the topic read by the subscriber This count increases when a newly matched publisher asserts its liveliness for the first time or when a publisher previously considered to be not alive reasserts its liveliness. The count decreases when a publisher considered alive fails to assert its liveliness and becomes not alive, whether because it was deleted normally or for some other reason

property alive_count_change
The change in the alive_count since the last time the listener was called or the status was read

property last_publication_handle
Handle to the last publisher whose change in liveliness caused this status to change

property not_alive_count
The total count of current publishers that write the topic read by the subscriber that are no longer asserting their liveliness This count increases when a publisher considered alive fails to assert its liveliness and becomes not alive for some reason other than the normal deletion of that publisher. It decreases when a previously not alive publisher either reasserts its liveliness or is deleted normally

property not_alive_count_change
The change in the not_alive_count since the last time the listener was called or the status was read

property thisown
The membership flag

MatchedStatus

class fastdds.**MatchedStatus**
A structure storing a matching status

property current_count
The number of writers currently matched to the concerned reader

property current_count_change
The change in current_count since the last time the listener was called or the status was read

property thisown
The membership flag

property total_count
Total cumulative count the concerned reader discovered a match with a writer It found a writer for the same topic with a requested QoS that is compatible with that offered by the reader

property total_count_change
The change in total_count since the last time the listener was called or the status was read

OfferedDeadlineMissedStatus

OfferedIncompatibleQosStatus

PublicationMatchedStatus

class fastdds.**PublicationMatchedStatus**
A structure storing the publication status

property last_subscription_handle
Handle to the last reader that matched the writer causing the status to change

property thisown
The membership flag

QosPolicyCount

class fastdds.**QosPolicyCount**(**args*)

A struct storing the id of the incompatible QoS Policy and the number of times it fails

property count

Total number of times that the concerned writer discovered a reader for the same topic The requested QoS is incompatible with the one offered by the writer

property policy_id

The id of the policy

property thisown

The membership flag

QosPolicyCountSeq

RequestedDeadlineMissedStatus

RequestedIncompatibleQosStatus

LivelinessLostStatus

SampleLostStatus

SampleRejectedStatus

class fastdds.**SampleRejectedStatus**

A struct storing the sample rejected status

property last_instance_handle

Handle to the instance being updated by the last sample that was rejected.

property last_reason

Reason for rejecting the last sample rejected. If no samples have been rejected, the reason is the special value NOT_REJECTED.

property thisown

The membership flag

property total_count

Total cumulative count of samples rejected by the DataReader.

property total_count_change

The incremental number of samples rejected since the last time the listener was called or the status was read.

SampleRejectedStatusKind

class fastdds.**NOT_REJECTED**(**args: Any*, ***kwargs: Any*)

class fastdds.**REJECTED_BY_INSTANCES_LIMIT**(**args: Any*, ***kwargs: Any*)

class fastdds.**REJECTED_BY_SAMPLES_LIMIT**(**args: Any*, ***kwargs: Any*)

class fastdds.**REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT**(**args: Any*, ***kwargs: Any*)

StatusMask

class fastdds.**StatusMask**(**args*)

StatusMask is a bitmap or bitset field.

This bitset is used to: - determine which listener functions to call - set conditions in dds::core::cond::StatusCondition - indicate status changes when calling dds::core::Entity::status_changes

static all()

Get all StatusMasks

Return type *StatusMask*

Returns StatusMask all

static data_available()

get the statusmask associated with dds::core::status::data_available

Return type *StatusMask*

Returns statusmask data_available

static data_on_readers()

Get the StatusMask associated with dds::core::status::data_on_readers

Return type *StatusMask*

Returns StatusMask data_on_readers

static inconsistent_topic()

Get the StatusMask associated with dds::core::status::InconsistentTopicStatus

Return type *StatusMask*

Returns StatusMask inconsistent_topic

is_active(*status*)

Checks if the status passed as parameter is 1 in the actual StatusMask :type status: *StatusMask* :param status: Status that need to be checked :rtype: boolean :return: true if the status is active and false if not

static liveliness_changed()

Get the StatusMask associated with dds::core::status::LivelinessChangedStatus

Return type *StatusMask*

Returns StatusMask liveliness_changed

static liveliness_lost()

Get the StatusMask associated with dds::core::status::LivelinessLostStatus

Return type *StatusMask*

Returns StatusMask liveliness_lost

static none()
Get no StatusMasks

Return type *StatusMask*

Returns StatusMask none

static offered_deadline_missed()
Get the StatusMask associated with dds::core::status::OfferedDeadlineMissedStatus

Return type *StatusMask*

Returns StatusMask offered_deadline_missed

static offered_incompatible_qos()
Get the StatusMask associated with dds::core::status::OfferedIncompatibleQosStatus

Return type *StatusMask*

Returns StatusMask offered_incompatible_qos

static publication_matched()
Get the statusmask associated with dds::core::status::PublicationMatchedStatus

Return type *StatusMask*

Returns StatusMask publication_matched

static requested_deadline_missed()
Get the StatusMask associated with dds::core::status::RequestedDeadlineMissedStatus

Return type *StatusMask*

Returns StatusMask requested_deadline_missed

static requested_incompatible_qos()
Get the StatusMask associated with dds::core::status::RequestedIncompatibleQosStatus

Return type *StatusMask*

Returns StatusMask requested_incompatible_qos

static sample_lost()
Get the StatusMask associated with dds::core::status::SampleLostStatus

Return type *StatusMask*

Returns StatusMask sample_lost

static sample_rejected()
Get the StatusMask associated with dds::core::status::SampleRejectedStatus

Return type *StatusMask*

Returns StatusMask sample_rejected

static subscription_matched()
Get the statusmask associated with dds::core::status::SubscriptionMatchedStatus

Return type *StatusMask*

Returns StatusMask subscription_matched

property thisown
The membership flag

SubscriptionMatchedStatus

class fastdds.**SubscriptionMatchedStatus**
A structure storing the subscription status

property **last_publication_handle**
Handle to the last writer that matched the reader causing the status change

property **thisown**
The membership flag

LoanableArray

LoanableCollection

class fastdds.**LoanableCollection**(**args*, ***kwargs*)
A collection of generic opaque pointers that can receive the buffer from outside (loan).

This is an abstract class. See 'LoanableSequence' for details.

buffer()
Get the pointer to the elements buffer.

The returned value may be nullptr if maximum() is 0. Otherwise it is guaranteed that up to maximum() elements can be accessed.

Return type void

Returns the pointer to the elements buffer.

has_ownership()
Get the ownership flag.

Return type boolean

Returns whether the collection has ownership of the buffer.

length(**args*)
Overload 1:

Get the number of elements currently accessible.

Return type int

Returns the number of elements currently accessible.

Overload 2:

Set the number of elements currently accessible.

This method tells the collection that a certain number of elements should be accessible. If the new length is greater than the current 'maximum()' the collection should allocate space for the new elements. If this is the case and the collection does not own the buffer (i.e. 'has_ownership()' is false) then no allocation will be performed, the length will remain unchanged, and false will be returned.

Parameters **[in]** – new_length New number of elements to be accessible.

Return type boolean

Returns true if the new length was correctly set.

`loan`(*buffer*, *new_maximum*, *new_length*)
Loan a buffer to the collection.

Parameters

- `[in]` – buffer pointer to the buffer to be loaned.
- `[in]` – new_maximum number of allocated elements in buffer.
- `[in]` – new_length number of accessible elements in buffer.

Return type boolean

Returns false if preconditions are not met.

Return type boolean

Returns true if operation succeeds.

`maximum`()
Get the maximum number of elements currently allocated.

Return type int

Returns the maximum number of elements currently allocated.

`property thisown`
The membership flag

`unloan`(**args*)
Overload 1:

Remove the loan from the collection.

Parameters

- `[out]` – maximum number of allocated elements on the returned buffer.
- `[out]` – length number of accessible elements on the returned buffer.

Return type void

Returns nullptr if preconditions are not met.

Return type void

Returns pointer to the previously loaned buffer of elements.

Overload 2:

Remove the loan from the collection.

Return type void

Returns nullptr if preconditions are not met.

Return type void

Returns pointer to the previously loaned buffer of elements.

LoanableSequence

StackAllocatedSequence

Domain

DomainParticipant

class fastdds.**DomainParticipant**(**args*, ***kwargs*)

Class DomainParticipant used to group Publishers and Subscribers into a single working unit.

assert_liveliness()

This operation manually asserts the liveliness of the DomainParticipant. This is used in combination with the LIVELINESS QoS policy to indicate to the Service that the entity remains active.

This operation needs to only be used if the DomainParticipant contains DataWriter entities with the LIVELINESS set to MANUAL_BY_PARTICIPANT and it only affects the liveliness of those DataWriter entities. Otherwise, it has no effect.

Notes: Writing data via the write operation on a DataWriter asserts liveliness on the DataWriter itself and its DomainParticipant. Consequently the use of assert_liveliness is only needed if the application is not writing data regularly.

Return type ReturnCode_t

Returns RETCODE_OK if the liveliness was asserted, RETCODE_ERROR otherwise.

contains_entity(*a_handle*, *recursive=True*)

This operation checks whether or not the given handle represents an Entity that was created from the DomainParticipant.

Parameters

- **a_handle** (InstanceHandle_t) – InstanceHandle of the entity to look for.
- **recursive** (*boolean*) – The containment applies recursively. That is, it applies both to entities (TopicDescription, Publisher, or Subscriber) created directly using the DomainParticipant as well as entities created using a contained Publisher, or Subscriber as the factory, and so forth. (default: true)

Return type boolean

Returns True if entity is contained. False otherwise.

create_contentfilteredtopic(**args*)

Overload 1:

Create a ContentFilteredTopic in this Participant.

Parameters

- **name** (*string*) – Name of the ContentFilteredTopic
- **related_topic** (*Topic*) – Related Topic to being subscribed
- **filter_expression** (*string*) – Logic expression to create filter
- **expression_parameters** (*std::vector< std::string,std::allocator< std::string > >*) – Parameters to filter content

Return type ContentFilteredTopic

Returns Pointer to the created ContentFilteredTopic.

Return type ContentFilteredTopic

Returns nullptr if related_topic does not belong to this participant.

Return type ContentFilteredTopic

Returns nullptr if a topic with the specified name has already been created.

Return type ContentFilteredTopic

Returns nullptr if a filter cannot be created with the specified filter_expression and expression_parameters.

Overload 2:

Create a ContentFilteredTopic in this Participant using a custom filter.

Parameters

- **name** (*string*) – Name of the ContentFilteredTopic
- **related_topic** (*Topic*) – Related Topic to being subscribed
- **filter_expression** (*string*) – Logic expression to create filter
- **expression_parameters** (*std::vector< std::string,std::allocator< std::string > >*) – Parameters to filter content
- **filter_class_name** (*string*) – Name of the filter class to use

Return type ContentFilteredTopic

Returns Pointer to the created ContentFilteredTopic.

Return type ContentFilteredTopic

Returns nullptr if related_topic does not belong to this participant.

Return type ContentFilteredTopic

Returns nullptr if a topic with the specified name has already been created.

Return type ContentFilteredTopic

Returns nullptr if a filter cannot be created with the specified filter_expression and expression_parameters.

Return type ContentFilteredTopic

Returns nullptr if the specified filter_class_name has not been registered.

create_multitopic(*name*, *type_name*, *subscription_expression*, *expression_parameters*)

Create a MultiTopic in this Participant.

Parameters

- **name** (*string*) – Name of the MultiTopic
- **type_name** (*string*) – Result type of the MultiTopic
- **subscription_expression** (*string*) – Logic expression to combine filter
- **expression_parameters** (*std::vector< std::string,std::allocator< std::string > >*) – Parameters to subscription content

Return type eprosima::fastdds::dds::MultiTopic

Returns Pointer to the created ContentFilteredTopic, nullptr in error case

create_publisher(**args*)
Create a Publisher in this Participant.

Parameters

- **qos** (*PublisherQos*) – QoS of the Publisher.
- **listener** (*PublisherListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all)

Return type *Publisher*

Returns Pointer to the created Publisher.

create_publisher_with_profile(**args*)
Create a Publisher in this Participant.

Parameters

- **profile_name** (*string*) – Publisher profile name.
- **listener** (*PublisherListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all)

Return type *Publisher*

Returns Pointer to the created Publisher.

create_subscriber(**args*)
Create a Subscriber in this Participant.

Parameters

- **qos** (*SubscriberQos*) – QoS of the Subscriber.
- **listener** (*SubscriberListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all)

Return type *Subscriber*

Returns Pointer to the created Subscriber.

create_subscriber_with_profile(**args*)
Create a Subscriber in this Participant.

Parameters

- **profile_name** (*string*) – Subscriber profile name.
- **listener** (*SubscriberListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all)

Return type *Subscriber*

Returns Pointer to the created Subscriber.

create_topic(**args*)
Create a Topic in this Participant.

Parameters

- **topic_name** (*string*) – Name of the Topic.
- **type_name** (*string*) – Data type of the Topic.
- **qos** (*TopicQos*) – QoS of the Topic.
- **listener** (*TopicListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all)

Return type *Topic*

Returns Pointer to the created Topic.

create_topic_with_profile(**args*)
Create a Topic in this Participant.

Parameters

- **topic_name** (*string*) – Name of the Topic.
- **type_name** (*string*) – Data type of the Topic.
- **profile_name** (*string*) – Topic profile name.
- **listener** (*TopicListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all)

Return type *Topic*

Returns Pointer to the created Topic.

delete_contained_entities()
Deletes all the entities that were created by means of the "create" methods

Return type ReturnCode_t

Returns RETURN_OK code if everything correct, error code otherwise

delete_contentfilteredtopic(*a_contentfilteredtopic*)
Deletes an existing ContentFilteredTopic.

Parameters **a_contentfilteredtopic** (ContentFilteredTopic) – ContentFilteredTopic to be deleted

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the topic passed is a nullptr, RETCODE_PRECONDITION_NOT_MET if the topic does not belong to this participant or if it is referenced by any entity and RETCODE_OK if the ContentFilteredTopic was deleted.

delete_multitopic(*a_multitopic*)
Deletes an existing MultiTopic.

Parameters **a_multitopic** (*eprosima::fastdds::dds::MultiTopic*) – MultiTopic to be deleted

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the topic passed is a nullptr, RETCODE_PRECONDITION_NOT_MET if the topic does not belong to this participant or if it is referenced by any entity and RETCODE_OK if the Topic was deleted.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

delete_publisher(*publisher*)
Deletes an existing Publisher.

Parameters **publisher** (*Publisher*) – to be deleted.

Return type ReturnCode_t

Returns RETCODE_PRECONDITION_NOT_MET if the publisher does not belong to this participant or if it has active DataWriters, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

delete_subscriber(*subscriber*)
Deletes an existing Subscriber.

Parameters **subscriber** (*Subscriber*) – to be deleted.

Return type ReturnCode_t

Returns RETCODE_PRECONDITION_NOT_MET if the subscriber does not belong to this participant or if it has active DataReaders, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

delete_topic(*topic*)
Deletes an existing Topic.

Parameters **topic** (*Topic*) – to be deleted.

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the topic passed is a nullptr, RETCODE_PRECONDITION_NOT_MET if the topic does not belong to this participant or if it is referenced by any entity and RETCODE_OK if the Topic was deleted.

enable()
This operation enables the DomainParticipant

Return type ReturnCode_t

Returns RETCODE_OK

find_topic(*topic_name*, *timeout*)
Gives access to an existing (or ready to exist) enabled Topic. It should be noted that the returned Topic is a local object that acts as a proxy to designate the global concept of topic. Topics obtained by means of find_topic, must also be deleted by means of delete_topic so that the local resources can be released. If a Topic is obtained multiple times by means of find_topic or create_topic, it must also be deleted that same number of times using delete_topic.

Parameters

- **topic_name** (*string*) – Topic name
- **timeout** (Duration_t) – Maximum time to wait for the Topic

Return type *Topic*

Returns Pointer to the existing Topic, nullptr in case of error or timeout

find_type(*type_name*)
This method gives access to a registered type based on its name.

Parameters **type_name** (*string*) – Name of the type

Return type *TypeSupport*

Returns TypeSupport corresponding to the type_name

get_builtin_subscriber()
Allows access to the builtin Subscriber.

Return type *Subscriber*

Returns Pointer to the builtin Subscriber, nullptr in error case

get_current_time(*current_time*)
This operation returns the current value of the time that the service uses to time-stamp data-writes and to set the reception-timestamp for the data-updates it receives.

Parameters **current_time** (Time_t) – Time_t reference where the current time is returned

Return type ReturnCode_t

Returns RETCODE_OK

get_default_publisher_qos(**args*)
Overload 1:

This operation retrieves the default value of the Publisher QoS, that is, the QoS policies which will be used for newly created Publisher entities in the case where the QoS policies are defaulted in the create_publisher operation.

The values retrieved get_default_publisher_qos will match the set of values specified on the last successful call to set_default_publisher_qos, or else, if the call was never made, the default values.

Return type *PublisherQos*

Returns Current default publisher qos.

Overload 2:

This operation retrieves the default value of the Publisher QoS, that is, the QoS policies which will be used for newly created Publisher entities in the case where the QoS policies are defaulted in the create_publisher operation.

The values retrieved get_default_publisher_qos will match the set of values specified on the last successful call to set_default_publisher_qos, or else, if the call was never made, the default values.

Parameters **qos** (*PublisherQos*) – PublisherQos reference where the default_publisher_qos is returned

Return type ReturnCode_t

Returns RETCODE_OK

get_default_subscriber_qos(**args*)
Overload 1:

This operation retrieves the default value of the Subscriber QoS, that is, the QoS policies which will be used for newly created Subscriber entities in the case where the QoS policies are defaulted in the create_subscriber operation.

The values retrieved get_default_subscriber_qos will match the set of values specified on the last successful call to set_default_subscriber_qos, or else, if the call was never made, the default values.

Return type *SubscriberQos*

Returns Current default subscriber qos.

Overload 2:

This operation retrieves the default value of the Subscriber QoS, that is, the QoS policies which will be used for newly created Subscriber entities in the case where the QoS policies are defaulted in the create_subscriber operation.

The values retrieved get_default_subscriber_qos will match the set of values specified on the last successful call to set_default_subscriber_qos, or else, if the call was never made, the default values.

Parameters **qos** (*SubscriberQos*) – SubscriberQos reference where the default_subscriber_qos is returned

Return type ReturnCode_t

Returns RETCODE_OK

get_default_topic_qos(**args*)

Overload 1:

This operation retrieves the default value of the Topic QoS, that is, the QoS policies that will be used for newly created Topic entities in the case where the QoS policies are defaulted in the create_topic operation.

The values retrieved get_default_topic_qos will match the set of values specified on the last successful call to set_default_topic_qos, or else, TOPIC_QOS_DEFAULT if the call was never made.

Return type *TopicQos*

Returns Current default topic qos.

Overload 2:

This operation retrieves the default value of the Topic QoS, that is, the QoS policies that will be used for newly created Topic entities in the case where the QoS policies are defaulted in the create_topic operation.

The values retrieved get_default_topic_qos will match the set of values specified on the last successful call to set_default_topic_qos, or else, TOPIC_QOS_DEFAULT if the call was never made.

Parameters **qos** (*TopicQos*) – TopicQos reference where the default_topic_qos is returned

Return type ReturnCode_t

Returns RETCODE_OK

get_discovered_participant_data(*participant_data*, *participant_handle*)

Retrieves the DomainParticipant data of a discovered not ignored participant.

participant_data Reference to the ParticipantBuiltinTopicData object to return the data :type participant_handle: InstanceHandle_t :param participant_handle: InstanceHandle of DomainParticipant to

retrieve the data from :rtype: ReturnCode_t :return: RETCODE_OK if everything correct, PRECONDITION_NOT_MET if participant does not exist Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_discovered_participants(*participant_handles*)
Retrieves the list of DomainParticipants that have been discovered in the domain and are not "ignored".

participant_handles Reference to the vector where discovered participants will be returned :rtype: ReturnCode_t :return: RETCODE_OK if everything correct, error code otherwise Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_discovered_topic_data(*topic_data*, *topic_handle*)
Retrieves the Topic data of a discovered not ignored topic.

topic_data Reference to the TopicBuiltinTopicData object to return the data :type topic_handle: InstanceHandle_t :param topic_handle: InstanceHandle of Topic to retrieve the data from :rtype: ReturnCode_t :return: RETCODE_OK if everything correct, PRECONDITION_NOT_MET if topic does not exist

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_discovered_topics(*topic_handles*)
Retrieves the list of topics that have been discovered in the domain and are not "ignored".

topic_handles Reference to the vector where discovered topics will be returned :rtype: ReturnCode_t :return: RETCODE_OK if everything correct, error code otherwise Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_domain_id()
This operation retrieves the domain_id used to create the DomainParticipant. The domain_id identifies the DDS domain to which the DomainParticipant belongs.

Return type int

Returns The Participant's domain_id

get_instance_handle()
Returns the DomainParticipant's handle.

Return type InstanceHandle_t

Returns InstanceHandle of this DomainParticipant.

get_listener()
Allows accessing the DomainParticipantListener.

Return type *DomainParticipantListener*

Returns DomainParticipantListener pointer

get_participant_names()
Getter for the participant names

Return type std::vector< std::string,std::allocator< std::string > >

Returns Vector with the names

get_publisher_qos_from_profile(*profile_name*, *qos*)
Fills the PublisherQos with the values of the XML profile.

Parameters

- **profile_name** (*string*) – Publisher profile name.
- **qos** (*PublisherQos*) – PublisherQos object where the qos is returned.

Return type ReturnCode_t

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

get_qos(**args*)

Overload 1:

This operation returns the value of the DomainParticipant QoS policies

Parameters **qos** (*DomainParticipantQos*) – DomainParticipantQos reference where the qos is going to be returned

Return type ReturnCode_t

Returns RETCODE_OK

Overload 2:

This operation returns the value of the DomainParticipant QoS policies

Return type *DomainParticipantQos*

Returns A reference to the DomainParticipantQos

get_resource_event()

Getter for the resource event

Return type eprosima::fastrtps::rtps::ResourceEvent

Returns A reference to the resource event

get_subscriber_qos_from_profile(*profile_name*, *qos*)

Fills the SubscriberQos with the values of the XML profile.

Parameters

- **profile_name** (*string*) – Subscriber profile name.
- **qos** (*SubscriberQos*) – SubscriberQos object where the qos is returned.

Return type ReturnCode_t

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

get_topic_qos_from_profile(*profile_name*, *qos*)

Fills the TopicQos with the values of the XML profile.

Parameters

- **profile_name** (*string*) – Topic profile name.
- **qos** (*TopicQos*) – TopicQos object where the qos is returned.

Return type ReturnCode_t

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

get_type_dependencies(*_in*)

When a DomainParticipant receives an incomplete list of TypeIdentifiers in a PublicationBuiltinTopicData or SubscriptionBuiltinTopicData, it may request the additional type dependencies by invoking the getTypeDependencies operation.

Parameters **in** (*eprosima::fastrtps::types::TypeIdentifierSeq*) – TypeIdentifier sequence

Return type SampleIdentity

Returns SampleIdentity

get_types(*_in*)

A DomainParticipant may invoke the operation getTypes to retrieve the TypeObjects associated with a list of TypeIdentifiers.

Parameters **in** (*eprosima::fastrtps::types::TypeIdentifierSeq*) – TypeIdentifier sequence

Return type SampleIdentity

Returns SampleIdentity

guid()

Getter for the Participant GUID

Return type GUID_t

Returns A reference to the GUID

ignore_participant(*handle*)

Locally ignore a remote domain participant.

Notes: This action is not reversible.

Parameters **handle** (InstanceHandle_t) – Identifier of the remote participant to ignore

Return type ReturnCode_t

Returns RETURN_OK code if everything correct, RETCODE_BAD_PARAMENTER otherwise

ignore_publication(*handle*)

Locally ignore a remote datawriter.

Notes: This action is not reversible.

Parameters **handle** (InstanceHandle_t) – Identifier of the datawriter to ignore

Return type ReturnCode_t

Returns RETURN_OK code if everything correct, error code otherwise

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

ignore_subscription(*handle*)

Locally ignore a remote datareader.

Notes: This action is not reversible.

Parameters **handle** (InstanceHandle_t) – Identifier of the datareader to ignore

Return type ReturnCode_t

Returns RETURN_OK code if everything correct, error code otherwise

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

ignore_topic(*handle*)

Locally ignore a topic.

Notes: This action is not reversible.

Parameters **handle** (InstanceHandle_t) – Identifier of the topic to ignore

Return type ReturnCode_t

Returns RETURN_OK code if everything correct, error code otherwise

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

lookup_content_filter_factory(*filter_class_name*)

Lookup a custom content filter factory previously registered with register_content_filter_factory.

Parameters **filter_class_name** (*string*) – Name of the filter class. Cannot be nullptr.

Return type IContentFilterFactory

Returns nullptr if the given filter_class_name has not been previously registered on this DomainParticipant. Otherwise, the content filter factory previously registered with the given filter_class_name.

lookup_topicdescription(*topic_name*)

Looks up an existing, locally created 'TopicDescription', based on its name. May be called on a disabled participant.

Parameters **topic_name** (*string*) – Name of the 'TopicDescription' to search for.

Return type *TopicDescription*

Returns Pointer to the topic description, if it has been created locally. Otherwise, nullptr is returned.

Remarks: UNSAFE. It is unsafe to lookup a topic description while another thread is creating a topic.

new_remote_endpoint_discovered(*partguid*, *userId*, *kind*)

This method can be used when using a StaticEndpointDiscovery mechanism different that the one included in FastRTPS, for example when communicating with other implementations. It indicates the Participant that an Endpoint from the XML has been discovered and should be activated.

Parameters

- **partguid** (GUID_t) – Participant GUID_t.
- **userId** (*int*) – User defined ID as shown in the XML file.
- **kind** (*int*) – EndpointKind (WRITER or READER)

Return type boolean

Returns True if correctly found and activated.

register_content_filter_factory(*filter_class_name*, *filter_factory*)

Register a custom content filter factory, which can be used to create a ContentFilteredTopic.

DDS specifies a SQL-like content filter to be used by content filtered topics. If this filter does not meet your filtering requirements, you can register a custom filter factory.

To use a custom filter, a factory for it must be registered in the following places:

- In any application that uses the custom filter factory to create a ContentFilteredTopic and the corresponding DataReader.
- In each application that writes the data to the applications mentioned above.

For example, suppose Application A on the subscription side creates a Topic named X and a ContentFilteredTopic named filteredX (and a corresponding DataReader), using a previously registered content filter factory, myFilterFactory. With only that, you will have filtering at the subscription side. If you also

want to perform filtering in any application that publishes Topic X, then you also need to register the same definition of the ContentFilterFactory myFilterFactory in that application.

Each `filter_class_name` can only be used to register a content filter factory once per DomainParticipant.

Parameters

- **filter_class_name** (*string*) – Name of the filter class. Cannot be nullptr, must not exceed 255 characters, and must be unique within this DomainParticipant.
- **filter_factory** (`IContentFilterFactory`) – Factory of content filters to be registered. Cannot be nullptr.

Return type `ReturnCode_t`

Returns RETCODE_BAD_PARAMETER if any parameter is nullptr, or the filter_class_name exceeds 255 characters.

Return type `ReturnCode_t`

Returns RETCODE_PRECONDITION_NOT_MET if the filter_class_name has been already registered.

Return type `ReturnCode_t`

Returns RETCODE_PRECONDITION_NOT_MET if filter_class_name is FASTDDS_SQLFILTER_NAME.

Return type `ReturnCode_t`

Returns RETCODE_OK if the filter is correctly registered.

register_remote_type(*type_information*, *type_name*, *callback*)

Helps the user to solve all dependencies calling internally to the type lookup service and registers the resulting dynamic type. The registration may be perform asynchronously, case in which the user will be notified through the given callback, which receives the type_name as unique argument.

Parameters

- **type_information** (*eprosima::fastrtps::types::TypeInformation*) –
- **type_name** (*string*) –
- **callback** (*std::function< void (std::string const &, eprosima::fastrtps::types::DynamicType_ptr const) >*) –

Return type `ReturnCode_t`

Returns RETCODE_OK If the given type_information is enough to build the type without using the typelookup service (callback will not be called).

Return type `ReturnCode_t`

Returns RETCODE_OK if the given type is already available (callback will not be called).

Return type `ReturnCode_t`

Returns RETCODE_NO_DATA if type is not available yet (the callback will be called if negotiation is success, and ignored in other case).

Return type `ReturnCode_t`

Returns RETCODE_NOT_ENABLED if the DomainParticipant is not enabled.

Return type `ReturnCode_t`

Returns RETCODE_PRECONDITION_NOT_MET if the DomainParticipant type lookup service is disabled.

register_type(**args*)

Overload 1:

Register a type in this participant.

Parameters

- **type** (*TypeSupport*) – TypeSupport.
- **type_name** (*string*) – The name that will be used to identify the Type.

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there is another TypeSupport with the same name and RETCODE_OK if it is correctly registered.

Overload 2:

Register a type in this participant.

Parameters **type** (*TypeSupport*) – TypeSupport.

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there is another TypeSupport with the same name and RETCODE_OK if it is correctly registered.

set_default_publisher_qos(*qos*)

This operation sets a default value of the Publisher QoS policies which will be used for newly created Publisher entities in the case where the QoS policies are defaulted in the create_publisher operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value PUBLISHER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_publisher_qos operation had never been called.

Parameters **qos** (*PublisherQos*) – PublisherQos to be set

Return type ReturnCode_t

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

set_default_subscriber_qos(*qos*)

This operation sets a default value of the Subscriber QoS policies that will be used for newly created Subscriber entities in the case where the QoS policies are defaulted in the create_subscriber operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value SUBSCRIBER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_subscriber_qos operation had never been called.

Parameters **qos** (*SubscriberQos*) – SubscriberQos to be set

Return type ReturnCode_t

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

set_default_topic_qos(*qos*)

This operation sets a default value of the Topic QoS policies which will be used for newly created Topic entities in the case where the QoS policies are defaulted in the create_topic operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value TOPIC_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_topic_qos operation had never been called.

Parameters **qos** (*TopicQos*) – TopicQos to be set

Return type ReturnCode_t

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

set_listener(**args*)

Overload 1:

Modifies the DomainParticipantListener, sets the mask to StatusMask::all()

Parameters **listener** (*DomainParticipantListener*) – new value for the DomainParticipantListener

Return type ReturnCode_t

Returns RETCODE_OK

Overload 2:

Modifies the DomainParticipantListener.

Parameters

- **listener** (*DomainParticipantListener*) – new value for the DomainParticipantListener
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to

Return type ReturnCode_t

Returns RETCODE_OK

set_qos(*qos*)

This operation sets the value of the DomainParticipant QoS policies.

Parameters **qos** (*DomainParticipantQos*) – DomainParticipantQos to be set

Return type ReturnCode_t

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

property thisown
The membership flag

unregister_content_filter_factory(*filter_class_name*)
Unregister a custom content filter factory previously registered with register_content_filter_factory.

A filter_class_name can be unregistered only if it has been previously registered to the DomainParticipant with register_content_filter_factory.

The unregistration of filter is not allowed if there are any existing ContentFilteredTopic objects that are using the filter.

If there is any existing discovered DataReader with the same filter_class_name, filtering on the writer side will be stopped, but this operation will not fail.

Parameters **filter_class_name** (*string*) – Name of the filter class. Cannot be nullptr.

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the filter_class_name is nullptr.

Return type ReturnCode_t

Returns RERCODE_PRECONDITION_NOT_MET if the filter_class_name has not been previously registered.

Return type ReturnCode_t

Returns RERCODE_PRECONDITION_NOT_MET if there is any ContentFilteredTopic referencing the filter.

Return type ReturnCode_t

Returns RETCODE_OK if the filter is correctly unregistered.

unregister_type(*typeName*)
Unregister a type in this participant.

Parameters **typeName** (*string*) – Name of the type

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there are entities using that TypeSupport and RETCODE_OK if it is correctly unregistered.

DomainParticipantFactory

class fastdds.**DomainParticipantFactory**(**args*, ***kwargs*)
Class DomainParticipantFactory

check_xml_static_discovery(*xml_file*)
Check the validity of the provided static discovery XML file

Parameters **xml_file** (*string*) – xml file path

Return type ReturnCode_t

Returns RETCODE_OK if the validation is successful, RETCODE_ERROR otherwise.

create_participant(*args)
Create a Participant.

Parameters

- **domain_id** (*int*) – Domain Id.
- **qos** (*DomainParticipantQos*) – DomainParticipantQos Reference.
- **listener** (*DomainParticipantListener*) – DomainParticipantListener Pointer (default: nullptr)
- **mask** (*StatusMask*) – StatusMask Reference (default: all)

Return type *DomainParticipant*

Returns DomainParticipant pointer. (nullptr if not created.)

create_participant_with_profile(*args)
Overload 1:

Create a Participant.

Parameters

- **domain_id** (*int*) – Domain Id.
- **profile_name** (*string*) – Participant profile name.
- **listener** (*DomainParticipantListener*) – DomainParticipantListener Pointer (default: nullptr)
- **mask** (*StatusMask*) – StatusMask Reference (default: all)

Return type *DomainParticipant*

Returns DomainParticipant pointer. (nullptr if not created.)

Overload 2:

Create a Participant.

Parameters

- **profile_name** (*string*) – Participant profile name.
- **listener** (*DomainParticipantListener*) – DomainParticipantListener Pointer (default: nullptr)
- **mask** (*StatusMask*) – StatusMask Reference (default: all)

Return type *DomainParticipant*

Returns DomainParticipant pointer. (nullptr if not created.)

Overload 3:

Create a Participant.

Parameters

- **profile_name** (*string*) – Participant profile name.
- **listener** (*DomainParticipantListener*) – DomainParticipantListener Pointer (default: nullptr)
- **mask** – StatusMask Reference (default: all)

Return type *DomainParticipant*

Returns DomainParticipant pointer. (nullptr if not created.)

Overload 4:

Create a Participant.

Parameters

- **profile_name** (*string*) – Participant profile name.
- **listener** – DomainParticipantListener Pointer (default: nullptr)
- **mask** – StatusMask Reference (default: all)

Return type *DomainParticipant*

Returns DomainParticipant pointer. (nullptr if not created.)

delete_participant(*part*)

Remove a Participant and all associated publishers and subscribers.

Parameters **part** (*DomainParticipant*) – Pointer to the participant.

Return type ReturnCode_t

Returns RETCODE_PRECONDITION_NOT_MET if the participant has active entities, RETCODE_OK if the participant is correctly deleted and RETCODE_ERROR otherwise.

get_default_participant_qos(**args*)

Overload 1:

This operation retrieves the default value of the DomainParticipant QoS, that is, the QoS policies which will be used for newly created DomainParticipant entities in the case where the QoS policies are defaulted in the create_participant operation. The values retrieved get_default_participant_qos will match the set of values specified on the last successful call to set_default_participant_qos, or else, if the call was never made, the default values.

Parameters **qos** (*DomainParticipantQos*) – DomainParticipantQos where the qos is returned

Return type ReturnCode_t

Returns RETCODE_OK

Overload 2:

This operation retrieves the default value of the DomainParticipant QoS, that is, the QoS policies which will be used for newly created DomainParticipant entities in the case where the QoS policies are defaulted in the create_participant operation. The values retrieved get_default_participant_qos will match the set of values specified on the last successful call to set_default_participant_qos, or else, if the call was never made, the default values.

Return type *DomainParticipantQos*

Returns A reference to the default DomainParticipantQos

static get_instance()
Returns the DomainParticipantFactory singleton instance.

Return type *DomainParticipantFactory*

Returns A raw pointer to the DomainParticipantFactory singleton instance.

get_participant_qos_from_profile(*profile_name*, *qos*)
Fills the DomainParticipantQos with the values of the XML profile.

Parameters

- **profile_name** (*string*) – DomainParticipant profile name.
- **qos** (*DomainParticipantQos*) – DomainParticipantQos object where the qos is returned.

Return type ReturnCode_t

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

get_qos(*qos*)
This operation returns the value of the DomainParticipantFactory QoS policies.

Parameters **qos** (*DomainParticipantFactoryQos*) – DomaParticipantFactoryQos reference where the qos is returned

Return type ReturnCode_t

Returns RETCODE_OK

static get_shared_instance()
Returns the DomainParticipantFactory singleton instance.

Return type std::shared_ptr< eprosima::fastdds::dds::DomainParticipantFactory >

Returns A shared pointer to the DomainParticipantFactory singleton instance.

load_XML_profiles_file(*xml_profile_file*)
Load profiles from XML file.

Parameters **xml_profile_file** (*string*) – XML profile file.

Return type ReturnCode_t

Returns RETCODE_OK if it is correctly loaded, RETCODE_ERROR otherwise.

load_XML_profiles_string(*data*, *length*)
Load profiles from XML string.

Parameters

- **data** (*string*) – buffer containing xml data.
- **length** (*int*) – length of data

Return type ReturnCode_t

Returns RETCODE_OK if it is correctly loaded, RETCODE_ERROR otherwise.

load_profiles()
Load profiles from default XML file.

Return type ReturnCode_t

Returns RETCODE_OK

lookup_participant(*domain_id*)
This operation retrieves a previously created DomainParticipant belonging to specified domain_id. If no such DomainParticipant exists, the operation will return 'nullptr'. If multiple DomainParticipant entities belonging to that domain_id exist, then the operation will return one of them. It is not specified which one.

Parameters **domain_id** (*int*) –

Return type *DomainParticipant*

Returns previously created DomainParticipant within the specified domain

lookup_participants(*domain_id*)
Returns all participants that belongs to the specified domain_id.

Parameters **domain_id** (*int*) –

Return type std::vector< eprosima::fastdds::dds::DomainParticipant *,std::allocator< eprosima::fastdds::dds::DomainParticipant * > >

Returns previously created DomainParticipants within the specified domain

set_default_participant_qos(*qos*)
This operation sets a default value of the DomainParticipant QoS policies which will be used for newly created DomainParticipant entities in the case where the QoS policies are defaulted in the create_participant operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value PARTICIPANT_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_participant_qos operation had never been called.

Parameters **qos** (*DomainParticipantQos*) – DomainParticipantQos to be set

Return type ReturnCode_t

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

set_qos(*qos*)
This operation sets the value of the DomainParticipantFactory QoS policies. These policies control the behavior of the object a factory for entities.

Note that despite having QoS, the DomainParticipantFactory is not an Entity.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

Parameters **qos** (*DomainParticipantFactoryQos*) – DomainParticipantFactoryQos to be set.

Return type ReturnCode_t

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

property thisown
The membership flag

DomainParticipantFactoryQos

class fastdds.**DomainParticipantFactoryQos**
Class DomainParticipantFactoryQos, contains all the possible Qos that can be set for a determined participant. Please consult each of them to check for implementation details and default values.

entity_factory(**args*)
Overload 1:

Getter for EntityFactoryQosPolicy :rtype: *EntityFactoryQosPolicy* :return: EntityFactoryQosPolicy reference

Overload 2:

Getter for EntityFactoryQosPolicy :rtype: *EntityFactoryQosPolicy* :return: EntityFactoryQosPolicy reference

Overload 3:

Setter for EntityFactoryQosPolicy :type entity_factory: *EntityFactoryQosPolicy* :param entity_factory: EntityFactoryQosPolicy

file_watch_threads(**args*)
Overload 1:

Getter for file watch related ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 2:

Getter for file watch related ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 3:

Setter for the file watch related ThreadSettings

Parameters **value** (ThreadSettings) – New ThreadSettings to be set

shm_watchdog_thread(**args*)

Overload 1:

Getter for SHM watchdog ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 2:

Getter for SHM watchdog ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 3:

Setter for the SHM watchdog ThreadSettings

Parameters **value** (ThreadSettings) – New ThreadSettings to be set

property thisown

The membership flag

DomainParticipantListener

class fastdds.**DomainParticipantListener**

Class DomainParticipantListener, overrides behaviour towards certain events.

on_participant_discovery(**args*)

This method is called when a new Participant is discovered, or a previously discovered participant changes its QOS or is removed.

participant Pointer to the Participant which discovered the remote participant. info Remote participant information. User can take ownership of the object. should_be_ignored Flag to indicate the library to automatically ignore the discovered Participant.

on_publisher_discovery(**args*)

Overload 1:

This method is called when a new DataWriter is discovered, or a previously discovered DataWriter changes its QOS or is removed.

participant Pointer to the Participant which discovered the remote DataWriter. info Remote DataWriter information. User can take ownership of the object.

Overload 2:

This method is called when a new DataWriter is discovered, or a previously discovered DataWriter changes its QOS or is removed.

Warning: Not Supported. This callback will never be called in the current version.

participant Pointer to the Participant which discovered the remote DataWriter. info Remote DataWriter information. User can take ownership of the object. should_be_ignored Flag to indicate the library to automatically ignore the discovered DataWriter.

on_subscriber_discovery(**args*)

Overload 1:

This method is called when a new DataReader is discovered, or a previously discovered DataReader changes its QOS or is removed.

participant Pointer to the Participant which discovered the remote DataReader. info Remote DataReader information. User can take ownership of the object.

Overload 2:

This method is called when a new DataReader is discovered, or a previously discovered DataReader changes its QOS or is removed.

Warning: Not Supported. This callback will never be called in the current version.

participant Pointer to the Participant which discovered the remote DataReader. info Remote DataReader information. User can take ownership of the object. should_be_ignored Flag to indicate the library to automatically ignore the discovered DataReader.

on_type_dependencies_reply(*participant*, *request_sample_id*, *dependencies*)

This method is called when the typelookup client received a reply to a getTypeDependencies request. The user may want to retrieve these new types using the getTypes request and create a new DynamicType using the retrieved TypeObject.

on_type_discovery(*participant*, *request_sample_id*, *topic*, *identifier*, *object*, *dyn_type*)

This method is called when a participant discovers a new Type The ownership of all object belongs to the caller so if needs to be used after the method ends, a full copy should be perform (except for dyn_type due to its shared_ptr nature. For example: fastrtps::types::TypeIdentifier new_type_id = *identifier;

on_type_information_received(*participant*, *topic_name*, *type_name*, *type_information*)

This method is called when a participant receives a TypeInformation while discovering another participant.

property thisown

The membership flag

DomainParticipantQos

class fastdds.**DomainParticipantQos**

Class DomainParticipantQos, contains all the possible Qos that can be set for a determined participant. Please consult each of them to check for implementation details and default values.

allocation(**args*)

Overload 1:

Getter for ParticipantResourceLimitsQos

Return type ParticipantResourceLimitsQos

Returns ParticipantResourceLimitsQos reference

Overload 2:

Getter for ParticipantResourceLimitsQos

Return type ParticipantResourceLimitsQos

Returns ParticipantResourceLimitsQos reference

Overload 3:

Setter for ParticipantResourceLimitsQos

Parameters **allocation** (ParticipantResourceLimitsQos) – ParticipantResourceLimitsQos

builtin_controllers_sender_thread(**args*)

Overload 1:

Getter for builtin flow controllers sender threads ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 2:

Getter for builtin flow controllers sender threads ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 3:

Setter for the builtin flow controllers sender threads ThreadSettings

Parameters **value** (ThreadSettings) – New ThreadSettings to be set

discovery_server_thread(**args*)

Overload 1:

Getter for discovery server ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 2:

Getter for discovery server ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 3:

Setter for the discovery server ThreadSettings

Parameters **value** (ThreadSettings) – New ThreadSettings to be set

entity_factory(**args*)

Overload 1:

Getter for EntityFactoryQosPolicy

Return type *EntityFactoryQosPolicy*

Returns EntityFactoryQosPolicy reference

Overload 2:

Getter for EntityFactoryQosPolicy

Return type *EntityFactoryQosPolicy*

Returns EntityFactoryQosPolicy reference

Overload 3:

Setter for EntityFactoryQosPolicy

Parameters **value** (*EntityFactoryQosPolicy*) – EntityFactoryQosPolicy

flow_controllers(**args*)

Overload 1:

Getter for FlowControllerDescriptorList

Return type FlowControllerDescriptorList

Returns FlowControllerDescriptorList reference

Overload 2:

Getter for FlowControllerDescriptorList

Return type FlowControllerDescriptorList

Returns FlowControllerDescriptorList reference

name(**args*)

Overload 1:

Getter for the Participant name

Return type string

Returns name

Overload 2:

Setter for the Participant name

Parameters **value** (*string*) – New name to be set.

properties(**args*)

Overload 1:

Getter for PropertyPolicyQos

Return type PropertyPolicyQos

Returns PropertyPolicyQos reference

Overload 2:

Getter for PropertyPolicyQos

Return type PropertyPolicyQos

Returns PropertyPolicyQos reference

Overload 3:

Setter for PropertyPolicyQos

Parameters **properties** (PropertyPolicyQos) – PropertyPolicyQos

setup_transports(**args*)

Provides a way of easily configuring transport related configuration on certain pre-defined scenarios with certain options.

Parameters

- **transports** (*eprosima::fastdds::rtps::BuiltinTransports*) – Defines the transport configuration scenario to setup.
- **options** (*eprosima::fastdds::rtps::BuiltinTransportsOptions*) – Defines the options to be used in the transport configuration.

property thisown

The membership flag

timed_events_thread(**args*)

Overload 1:

Getter for timed event ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 2:

Getter for timed event ThreadSettings

Return type ThreadSettings

Returns rtps::ThreadSettings reference

Overload 3:

Setter for the timed event ThreadSettings

Parameters **value** (ThreadSettings) – New ThreadSettings to be set

transport(**args*)

Overload 1:

Getter for TransportConfigQos

Return type *TransportConfigQos*

Returns TransportConfigQos reference

Overload 2:

Getter for TransportConfigQos

Return type *TransportConfigQos*

Returns TransportConfigQos reference

Overload 3:

Setter for TransportConfigQos

Parameters **transport** (*TransportConfigQos*) – TransportConfigQos

user_data(**args*)

Overload 1:

Getter for UserDataQosPolicy

Return type *UserDataQosPolicy*

Returns UserDataQosPolicy reference

Overload 2:

Getter for UserDataQosPolicy

Return type *UserDataQosPolicy*

Returns UserDataQosPolicy reference

Overload 3:

Setter for UserDataQosPolicy

Parameters **value** (*UserDataQosPolicy*) – UserDataQosPolicy

wire_protocol(**args*)

Overload 1:

Getter for WireProtocolConfigQos

Return type *WireProtocolConfigQos*

Returns WireProtocolConfigQos reference

Overload 2:

Getter for WireProtocolConfigQos

Return type *WireProtocolConfigQos*

Returns WireProtocolConfigQos reference

Overload 3:

Setter for WireProtocolConfigQos

Parameters **wire_protocol** (*WireProtocolConfigQos*) – WireProtocolConfigQos

Publisher

DataWriter

class fastdds.**DataWriter**(**args*, ***kwargs*)

Class DataWriter, contains the actual implementation of the behaviour of the DataWriter.

assert_liveliness()

This operation manually asserts the liveliness of the DataWriter. This is used in combination with the LivelinessQosPolicy to indicate to the Service that the entity remains active. This operation need only be used if the LIVELINESS setting is either MANUAL_BY_PARTICIPANT or MANUAL_BY_TOPIC. Otherwise, it has no effect.

Notes: Writing data via the write operation on a DataWriter asserts liveliness on the DataWriter itself and its DomainParticipant. Consequently the use of assert_liveliness is only needed if the application is not writing data regularly.

Return type ReturnCode_t

Returns RETCODE_OK if asserted, RETCODE_ERROR otherwise

clear_history()

dispose(*data*, *handle*)

This operation requests the middleware to delete the data (the actual deletion is postponed until there is no more use for that data in the whole system). In general, applications are made aware of the deletion by means of operations on the DataReader objects that already knew that instance. This operation does not modify the value of the instance. The instance parameter is passed just for the purposes of identifying the instance. When this operation is used, the Service will automatically supply the value of the source_timestamp that is made available to DataReader objects by means of the source_timestamp attribute inside the SampleInfo. The constraints on the values of the handle parameter and the corresponding error behavior are the same specified for the unregister_instance operation.

data Sample used to deduce instance's key in case of *handle* parameter is HANDLE_NIL. handle InstanceHandle of the data :rtype: `ReturnCode_t` :return: RETCODE_PRECONDITION_NOT_MET if the handle introduced does not match with the one associated to the data,

RETCODE_OK if the data is correctly sent and RETCODE_ERROR otherwise.

dispose_w_timestamp(*instance*, *handle*, *timestamp*)

This operation performs the same functions as 'dispose' except that the application provides the value for the 'source_timestamp' that is made available to DataReader objects by means of the 'source_timestamp' attribute inside the SampleInfo.

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the 'dispose' operation.

This operation may return RETCODE_PRECONDITION_NOT_MET and RETCODE_BAD_PARAMETER under the same circumstances described for the 'dispose' operation.

This operation may return RETCODE_TIMEOUT and RETCODE_OUT_OF_RESOURCES under the same circumstances described for the 'write' operation.

Parameters

- **instance** (*void*) – Sample used to deduce instance's key in case of *handle* parameter is HANDLE_NIL.
- **handle** (`InstanceHandle_t`) – Instance's key to be disposed.
- **timestamp** (`Time_t`) – Time_t used to set the source_timestamp.

Return type `ReturnCode_t`

Returns RTPS_DllAPI

enable()

This operation enables the DataWriter

Return type `ReturnCode_t`

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the Publisher creating this DataWriter is not enabled.

get_instance_handle()

Returns the DataWriter's InstanceHandle

Return type `InstanceHandle_t`

Returns Copy of the DataWriter InstanceHandle

get_key_value(*key_holder*, *handle*)

This operation can be used to retrieve the instance key that corresponds to an 'instance_handle'. The operation will only fill the fields that form the key inside the key_holder instance.

This operation may return BAD_PARAMETER if the InstanceHandle_t handle does not correspond to an existing data-object known to the DataWriter. If the implementation is not able to check invalid handles then the result in this situation is unspecified.

,out] key_holder Sample where the key fields will be returned. handle Handle to the instance to retrieve the key values from.

Return type `ReturnCode_t`

Returns Any of the standard return codes.

get_listener()
Retrieves the listener for this DataWriter.

Return type *DataWriterListener*

Returns Pointer to the DataWriterListener

get_liveliness_lost_status(*status*)
Returns the liveliness lost status

Parameters **status** (LivelinessLostStatus) – Liveliness lost status struct

Return type ReturnCode_t

Returns RETCODE_OK

get_matched_subscription_data(*subscription_data*, *subscription_handle*)
Retrieves in a subscription associated with the DataWriter

subscription_data subscription data struct :type subscription_handle: InstanceHandle_t :param subscription_handle: InstanceHandle_t of the subscription :rtype: ReturnCode_t :return: RETCODE_OK Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_matched_subscriptions(*subscription_handles*)
Fills the given vector with the InstanceHandle_t of matched DataReaders

subscription_handles Vector where the InstanceHandle_t are returned :rtype: ReturnCode_t :return: RETCODE_OK Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_offered_deadline_missed_status(*status*)
Returns the offered deadline missed status

status Deadline missed status struct :rtype: ReturnCode_t :return: RETCODE_OK

get_offered_incompatible_qos_status(*status*)
Returns the offered incompatible qos status

status Offered incompatible qos status struct :rtype: ReturnCode_t :return: RETCODE_OK

get_publication_matched_status(*status*)
Returns the publication matched status

status publication matched status struct :rtype: ReturnCode_t :return: RETCODE_OK

get_publisher()
Getter for the Publisher that creates this DataWriter

Return type *Publisher*

Returns Pointer to the Publisher

get_qos(**args*)
Overload 1:

Retrieves the DataWriterQos for this DataWriter.

Return type *DataWriterQos*

Returns Reference to the current DataWriterQos

Overload 2:

Fills the DataWriterQos with the values of this DataWriter.

Parameters **qos** (*DataWriterQos*) – DataWriterQos object where the qos is returned.

Return type ReturnCode_t

Returns RETCODE_OK

get_sending_locators(*locators*)

Get the list of locators from which this DataWriter may send data.

Parameters **[out]** – locators LocatorList where the list of locators will be stored.

Return type ReturnCode_t

Returns NOT_ENABLED if the reader has not been enabled.

Return type ReturnCode_t

Returns OK if a list of locators is returned.

get_topic()

Retrieves the topic for this DataWriter.

Return type *Topic*

Returns Pointer to the associated Topic

get_type()

Get data type associated to the DataWriter

Return type *TypeSupport*

Returns Copy of the TypeSupport

guid()

Returns the DataWriter's GUID

Return type GUID_t

Returns Reference to the DataWriter GUID

loan_sample(**args*)

Get a pointer to the internal pool where the user could directly write.

This method can only be used on a DataWriter for a plain data type. It will provide the user with a pointer to an internal buffer where the data type can be prepared for sending.

When using NO_LOAN_INITIALIZATION on the initialization parameter, which is the default, no assumptions should be made on the contents where the pointer points to, as it may be an old pointer being reused. See 'LoanInitializationKind' for more details.

Once the sample has been prepared, it can then be published by calling 'write'. After a successful call to 'write', the middleware takes ownership of the loaned pointer again, and the user should not access that memory again.

If, for whatever reason, the sample is not published, the loan can be returned by calling 'discard_loan'.

Parameters

- **[out]** – sample Pointer to the sample on the internal pool.
- **[in]** – initialization How to initialize the loaned sample.

Return type ReturnCode_t

Returns ReturnCode_t::RETCODE_ILLEGAL_OPERATION when the data type does not support loans.

Return type ReturnCode_t

Returns ReturnCode_t::RETCODE_NOT_ENABLED if the writer has not been enabled.

Return type ReturnCode_t

Returns ReturnCode_t::RETCODE_OUT_OF_RESOURCES if the pool has been exhausted.

Return type ReturnCode_t

Returns ReturnCode_t::RETCODE_OK if a pointer to a sample is successfully obtained.

lookup_instance(*instance*)
NOT YET IMPLEMENTED

Takes as a parameter an instance and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The instance parameter is only used for the purpose of examining the fields that define the key.

instance Data pointer to the sample

Return type InstanceHandle_t

Returns handle of the given instance

register_instance(*instance*)
Informs that the application will be modifying a particular instance. It gives an opportunity to the middleware to pre-configure itself to improve performance.

instance Sample used to get the instance's key. :rtype: InstanceHandle_t :return: Handle containing the instance's key.

This handle could be used in successive *write* or *dispose* operations. In case of error, HANDLE_NIL will be returned.

register_instance_w_timestamp(*instance*, *timestamp*)
This operation performs the same function as register_instance and can be used instead of 'register_instance' in the cases where the application desires to specify the value for the 'source_timestamp'. The 'source_timestamp' potentially affects the relative order in which readers observe events from multiple writers. See the QoS policy 'DESTINATION_ORDER'.

This operation may block and return RETCODE_TIMEOUT under the same circumstances described for the 'write' operation.

This operation may return RETCODE_OUT_OF_RESOURCES under the same circumstances described for the 'write' operation.

Parameters

- **instance** (*void*) – Sample used to get the instance's key.
- **timestamp** (Time_t) – Time_t used to set the source_timestamp.

Return type InstanceHandle_t

Returns Handle containing the instance's key.

set_listener(**args*)

set_qos(*qos*)
Establishes the DataWriterQos for this DataWriter.

Parameters **qos** (*DataWriterQos*) – DataWriterQos to be set

Return type ReturnCode_t

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

property thisown
The membership flag

unregister_instance(*instance*, *handle*)
This operation reverses the action of *register_instance*. It should only be called on an instance that is currently registered. Informs the middleware that the DataWriter is not intending to modify any more of that data instance. Also indicates that the middleware can locally remove all information regarding that instance.

instance Sample used to deduce instance's key in case of *handle* parameter is HANDLE_NIL. handle Instance's key to be unregistered. :rtype: `ReturnCode_t` :return: Returns the operation's result.

If the operation finishes successfully, ReturnCode_t::RETCODE_OK is returned.

unregister_instance_w_timestamp(*instance*, *handle*, *timestamp*)
This operation performs the same function as 'unregister_instance' and can be used instead of 'unregister_instance' in the cases where the application desires to specify the value for the 'source_timestamp'. The 'source_timestamp' potentially affects the relative order in which readers observe events from multiple writers. See the QoS policy 'DESTINATION_ORDER'.

The constraints on the values of the `handle` parameter and the corresponding error behavior are the same specified for the 'unregister_instance' operation.

This operation may block and return RETCODE_TIMEOUT under the same circumstances described for the write operation

Parameters

- **instance** (*void*) – Sample used to deduce instance's key in case of *handle* parameter is HANDLE_NIL.
- **handle** (`InstanceHandle_t`) – Instance's key to be unregistered.
- **timestamp** (`Time_t`) – Time_t used to set the source_timestamp.

Return type `ReturnCode_t`

Returns Handle containing the instance's key.

wait_for_acknowledgments(**args*)
Overload 1:

Waits the current thread until all writers have received their acknowledgments.

Parameters **max_wait** (`Duration_t`) – Maximum blocking time for this operation

Return type `ReturnCode_t`

Returns RETCODE_OK if the DataWriter receive the acknowledgments before the time expires and RETCODE_ERROR otherwise

Overload 2:

Block the current thread until the writer has received the acknowledgment corresponding to the given instance. Operations performed on the same instance while the current thread is waiting will not be taken into consideration, i.e. this method may return *RETCODE_OK* with those operations unacknowledged.

Parameters

- **instance** (*void*) – Sample used to deduce instance's key in case of *handle* parameter is HANDLE_NIL.
- **handle** (InstanceHandle_t) – Instance handle of the data.
- **max_wait** (Duration_t) – Maximum blocking time for this operation.

Return type ReturnCode_t

Returns RETCODE_NOT_ENABLED if the writer has not been enabled.

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if *instance* is not a valid pointer.

Return type ReturnCode_t

Returns RETCODE_PRECONDITION_NOT_MET if the topic does not have a key, the key is unknown to the writer, or the key is not consistent with *handle*.

Return type ReturnCode_t

Returns RETCODE_OK if the DataWriter received the acknowledgments before the time expired.

Return type ReturnCode_t

Returns RETCODE_TIMEOUT otherwise.

write(**args*)

Overload 1:

Write data to the topic.

Parameters **data** (*void*) – Pointer to the data

Return type boolean

Returns True if correct, false otherwise

Overload 2:

Write data with params to the topic.

Parameters

- **data** (*void*) – Pointer to the data
- **params** (WriteParams) – Extra write parameters.

Return type boolean

Returns True if correct, false otherwise

Overload 3:

Write data with handle.

The special value HANDLE_NIL can be used for the parameter handle.This indicates that the identity of the instance should be automatically deduced from the instance_data (by means of the key).

Parameters

- **data** (*void*) – Pointer to the data
- **handle** (InstanceHandle_t) – InstanceHandle_t.

Return type ReturnCode_t

Returns RETCODE_PRECONDITION_NOT_MET if the handle introduced does not match with the one associated to the data, RETCODE_OK if the data is correctly sent and RETCODE_ERROR otherwise.

write_w_timestamp(*data*, *handle*, *timestamp*)

This operation performs the same function as write except that it also provides the value for the 'source_timestamp' that is made available to DataReader objects by means of the 'eprosima::fastdds::dds::SampleInfo::source_timestamp' attribute "source_timestamp" inside the SampleInfo. The constraints on the values of the handle parameter and the corresponding error behavior are the same specified for the 'write' operation. This operation may block and return RETCODE_TIMEOUT under the same circumstances described for the 'write' operation. This operation may return RETCODE_OUT_OF_RESOURCES, RETCODE_PRECONDITION_NOT_MET or RETCODE_BAD_PARAMETER under the same circumstances described for the write operation.

Parameters

- **data** (*void*) – Pointer to the data
- **handle** (InstanceHandle_t) – InstanceHandle_t
- **timestamp** (Time_t) – Time_t used to set the source_timestamp.

Return type ReturnCode_t

Returns Any of the standard return codes.

DataWriterListener

class fastdds.**DataWriterListener**

Class DataWriterListener, allows the end user to implement callbacks triggered by certain events.

on_liveliness_lost(*writer*, *status*)

Method called when the liveliness of a DataWriter is lost

Parameters

- **writer** (*DataWriter*) – Pointer to the associated DataWriter
- **status** (LivelinessLostStatus) – The liveliness lost status

on_offered_deadline_missed(*writer*, *status*)

A method called when a deadline is missed

Parameters

- **writer** (*DataWriter*) – Pointer to the associated DataWriter
- **status** (OfferedDeadlineMissedStatus) – The deadline missed status

on_offered_incompatible_qos(*writer*, *status*)
A method called when an incompatible QoS is offered

Parameters

- **writer** (*DataWriter*) – Pointer to the associated DataWriter
- **status** (OfferedIncompatibleQosStatus) – The deadline missed status

on_publication_matched(*writer*, *info*)
This method is called when the DataWriter is matched (or unmatched) against an endpoint.

Parameters

- **writer** (*DataWriter*) – Pointer to the associated DataWriter
- **info** (*PublicationMatchedStatus*) – Information regarding the matched DataReader

on_unacknowledged_sample_removed(*writer*, *instance*)
Method called when a sample has been removed unacknowledged

Parameters

- **writer** (*DataWriter*) – Pointer to the associated DataWriter
- **instance** (InstanceHandle_t) – Handle to the instance the sample was removed from

property thisown
The membership flag

DataWriterQos

class fastdds.**DataWriterQos**
Class DataWriterQos, containing all the possible Qos that can be set for a determined DataWriter. Although these values can be and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

data_sharing(**args*)
Overload 1:

Getter for DataSharingQosPolicy

Return type *DataSharingQosPolicy*

Returns DataSharingQosPolicy reference

Overload 2:

Getter for DataSharingQosPolicy

Return type *DataSharingQosPolicy*

Returns DataSharingQosPolicy reference

Overload 3:

Setter for DataSharingQosPolicy

Parameters **data_sharing** (*DataSharingQosPolicy*) – new value for the DataSharingQosPolicy

deadline(**args*)

Overload 1:

Getter for DeadlineQosPolicy

Return type *DeadlineQosPolicy*

Returns DeadlineQosPolicy reference

Overload 2:

Getter for DeadlineQosPolicy

Return type *DeadlineQosPolicy*

Returns DeadlineQosPolicy reference

Overload 3:

Setter for DeadlineQosPolicy

Parameters **deadline** (*DeadlineQosPolicy*) – new value for the DeadlineQosPolicy

destination_order(**args*)

Overload 1:

Getter for DestinationOrderQosPolicy

Return type *DestinationOrderQosPolicy*

Returns DestinationOrderQosPolicy reference

Overload 2:

Getter for DestinationOrderQosPolicy

Return type *DestinationOrderQosPolicy*

Returns DestinationOrderQosPolicy reference

Overload 3:

Setter for DestinationOrderQosPolicy

Parameters **destination_order** (*DestinationOrderQosPolicy*) – new value for the DestinationOrderQosPolicy

durability(**args*)

Overload 1:

Getter for DurabilityQosPolicy

Return type *DurabilityQosPolicy*

Returns DurabilityQosPolicy reference

Overload 2:

Getter for DurabilityQosPolicy

Return type *DurabilityQosPolicy*

Returns DurabilityQosPolicy reference

Overload 3:

Setter for DurabilityQosPolicy

Parameters **durability** (*DurabilityQosPolicy*) – new value for the DurabilityQosPolicy

durability_service(**args*)

Overload 1:

Getter for DurabilityServiceQosPolicy

Return type *DurabilityServiceQosPolicy*

Returns DurabilityServiceQosPolicy reference

Overload 2:

Getter for DurabilityServiceQosPolicy

Return type *DurabilityServiceQosPolicy*

Returns DurabilityServiceQosPolicy reference

Overload 3:

Setter for DurabilityServiceQosPolicy

Parameters **durability_service** (*DurabilityServiceQosPolicy*) – new value for the DurabilityServiceQosPolicy

endpoint(**args*)

Overload 1:

Getter for RTPSEndpointQos

Return type *RTPSEndpointQos*

Returns RTPSEndpointQos reference

Overload 2:

Getter for RTPSEndpointQos :rtype: *RTPSEndpointQos* :return: RTPSEndpointQos reference

Overload 3:

Setter for RTPSEndpointQos

Parameters **endpoint** (*RTPSEndpointQos*) – new value for the RTPSEndpointQos

get_writerqos(*pqos*, *tqos*)

history(**args*)

Overload 1:

Getter for HistoryQosPolicy

Return type *HistoryQosPolicy*

Returns HistoryQosPolicy reference

Overload 2:

Getter for HistoryQosPolicy

Return type *HistoryQosPolicy*

Returns HistoryQosPolicy reference

Overload 3:

Setter for HistoryQosPolicy

Parameters **history** (*HistoryQosPolicy*) – new value for the HistoryQosPolicy

latency_budget(**args*)

Overload 1:

Getter for LatencyBudgetQosPolicy

Return type *LatencyBudgetQosPolicy*

Returns LatencyBudgetQosPolicy reference

Overload 2:

Getter for LatencyBudgetQosPolicy

Return type *LatencyBudgetQosPolicy*

Returns LatencyBudgetQosPolicy reference

Overload 3:

Setter for LatencyBudgetQosPolicy

Parameters **latency_budget** (*LatencyBudgetQosPolicy*) – new value for the Latency-BudgetQosPolicy

lifespan(**args*)

Overload 1:

Getter for LifespanQosPolicy

Return type *LifespanQosPolicy*

Returns LifespanQosPolicy reference

Overload 2:

Getter for LifespanQosPolicy

Return type *LifespanQosPolicy*

Returns LifespanQosPolicy reference

Overload 3:

Setter for LifespanQosPolicy

Parameters **lifespan** (*LifespanQosPolicy*) – new value for the LifespanQosPolicy

liveliness(**args*)

Overload 1:

Getter for LivelinessQosPolicy

Return type *LivelinessQosPolicy*

Returns LivelinessQosPolicy reference

Overload 2:

Getter for LivelinessQosPolicy

Return type *LivelinessQosPolicy*

Returns LivelinessQosPolicy reference

Overload 3:

Setter for LivelinessQosPolicy

Parameters **liveliness** (*LivelinessQosPolicy*) – new value for the LivelinessQosPolicy

ownership(**args*)

Overload 1:

Getter for OwnershipQosPolicy

Return type *OwnershipQosPolicy*

Returns OwnershipQosPolicy reference

Overload 2:

Getter for OwnershipQosPolicy

Return type *OwnershipQosPolicy*

Returns OwnershipQosPolicy reference

Overload 3:

Setter for OwnershipQosPolicy

Parameters **ownership** (*OwnershipQosPolicy*) – new value for the OwnershipQosPolicy

ownership_strength(**args*)

Overload 1:

Getter for OwnershipStrengthQosPolicy

Return type *OwnershipStrengthQosPolicy*

Returns OwnershipStrengthQosPolicy reference

Overload 2:

Getter for OwnershipStrengthQosPolicy

Return type *OwnershipStrengthQosPolicy*

Returns OwnershipStrengthQosPolicy reference

Overload 3:

Setter for OwnershipStrengthQosPolicy

Parameters **ownership_strength** (*OwnershipStrengthQosPolicy*) – new value for the OwnershipStrengthQosPolicy

properties(**args*)

Overload 1:

Getter for PropertyPolicyQos

Return type PropertyPolicyQos

Returns PropertyPolicyQos reference

Overload 2:

Getter for PropertyPolicyQos

Return type PropertyPolicyQos

Returns PropertyPolicyQos reference

Overload 3:

Setter for PropertyPolicyQos

Parameters **properties** (PropertyPolicyQos) – new value for the PropertyPolicyQos

publish_mode(**args*)

Overload 1:

Getter for PublishModeQosPolicy

Return type *PublishModeQosPolicy*

Returns PublishModeQosPolicy reference

Overload 2:

Getter for PublishModeQosPolicy

Return type *PublishModeQosPolicy*

Returns PublishModeQosPolicy reference

Overload 3:

Setter for PublishModeQosPolicy

Parameters **publish_mode** (*PublishModeQosPolicy*) – new value for the PublishModeQosPolicy

reliability(**args*)

Overload 1:

Getter for ReliabilityQosPolicy

Return type *ReliabilityQosPolicy*

Returns ReliabilityQosPolicy reference

Overload 2:

Getter for ReliabilityQosPolicy

Return type *ReliabilityQosPolicy*

Returns ReliabilityQosPolicy reference

Overload 3:

Setter for ReliabilityQosPolicy

Parameters **reliability** (*ReliabilityQosPolicy*) – new value for the ReliabilityQosPolicy

reliable_writer_qos(**args*)

Overload 1:

Getter for RTPSReliableWriterQos

Return type *RTPSReliableWriterQos*

Returns RTPSReliableWriterQos reference

Overload 2:

Getter for RTPSReliableWriterQos

Return type *RTPSReliableWriterQos*

Returns RTPSReliableWriterQos reference

Overload 3:

Setter for RTPSReliableWriterQos

Parameters **reliable_writer_qos** (*RTPSReliableWriterQos*) – new value for the RTPSReliableWriterQos

resource_limits(**args*)

Overload 1:

Getter for ResourceLimitsQosPolicy

Return type *ResourceLimitsQosPolicy*

Returns ResourceLimitsQosPolicy reference

Overload 2:

Getter for ResourceLimitsQosPolicy

Return type *ResourceLimitsQosPolicy*

Returns ResourceLimitsQosPolicy reference

Overload 3:

Setter for ResourceLimitsQosPolicy

Parameters **resource_limits** (*ResourceLimitsQosPolicy*) – new value for the ResourceLimitsQosPolicy

property thisown
The membership flag

transport_priority(**args*)
Overload 1:

Getter for TransportPriorityQosPolicy

Return type *TransportPriorityQosPolicy*

Returns TransportPriorityQosPolicy reference

Overload 2:

Getter for TransportPriorityQosPolicy

Return type *TransportPriorityQosPolicy*

Returns TransportPriorityQosPolicy reference

Overload 3:

Setter for TransportPriorityQosPolicy

Parameters **transport_priority** (*TransportPriorityQosPolicy*) – new value for the TransportPriorityQosPolicy

user_data(**args*)
Overload 1:

Getter for UserDataQosPolicy

Return type *UserDataQosPolicy*

Returns UserDataQosPolicy reference

Overload 2:

Getter for UserDataQosPolicy

Return type *UserDataQosPolicy*

Returns UserDataQosPolicy reference

Overload 3:

Setter for UserDataQosPolicy

Parameters **user_data** (*UserDataQosPolicy*) – new value for the UserDataQosPolicy

writer_data_lifecycle(**args*)

Overload 1:

Getter for WriterDataLifecycleQosPolicy

Return type *WriterDataLifecycleQosPolicy*

Returns WriterDataLifecycleQosPolicy reference

Overload 2:

Getter for WriterDataLifecycleQosPolicy

Return type *WriterDataLifecycleQosPolicy*

Returns WriterDataLifecycleQosPolicy reference

Overload 3:

Setter for WriterDataLifecycleQosPolicy

Parameters **writer_data_lifecycle** (*WriterDataLifecycleQosPolicy*) – new value for the WriterDataLifecycleQosPolicy

writer_resource_limits(**args*)

Overload 1:

Getter for WriterResourceLimitsQos

Return type *WriterResourceLimitsQos*

Returns WriterResourceLimitsQos reference

Overload 2:

Getter for WriterResourceLimitsQos

Return type *WriterResourceLimitsQos*

Returns WriterResourceLimitsQos reference

Overload 3:

Setter for WriterResourceLimitsQos

Parameters **writer_resource_limits** (*WriterResourceLimitsQos*) – new value for the WriterResourceLimitsQos

class fastdds.**DATAWRITER_QOS_DEFAULT**(**args: Any*, ***kwargs: Any*)

Publisher

class fastdds.**Publisher**(**args*, ***kwargs*)

Class Publisher, used to send data to associated subscribers.

begin_coherent_changes()

Signals the beginning of a set of coherent cache changes using the Datawriters attached to the publisher

Return type ReturnCode_t

Returns RETCODE_OK if successful, an error code otherwise

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

copy_from_topic_qos(*writer_qos*, *topic_qos*)

Copies TopicQos into the corresponding DataWriterQos

writer_qos topic_qos :rtype: ReturnCode_t :return: RETCODE_OK if successful, an error code otherwise Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

create_datawriter(**args*)

This operation creates a DataWriter. The returned DataWriter will be attached and belongs to the Publisher.

Parameters

- **topic** (*Topic*) – Topic the DataWriter will be listening
- **qos** – QoS of the DataWriter.
- **listener** (*DataWriterListener*) – Pointer to the listener (default: nullptr).
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all).

Return type *DataWriter*

Returns Pointer to the created DataWriter. nullptr if failed.

create_datawriter_with_profile(**args*)

This operation creates a DataWriter. The returned DataWriter will be attached and belongs to the Publisher.

Parameters

- **topic** (*Topic*) – Topic the DataWriter will be listening
- **profile_name** (*string*) – DataWriter profile name.
- **listener** (*DataWriterListener*) – Pointer to the listener (default: nullptr).
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all).

Return type *DataWriter*

Returns Pointer to the created DataWriter. nullptr if failed.

delete_contained_entities()
Deletes all contained DataWriters

Return type ReturnCode_t

Returns RETCODE_OK if successful, an error code otherwise

delete_datawriter(*writer*)
This operation deletes a DataWriter that belongs to the Publisher.

The delete_datawriter operation must be called on the same Publisher object used to create the DataWriter. If delete_datawriter is called on a different Publisher, the operation will have no effect and it will return false.

The deletion of the DataWriter will automatically unregister all instances. Depending on the settings of the WRITER_DATA_LIFECYCLE QosPolicy, the deletion of the DataWriter may also dispose all instances.

Parameters **writer** (*DataWriter*) – DataWriter to delete

Return type ReturnCode_t

Returns RETCODE_PRECONDITION_NOT_MET if it does not belong to this Publisher, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

enable()
This operation enables the Publisher

Return type ReturnCode_t

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the participant creating this Publisher is not enabled.

end_coherent_changes()
Signals the end of a set of coherent cache changes

Return type ReturnCode_t

Returns RETCODE_OK if successful, an error code otherwise

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_datawriter_qos_from_profile(*profile_name*, *qos*)
Fills the DataWriterQos with the values of the XML profile.

Parameters

- **profile_name** (*string*) – DataWriter profile name.
- **qos** (*DataWriterQos*) – DataWriterQos object where the qos is returned.

Return type ReturnCode_t

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

get_datawriters(*writers*)
Fills the given vector with all the datawriters of this publisher.

Parameters **writers** (std::vector< eprosima::fastdds::dds::DataWriter *,std::allocator< eprosima::fastdds::dds::DataWriter * > >) – Vector where the DataWriters are returned

Return type boolean

Returns true

get_default_datawriter_qos(*args)

Overload 1:

This operation returns the default value of the DataWriter QoS, that is, the QoS policies which will be used for newly created DataWriter entities in the case where the QoS policies are defaulted in the create_datawriter operation.

The values retrieved by get_default_datawriter_qos will match the set of values specified on the last successful call to set_default_datawriter_qos, or else, if the call was never made, the default values.

Return type *DataWriterQos*

Returns Current default WriterQos

Overload 2:

This operation retrieves the default value of the DataWriter QoS, that is, the QoS policies which will be used for newly created DataWriter entities in the case where the QoS policies are defaulted in the create_datawriter operation.

The values retrieved by get_default_datawriter_qos will match the set of values specified on the last successful call to set_default_datawriter_qos, or else, if the call was never made, the default values.

Parameters **qos** (*DataWriterQos*) – Reference to the current default WriterQos.

Return type ReturnCode_t

Returns RETCODE_OK

get_instance_handle()

Returns the Publisher's handle.

Return type InstanceHandle_t

Returns InstanceHandle of this Publisher.

get_listener()

Retrieves the attached PublisherListener.

Return type *PublisherListener*

Returns PublisherListener pointer

get_participant()

This operation returns the DomainParticipant to which the Publisher belongs.

Return type *DomainParticipant*

Returns Pointer to the DomainParticipant

get_qos(*args)

Overload 1:

Allows accessing the Publisher Qos.

Return type *PublisherQos*

Returns PublisherQos reference

Overload 2:

Retrieves the Publisher Qos.

Return type ReturnCode_t

Returns RETCODE_OK

has_datawriters()
This operation checks if the publisher has DataWriters

Return type boolean

Returns true if the publisher has one or several DataWriters, false otherwise

lookup_datawriter(*topic_name*)
This operation retrieves a previously created DataWriter belonging to the Publisher that is attached to a Topic with a matching topic_name. If no such DataWriter exists, the operation will return nullptr.

If multiple DataWriter attached to the Publisher satisfy this condition, then the operation will return one of them. It is not specified which one.

Parameters **topic_name** (*string*) – Name of the Topic

Return type *DataWriter*

Returns Pointer to a previously created DataWriter associated to a Topic with the requested topic_name

resume_publications()
Indicates to FastDDS that the modifications to the DataWriters are complete.

Return type ReturnCode_t

Returns RETCODE_OK if successful, an error code otherwise

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

set_default_datawriter_qos(*qos*)
This operation sets a default value of the DataWriter QoS policies which will be used for newly created DataWriter entities in the case where the QoS policies are defaulted in the create_datawriter operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value DATAWRITER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_datawriter_qos operation had never been called.

Parameters **qos** (*DataWriterQos*) – DataWriterQos to be set

Return type ReturnCode_t

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

set_listener(**args*)
Overload 1:

Modifies the PublisherListener, sets the mask to StatusMask::all()

Parameters **listener** (*PublisherListener*) – new value for the PublisherListener

Return type ReturnCode_t

Returns RETCODE_OK

Overload 2:

Modifies the PublisherListener.

Parameters

- **listener** (*PublisherListener*) – new value for the PublisherListener
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to

Return type ReturnCode_t

Returns RETCODE_OK

set_qos(*qos*)

Allows modifying the Publisher Qos. The given Qos must be supported by the PublisherQos.

Parameters **qos** (*PublisherQos*) – PublisherQos to be set

Return type ReturnCode_t

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

suspend_publications()

Indicates to FastDDS that the contained DataWriters are about to be modified

Return type ReturnCode_t

Returns RETCODE_OK if successful, an error code otherwise

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

property thisown

The membership flag

wait_for_acknowledgments(*max_wait*)

This operation blocks the calling thread until either all data written by the reliable DataWriter entities is acknowledged by all matched reliable DataReader entities, or else the duration specified by the max_wait parameter elapses, whichever happens first. A return value of true indicates that all the samples written have been acknowledged by all reliable matched data readers; a return value of false indicates that max_wait elapsed before all the data was acknowledged.

Parameters **max_wait** (Duration_t) – Maximum blocking time for this operation

Return type ReturnCode_t

Returns RETCODE_TIMEOUT if the function takes more than the maximum blocking time established, RETCODE_OK if the Publisher receives the acknowledgments and RETCODE_ERROR otherwise.

PublisherListener

class fastdds.**PublisherListener**

Class PublisherListener, allows the end user to implement callbacks triggered by certain events. It inherits all the DataWriterListener callbacks.

property **thisown**

The membership flag

PublisherQos

class fastdds.**PublisherQos**

Class PublisherQos, containing all the possible Qos that can be set for a determined Publisher. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

entity_factory(**args*)

Overload 1:

Getter for EntityFactoryQosPolicy

Return type *EntityFactoryQosPolicy*

Returns EntityFactoryQosPolicy reference

Overload 2:

Getter for EntityFactoryQosPolicy

Return type *EntityFactoryQosPolicy*

Returns EntityFactoryQosPolicy reference

Overload 3:

Setter for EntityFactoryQosPolicy

Parameters **entity_factory** (*EntityFactoryQosPolicy*) – EntityFactoryQosPolicy

group_data(**args*)

Overload 1:

Getter for GroupDataQosPolicy

Return type *GroupDataQosPolicy*

Returns GroupDataQosPolicy reference

Overload 2:

Getter for GroupDataQosPolicy

Return type *GroupDataQosPolicy*

Returns GroupDataQosPolicy reference

Overload 3:

Setter for GroupDataQosPolicy

Parameters **group_data** (*GroupDataQosPolicy*) – GroupDataQosPolicy

partition(**args*)

Overload 1:

Getter for PartitionQosPolicy

Return type *PartitionQosPolicy*

Returns PartitionQosPolicy reference

Overload 2:

Getter for PartitionQosPolicy

Return type *PartitionQosPolicy*

Returns PartitionQosPolicy reference

Overload 3:

Setter for PartitionQosPolicy

Parameters **partition** (*PartitionQosPolicy*) – PartitionQosPolicy

presentation(**args*)

Overload 1:

Getter for PresentationQosPolicy

Return type *PresentationQosPolicy*

Returns PresentationQosPolicy reference

Overload 2:

Getter for PresentationQosPolicy

Return type *PresentationQosPolicy*

Returns PresentationQosPolicy reference

Overload 3:

Setter for PresentationQosPolicy

Parameters **presentation** (*PresentationQosPolicy*) – PresentationQosPolicy

property thisown
The membership flag

class fastdds.**PUBLISHER_QOS_DEFAULT**(**args: Any*, ***kwargs: Any*)

RTPSReliableWriterQos

class fastdds.**RTPSReliableWriterQos**
Qos Policy to configure the DisablePositiveACKsQos and the writer timing attributes

property disable_heartbeat_piggyback
Disable heartbeat piggyback mechanism.

property disable_positive_acks
Disable positive acks QoS, implemented in the library.

property thisown
The membership flag

property times
Writer Timing Attributes

Subscriber

DataReader

class fastdds.**DataReader**(**args*, ***kwargs*)
Class DataReader, contains the actual implementation of the behaviour of the Subscriber.

create_querycondition(*sample_states*, *view_states*, *instance_states*, *query_expression*, *query_parameters*)
This operation creates a QueryCondition. The returned QueryCondition will be attached and belong to the DataReader.

Parameters

- **[in]** – sample_states Only data samples with `sample_state` matching one of these will trigger the created condition.
- **[in]** – view_states Only data samples with `view_state` matching one of these will trigger the created condition.

- **[in]** – instance_states Only data samples with `instance_state` matching one of these will trigger the created condition.
- **[in]** – query_expression Only data samples matching this query will trigger the created condition.
- **[in]** – query_parameters Value of the parameters on the query expression.

Return type eprosima::fastdds::dds::QueryCondition

Returns pointer to the created QueryCondition, nullptr in case of error.

`create_readcondition`(*sample_states*, *view_states*, *instance_states*)

This operation creates a ReadCondition. The returned ReadCondition will be attached and belong to the DataReader.

Parameters

- **[in]** – sample_states Only data samples with `sample_state` matching one of these will trigger the created condition.
- **[in]** – view_states Only data samples with `view_state` matching one of these will trigger the created condition.
- **[in]** – instance_states Only data samples with `instance_state` matching one of these will trigger the created condition.

Return type `ReadCondition`

Returns pointer to the created ReadCondition, nullptr in case of error.

`delete_contained_entities`()

This operation deletes all the entities that were created by means of the "create" operations on the DataReader. That is, it deletes all contained ReadCondition and QueryCondition objects.

The operation will return PRECONDITION_NOT_MET if the any of the contained entities is in a state where it cannot be deleted.

Return type `ReturnCode_t`

Returns Any of the standard return codes.

`delete_readcondition`(*a_condition*)

This operation deletes a ReadCondition attached to the DataReader.

Parameters **a_condition** (`ReadCondition`) – pointer to a ReadCondition belonging to the DataReader

Return type `ReturnCode_t`

Returns RETCODE_OK

`enable`()

This operation enables the DataReader.

Return type `ReturnCode_t`

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the Subscriber creating this DataReader is not enabled.

`get_first_untaken_info`(*info*)

Returns information about the first untaken sample.

Parameters **[out]** – info Pointer to a SampleInfo_t structure to store first untaken sample information.

Return type ReturnCode_t

Returns RETCODE_OK if sample info was returned. RETCODE_NO_DATA if there is no sample to take.

get_instance_handle()
Getter for the associated InstanceHandle.

Return type InstanceHandle_t

Returns Copy of the InstanceHandle

get_key_value(*key_holder*, *handle*)
NOT YET IMPLEMENTED

This operation can be used to retrieve the instance key that corresponds to an instance_handle. The operation will only fill the fields that form the key inside the key_holder instance.

This operation may return BAD_PARAMETER if the InstanceHandle_t a_handle does not correspond to an existing data-object known to the DataReader. If the implementation is not able to check invalid handles then the result in this situation is unspecified.

,out] key_holder handle

Return type ReturnCode_t

Returns Any of the standard return codes.

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_listener()
Getter for the DataReaderListener

Return type *DataReaderListener*

Returns Pointer to the DataReaderListener

get_listening_locators(*locators*)
Get the list of locators on which this DataReader is listening.

Parameters **[out]** – locators LocatorList where the list of locators will be stored.

Return type ReturnCode_t

Returns NOT_ENABLED if the reader has not been enabled.

Return type ReturnCode_t

Returns OK if a list of locators is returned.

get_liveliness_changed_status(*status*)
Get the liveliness changed status.

Parameters **[out]** – status LivelinessChangedStatus object where the status is returned.

Return type ReturnCode_t

Returns RETCODE_OK

get_matched_publication_data(*publication_data*, *publication_handle*)
Retrieves in a publication associated with the DataWriter

publication_data publication data struct :type publication_handle: InstanceHandle_t :param publication_handle: InstanceHandle_t of the publication :rtype: ReturnCode_t :return: RETCODE_OK Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_matched_publications(*publication_handles*)
Fills the given vector with the InstanceHandle_t of matched DataReaders

publication_handles Vector where the InstanceHandle_t are returned :rtype: `ReturnCode_t` :return: RETCODE_OK Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_qos(**args*)
Overload 1:

Getter for the DataReaderQos.

Return type *DataReaderQos*

Returns Pointer to the DataReaderQos.

Overload 2:

Getter for the DataReaderQos.

Parameters **[in]** – qos DataReaderQos where the qos is returned.

Return type `ReturnCode_t`

Returns RETCODE_OK

get_requested_deadline_missed_status(*status*)
Get the requested deadline missed status.

Return type `ReturnCode_t`

Returns The deadline missed status.

get_requested_incompatible_qos_status(*status*)
Get the requested incompatible qos status.

Parameters **[out]** – status Requested incompatible qos status.

Return type `ReturnCode_t`

Returns RETCODE_OK

get_sample_lost_status(*status*)
Get the SAMPLE_LOST communication status

status SampleLostStatus object where the status is returned.

Return type `ReturnCode_t`

Returns RETCODE_OK

get_sample_rejected_status(*status*)
Get the SAMPLE_REJECTED communication status

status SampleRejectedStatus object where the status is returned.

Return type `ReturnCode_t`

Returns RETCODE_OK

get_subscriber()
Getter for the Subscriber :rtype: *Subscriber* :return: Subscriber pointer

get_subscription_matched_status(*status*)
Returns the subscription matched status

status subscription matched status struct :rtype: ReturnCode_t :return: RETCODE_OK

get_topicdescription()
Get TopicDescription.

Return type *TopicDescription*

Returns TopicDescription pointer.

get_unread_count(**args*)
Overload 1:

Get the number of samples pending to be read. The number includes samples that may not yet be available to be read or taken by the user, due to samples being received out of order.

Return type int

Returns the number of samples on the reader history that have never been read.

Overload 2:

Get the number of samples pending to be read.

Parameters **mark_as_read** (*boolean*) – Whether the unread samples should be marked as read or not.

Return type int

Returns the number of samples on the reader history that have never been read.

guid(**args*)
Overload 1:

Get associated GUID.

Return type GUID_t

Returns Associated GUID

Overload 2:

Get associated GUID.

Return type GUID_t

Returns Associated GUID

is_sample_valid(*data*, *info*)
Checks whether a loaned sample is still valid or is corrupted. Calling this method on a sample which has not been loaned, or one for which the loan has been returned yields undefined behavior.

Parameters

- **data** (*void*) – Pointer to the sample data to check

- **info** (*SampleInfo*) – Pointer to the SampleInfo related to `data`

Return type boolean

Returns true if the sample is valid

`lookup_instance`(*instance*)

Takes as a parameter an instance and returns a handle that can be used in subsequent operations that accept an instance handle as an argument. The instance parameter is only used for the purpose of examining the fields that define the key.

Parameters `[in]` – instance Data pointer to the sample

Return type `InstanceHandle_t`

Returns handle of the given `instance`.

Return type `InstanceHandle_t`

Returns HANDLE_NIL if `instance` is nullptr.

Return type `InstanceHandle_t`

Returns HANDLE_NIL if there is no instance on the DataReader's history with the same key as `instance`.

`read`(**args*)

Access a collection of data samples from the DataReader.

This operation accesses a collection of Data values from the DataReader. The caller can limit the size of the returned collection with the `max_samples` parameter.

The properties of the `data_values` collection and the setting of the 'PresentationQosPolicy' may impose further limits on the size of the returned 'list.'

1. If 'PresentationQosPolicy::access_scope' is 'INSTANCE_PRESENTATION_QOS', then the returned collection is a 'list' where samples belonging to the same data-instance are consecutive.
2. If 'PresentationQosPolicy::access_scope' is 'TOPIC_PRESENTATION_QOS' and 'PresentationQosPolicy::ordered_access' is set to `false`, then the returned collection is a 'list' where samples belonging to the same data-instance are consecutive.
3. If 'PresentationQosPolicy::access_scope' is 'TOPIC_PRESENTATION_QOS' and 'PresentationQosPolicy::ordered_access' is set to `true`, then the returned collection is a 'list' where samples belonging to the same instance may or may not be consecutive. This is because to preserve order it may be necessary to mix samples from different instances.
4. If 'PresentationQosPolicy::access_scope' is 'GROUP_PRESENTATION_QOS' and 'PresentationQosPolicy::ordered_access' is set to `false`, then the returned collection is a 'list' where samples belonging to the same data instance are consecutive.
5. If 'PresentationQosPolicy::access_scope' is 'GROUP_PRESENTATION_QOS' and 'PresentationQosPolicy::ordered_access' is set to `true`, then the returned collection contains at most one sample. The difference in this case is due to the fact that it is required that the application is able to read samples belonging to different DataReader objects in a specific order.

In any case, the relative order between the samples of one instance is consistent with the 'DestinationOrderQosPolicy':

- If 'DestinationOrderQosPolicy::kind' is 'BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS', samples belonging to the same instances will appear in the relative order in which there were received (FIFO, earlier samples ahead of the later samples).

- If 'DestinationOrderQosPolicy::kind' is 'BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS', samples belonging to the same instances will appear in the relative order implied by the source_timestamp (FIFO, smaller values of source_timestamp ahead of the larger values).

The actual number of samples returned depends on the information that has been received by the middleware as well as the 'HistoryQosPolicy', 'ResourceLimitsQosPolicy', and 'ReaderResourceLimitsQos':

- In the case where the 'HistoryQosPolicy::kind' is KEEP_LAST_HISTORY_QOS, the call will return at most 'HistoryQosPolicy::depth' samples per instance.
- The maximum number of samples returned is limited by 'ResourceLimitsQosPolicy::max_samples', and by 'ReaderResourceLimitsQos::max_samples_per_read'.
- For multiple instances, the number of samples returned is additionally limited by the product ('ResourceLimitsQosPolicy::max_samples_per_instance' * 'ResourceLimitsQosPolicy::max_instances)'.
- If ReaderResourceLimitsQos::sample_infos_allocation has a maximum limit, the number of samples returned may also be limited if insufficient 'SampleInfo' resources are available.

If the operation succeeds and the number of samples returned has been limited (by means of a maximum limit, as listed above, or insufficient 'SampleInfo' resources), the call will complete successfully and provide those samples the reader is able to return. The user may need to make additional calls, or return outstanding loaned buffers in the case of insufficient resources, in order to access remaining samples.

In addition to the collection of samples, the read operation also uses a collection of 'SampleInfo' structures (`sample_infos`).

The initial (input) properties of the `data_values` and `sample_infos` collections will determine the precise behavior of this operation. For the purposes of this description the collections are modeled as having three properties:

- the current length (`len`, see 'LoanableCollection::length())'
- the maximum length (`max_len`, see 'LoanableCollection::maximum())'
- whether the collection container owns the memory of the elements within (`owns`, see 'LoanableCollection::has_ownership())'

The initial (input) values of the `len`, `max_len`, and `owns` properties for the `data_values` and `sample_infos` collections govern the behavior of the read operation as specified by the following rules:

1. The values of `len`, `max_len`, and `owns` for the two collections must be identical. Otherwise read will fail with RETCODE_PRECONDITION_NOT_MET.
2. On successful output, the values of `len`, `max_len`, and `owns` will be the same for both collections.
3. If the input max_len == 0 , then the `data_values` and `sample_infos` collections will be filled with elements that are 'loaned' by the DataReader. On output, `owns` will be `false`, `len` will be set to the number of values returned, and `max_len` will be set to a value verifying max_len >= len . The use of this variant allows for zero-copy access to the data and the application will need to return the loan to the DataReader using the 'return_loan' operation.
4. If the input max_len > 0 and the input owns == false , then the read operation will fail with RETCODE_PRECONDITION_NOT_MET. This avoids the potential hard-to-detect memory leaks caused by an application forgetting to return the loan.
5. If input max_len > 0 and the input owns == true , then the read operation will copy the Data values and SampleInfo values into the elements already inside the collections. On output, `owns` will be `true`, `len` will be set to the number of values copied, and `max_len` will remain unchanged. The use of this variant forces a copy but the application can control where the copy is placed and the

application will not need to return the loan. The number of samples copied depends on the values of `max_len` and `max_samples`:

- If max_samples == LENGTH_UNLIMITED , then at most `max_len` values will be copied. The use of this variant lets the application limit the number of samples returned to what the sequence can accommodate.
- If max_samples <= max_len , then at most `max_samples` values will be copied. The use of this variant lets the application limit the number of samples returned to fewer that what the sequence can accommodate.
- If max_samples > max_len , then the read operation will fail with RETCODE_PRECONDITION_NOT_MET. This avoids the potential confusion where the application expects to be able to access up to `max_samples`, but that number can never be returned, even if they are available in the DataReader, because the output sequence cannot accommodate them.

As described above, upon return the `data_values` and `sample_infos` collections may contain elements 'loaned' from the DataReader. If this is the case, the application will need to use the 'return_loan' operation to return the loan once it is no longer using the Data in the collection. Upon return from 'return_loan', the collection will have max_len == 0 and owns == false .

The application can determine whether it is necessary to return the loan or not based on the state of the collections when the read operation was called, or by accessing the `owns` property. However, in many cases it may be simpler to always call 'return_loan', as this operation is harmless (i.e., leaves all elements unchanged) if the collection does not have a loan.

On output, the collection of Data values and the collection of SampleInfo structures are of the same length and are in a one-to-one correspondence. Each SampleInfo provides information, such as the `source_timestamp`, the `sample_state`, `view_state`, and `instance_state`, etc., about the corresponding sample.

Some elements in the returned collection may not have valid data. If the `instance_state` in the SampleInfo is 'NOT_ALIVE_DISPOSED_INSTANCE_STATE' or 'NOT_ALIVE_NO_WRITERS_INSTANCE_STATE', then the last sample for that instance in the collection, that is, the one whose SampleInfo has sample_rank == 0 does not contain valid data. Samples that contain no data do not count towards the limits imposed by the 'ResourceLimitsQosPolicy'.

The act of reading a sample changes its `sample_state` to 'READ_SAMPLE_STATE'. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to be 'NOT_NEW_VIEW_STATE'. It will not affect the `instance_state` of the instance.

If the DataReader has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Important: If the samples "returned" by this method are loaned from the middleware (see 'take' for more information on memory loaning), it is important that their contents not be changed. Because the memory in which the data is stored belongs to the middleware, any modifications made to the data will be seen the next time the same samples are read or taken; the samples will no longer reflect the state that was received from the network.

Parameters

- **`[in,out]`** – data_values A LoanableCollection object where the received data samples will be returned.
- **`[in,out]`** – sample_infos A SampleInfoSeq object where the received sample info will be returned.
- **`[in]`** – max_samples The maximum number of samples to be returned. If the special value 'LENGTH_UNLIMITED' is provided, as many samples will be returned as are

available, up to the limits described above.

- **[in]** – sample_states Only data samples with `sample_state` matching one of these will be returned.
- **[in]** – view_states Only data samples with `view_state` matching one of these will be returned.
- **[in]** – instance_states Only data samples with `instance_state` matching one of these will be returned.

Return type `ReturnCode_t`

Returns Any of the standard return codes.

`read_instance`(**args*)

Access a collection of data samples from the DataReader.

This operation accesses a collection of data values from the DataReader. The behavior is identical to 'read', except that all samples returned belong to the single specified instance whose handle is `a_handle`.

Upon successful completion, the data collection will contain samples all belonging to the same instance. The corresponding 'SampleInfo' verifies 'SampleInfo::instance_handle' == `a_handle`.

This operation is semantically equivalent to the 'read' operation, except in building the collection. The DataReader will check that the sample belongs to the specified instance and otherwise it will not place the sample in the returned collection.

The behavior of this operation follows the same rules as the 'read' operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos`. Similar to 'read', this operation may 'loan' elements to the output collections, which must then be returned by means of 'return_loan'.

If the DataReader has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **[in,out]** – data_values A LoanableCollection object where the received data samples will be returned.
- **[in,out]** – sample_infos A SampleInfoSeq object where the received sample info will be returned.
- **[in]** – max_samples The maximum number of samples to be returned. If the special value 'LENGTH_UNLIMITED' is provided, as many samples will be returned as are available, up to the limits described in the documentation for 'read()'.
- **[in]** – a_handle The specified instance to return samples for. The method will fail with RETCODE_BAD_PARAMETER if the handle does not correspond to an existing data-object known to the DataReader.
- **[in]** – sample_states Only data samples with `sample_state` matching one of these will be returned.
- **[in]** – view_states Only data samples with `view_state` matching one of these will be returned.
- **[in]** – instance_states Only data samples with `instance_state` matching one of these will be returned.

Return type `ReturnCode_t`

Returns Any of the standard return codes.

`read_next_instance`(**args*)

Access a collection of data samples from the DataReader.

This operation accesses a collection of data values from the DataReader where all the samples belong to a single instance. The behavior is similar to 'read_instance', except that the actual instance is not directly specified. Rather, the samples will all belong to the 'next' instance with `instance_handle` 'greater' than the specified 'previous_handle' that has available samples.

This operation implies the existence of a total order 'greater-than' relationship between the instance handles. The specifics of this relationship are not all important and are implementation specific. The important thing is that, according to the middleware, all instances are ordered relative to each other. This ordering is between the instance handles, and should not depend on the state of the instance (e.g. whether it has data or not) and must be defined even for instance handles that do not correspond to instances currently managed by the DataReader. For the purposes of the ordering, it should be 'as if' each instance handle was represented as an integer.

The behavior of this operation is 'as if' the DataReader invoked 'read_instance', passing the smallest `instance_handle` among all the ones that: (a) are greater than `previous_handle`, and (b) have available samples (i.e. samples that meet the constraints imposed by the specified states).

The special value 'HANDLE_NIL' is guaranteed to be 'less than' any valid `instance_handle`. So the use of the parameter value `previous_handle` == 'HANDLE_NIL' will return the samples for the instance which has the smallest `instance_handle` among all the instances that contain available samples.

This operation is intended to be used in an application-driven iteration, where the application starts by passing `previous_handle` == 'HANDLE_NIL', examines the samples returned, and then uses the `instance_handle` returned in the 'SampleInfo' as the value of the `previous_handle` argument to the next call to 'read_next_instance'. The iteration continues until 'read_next_instance' fails with RETCODE_NO_DATA.

Note that it is possible to call the 'read_next_instance' operation with a `previous_handle` that does not correspond to an instance currently managed by the DataReader. This is because as stated earlier the 'greater-than' relationship is defined even for handles not managed by the DataReader. One practical situation where this may occur is when an application is iterating through all the instances, takes all the samples of a 'NOT_ALIVE_NO_WRITERS_INSTANCE_STATE' instance, returns the loan (at which point the instance information may be removed, and thus the handle becomes invalid), and tries to read the next instance.

The behavior of this operation follows the same rules as the 'read' operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos`. Similar to 'read', this operation may 'loan' elements to the output collections, which must then be returned by means of 'return_loan'.

If the DataReader has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **[in,out]** – data_values A LoanableCollection object where the received data samples will be returned.
- **[in,out]** – sample_infos A SampleInfoSeq object where the received sample info will be returned.
- **[in]** – max_samples The maximum number of samples to be returned. If the special value 'LENGTH_UNLIMITED' is provided, as many samples will be returned as are available, up to the limits described in the documentation for 'read()'.
- **[in]** – previous_handle The 'next smallest' instance with a value greater than this value that has available samples will be returned.

- **[in]** – sample_states Only data samples with `sample_state` matching one of these will be returned.
- **[in]** – view_states Only data samples with `view_state` matching one of these will be returned.
- **[in]** – instance_states Only data samples with `instance_state` matching one of these will be returned.

Return type `ReturnCode_t`

Returns Any of the standard return codes.

`read_next_instance_w_condition`(*data_values*, *sample_infos*, *max_samples*, *previous_handle*, *a_condition*)

This operation accesses a collection of Data values from the DataReader. The behavior is identical to 'read_next_instance' except that all samples returned satisfy the specified condition. In other words, on success all returned samples belong to the same instance, and the instance is the instance with 'smallest' `instance_handle` among the ones that verify (a) `instance_handle >= previous_handle` and (b) have samples for which the specified ReadCondition evaluates to TRUE.

Similar to the operation 'read_next_instance' it is possible to call 'read_next_instance_w_condition' with a `previous_handle` that does not correspond to an instance currently managed by the DataReader.

The behavior of the 'read_next_instance_w_condition' operation follows the same rules than the read operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos` collections. Similar to read, the 'read_next_instance_w_condition' operation may 'loan' elements to the output collections which must then be returned by means of 'return_loan'.

If the DataReader has no samples that meet the constraints, the return value will be RETCODE_NO_DATA.

,out] data_values A LoanableCollection object where the received data samples will be returned. ,out] sample_infos A SampleInfoSeq object where the received sample info will be returned. max_samples The maximum number of samples to be returned. If the special value

'LENGTH_UNLIMITED' is provided, as many samples will be returned as are available, up to the limits described in the documentation for 'read()'.

previous_handle The 'next smallest' instance with a value greater than this value that has available samples will be returned.

a_condition A ReadCondition that returned `data_values` must pass

Return type `ReturnCode_t`

Returns Any of the standard return codes.

`read_next_sample`(*data*, *info*)

This operation copies the next, non-previously accessed Data value from the DataReader; the operation also copies the corresponding SampleInfo. The implied order among the samples stored in the DataReader is the same as for the read operation.

The read_next_sample operation is semantically equivalent to the read operation where the input Data sequence has max_length = 1 , the sample_states = NOT_READ_SAMPLE_STATE , the view_states = ANY_VIEW_STATE , and the instance_states = ANY_INSTANCE_STATE .

The read_next_sample operation provides a simplified API to 'read' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the DataReader, the operation will return RETCODE_NO_DATA and nothing is copied

Parameters

- **[out]** – data Data pointer to store the sample
- **[out]** – info SampleInfo pointer to store the sample information

Return type ReturnCode_t

Returns Any of the standard return codes.

read_w_condition(*data_values*, *sample_infos*, *max_samples*, *a_condition*)

This operation accesses via 'read' the samples that match the criteria specified in the ReadCondition. This operation is especially useful in combination with QueryCondition to filter data samples based on the content.

The specified ReadCondition must be attached to the DataReader; otherwise the operation will fail and return RETCODE_PRECONDITION_NOT_MET.

In case the ReadCondition is a 'plain' ReadCondition and not the specialized QueryCondition, the operation is equivalent to calling read and passing as `sample_states`, `view_states` and `instance_states` the value of the corresponding attributes in `a_condition`. Using this operation the application can avoid repeating the same parameters specified when creating the ReadCondition.

The samples are accessed with the same semantics as the read operation. If the DataReader has no samples that meet the constraints, the return value will be RETCODE_NO_DATA.

,out] data_values A LoanableCollection object where the received data samples will be returned. ,out] sample_infos A SampleInfoSeq object where the received sample info will be returned. max_samples The maximum number of samples to be returned. a_condition A ReadCondition that returned `data_values` must pass

Return type ReturnCode_t

Returns Any of the standard return codes.

return_loan(*data_values*, *sample_infos*)

This operation indicates to the DataReader that the application is done accessing the collection of `data_values` and `sample_infos` obtained by some earlier invocation of 'read' or 'take' on the DataReader.

The `data_values` and `sample_infos` must belong to a single related 'pair'; that is, they should correspond to a pair returned from a single call to read or take. The `data_values` and `sample_infos` must also have been obtained from the same DataReader to which they are returned. If either of these conditions is not met, the operation will fail and return RETCODE_PRECONDITION_NOT_MET.

This operation allows implementations of the 'read' and 'take' operations to "loan" buffers from the DataReader to the application and in this manner provide "zero-copy" access to the data. During the loan, the DataReader will guarantee that the data and sample-information are not modified.

It is not necessary for an application to return the loans immediately after the read or take calls. However, as these buffers correspond to internal resources inside the DataReader, the application should not retain them indefinitely.

The use of the 'return_loan' operation is only necessary if the read or take calls "loaned" buffers to the application. This only occurs if the `data_values` and `sample_infos` collections had max_len == 0 at the time read or take was called. The application may also examine the `owns` property of the collection to determine if there is an outstanding loan. However, calling 'return_loan' on a collection that does not have a loan is safe and has no side effects.

If the collections had a loan, upon return from return_loan the collections will have max_len == 0 .

Parameters

- **[in,out]** – data_values A LoanableCollection object where the received data samples were obtained from an earlier invocation of read or take on this DataReader.
- **[in,out]** – sample_infos A SampleInfoSeq object where the received sample infos were obtained from an earlier invocation of read or take on this DataReader.

Return type ReturnCode_t

Returns Any of the standard return codes.

set_listener(**args*)

set_qos(*qos*)

Setter for the DataReaderQos.

Parameters **[in]** – qos new value for the DataReaderQos.

Return type ReturnCode_t

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

take(**args*)

Access a collection of data samples from the DataReader.

This operation accesses a collection of data-samples from the DataReader and a corresponding collection of SampleInfo structures, and 'removes' them from the DataReader. The operation will return either a 'list' of samples or else a single sample. This is controlled by the 'PresentationQosPolicy' using the same logic as for the 'read' operation.

The act of taking a sample removes it from the DataReader so it cannot be 'read' or 'taken' again. If the sample belongs to the most recent generation of the instance, it will also set the `view_state` of the instance to NOT_NEW. It will not affect the `instance_state` of the instance.

The behavior of the take operation follows the same rules than the 'read' operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos` collections. Similar to 'read', the take operation may 'loan' elements to the output collections which must then be returned by means of 'return_loan'. The only difference with 'read' is that, as stated, the samples returned by take will no longer be accessible to successive calls to read or take.

If the DataReader has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

Parameters

- **[in,out]** – data_values A LoanableCollection object where the received data samples will be returned.
- **[in,out]** – sample_infos A SampleInfoSeq object where the received sample info will be returned.
- **[in]** – max_samples The maximum number of samples to be returned. If the special value 'LENGTH_UNLIMITED' is provided, as many samples will be returned as are available, up to the limits described in the documentation for 'read()'.
- **[in]** – sample_states Only data samples with `sample_state` matching one of these will be returned.
- **[in]** – view_states Only data samples with `view_state` matching one of these will be returned.
- **[in]** – instance_states Only data samples with `instance_state` matching one of these will be returned.

Return type ReturnCode_t

Returns Any of the standard return codes.

take_instance(**args*)

Access a collection of data samples from the DataReader.

This operation accesses a collection of data values from the DataReader and 'removes' them from the DataReader.

This operation has the same behavior as 'read_instance', except that the samples are 'taken' from the DataReader such that they are no longer accessible via subsequent 'read' or 'take' operations.

The behavior of this operation follows the same rules as the 'read' operation regarding the pre-conditions and post-conditions for the data_values and sample_infos. Similar to 'read', this operation may 'loan' elements to the output collections, which must then be returned by means of 'return_loan'.

If the DataReader has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

,out] data_values A LoanableCollection object where the received data samples will be returned. ,out] sample_infos A SampleInfoSeq object where the received sample info will be returned. max_samples The maximum number of samples to be returned. If the special value

'LENGTH_UNLIMITED' is provided, as many samples will be returned as are available, up to the limits described in the documentation for 'read()'.

a_handle The specified instance to return samples for. The method will fail with RETCODE_BAD_PARAMETER if the handle does not correspond to an existing data-object known to the DataReader.

sample_states Only data samples with sample_state matching one of these will be returned. view_states Only data samples with view_state matching one of these will be returned. instance_states Only data samples with instance_state matching one of these will be returned.

Return type ReturnCode_t

Returns Any of the standard return codes.

take_next_instance(**args*)

Access a collection of data samples from the DataReader.

This operation accesses a collection of data values from the DataReader and 'removes' them from the DataReader.

This operation has the same behavior as 'read_next_instance', except that the samples are 'taken' from the DataReader such that they are no longer accessible via subsequent 'read' or 'take' operations.

Similar to the operation 'read_next_instance', it is possible to call this operation with a previous_handle that does not correspond to an instance currently managed by the DataReader.

The behavior of this operation follows the same rules as the 'read' operation regarding the pre-conditions and post-conditions for the data_values and sample_infos. Similar to 'read', this operation may 'loan' elements to the output collections, which must then be returned by means of 'return_loan'.

If the DataReader has no samples that meet the constraints, the operations fails with RETCODE_NO_DATA.

,out] data_values A LoanableCollection object where the received data samples will be returned. ,out] sample_infos A SampleInfoSeq object where the received sample info will be returned. max_samples The maximum number of samples to be returned. If the special value

'LENGTH_UNLIMITED' is provided, as many samples will be returned as are available, up to the limits described in the documentation for 'read()'.

previous_handle The 'next smallest' instance with a value greater than this value that has available samples will be returned.

sample_states Only data samples with `sample_state` matching one of these will be returned. view_states Only data samples with `view_state` matching one of these will be returned. instance_states Only data samples with `instance_state` matching one of these will be returned.

Return type `ReturnCode_t`

Returns Any of the standard return codes.

`take_next_instance_w_condition`(*data_values*, *sample_infos*, *max_samples*, *previous_handle*, *a_condition*)

This operation accesses a collection of Data values from the DataReader. The behavior is identical to 'read_next_instance' except that all samples returned satisfy the specified condition. In other words, on success all returned samples belong to the same instance, and the instance is the instance with 'smallest' `instance_handle` among the ones that verify (a) `instance_handle >= previous_handle` and (b) have samples for which the specified ReadCondition evaluates to TRUE.

Similar to the operation 'read_next_instance' it is possible to call 'read_next_instance_w_condition' with a `previous_handle` that does not correspond to an instance currently managed by the DataReader.

The behavior of the 'read_next_instance_w_condition' operation follows the same rules than the read operation regarding the pre-conditions and post-conditions for the `data_values` and `sample_infos` collections. Similar to read, the 'read_next_instance_w_condition' operation may 'loan' elements to the output collections which must then be returned by means of 'return_loan'.

If the DataReader has no samples that meet the constraints, the return value will be RETCODE_NO_DATA

,out] data_values A LoanableCollection object where the received data samples will be returned. ,out] sample_infos A SampleInfoSeq object where the received sample info will be returned. max_samples The maximum number of samples to be returned. If the special value

'LENGTH_UNLIMITED' is provided, as many samples will be returned as are available, up to the limits described in the documentation for 'read()'.

previous_handle The 'next smallest' instance with a value greater than this value that has available samples will be returned.

a_condition A ReadCondition that returned `data_values` must pass

Return type `ReturnCode_t`

Returns Any of the standard return codes.

`take_next_sample`(*data*, *info*)

This operation copies the next, non-previously accessed Data value from the DataReader and 'removes' it from the DataReader so it is no longer accessible. The operation also copies the corresponding SampleInfo.

This operation is analogous to 'read_next_sample' except for the fact that the sample is 'removed' from the DataReader.

This operation is semantically equivalent to the 'take' operation where the input sequence has max_length = 1 , the sample_states = NOT_READ_SAMPLE_STATE , the view_states = ANY_VIEW_STATE , and the instance_states = ANY_INSTANCE_STATE .

This operation provides a simplified API to 'take' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the DataReader, the operation will return RETCODE_NO_DATA and nothing is copied.

Parameters

- **[out]** – data Data pointer to store the sample
- **[out]** – info SampleInfo pointer to store the sample information

Return type ReturnCode_t

Returns Any of the standard return codes.

take_w_condition(*data_values*, *sample_infos*, *max_samples*, *a_condition*)

This operation is analogous to 'read_w_condition' except it accesses samples via the 'take' operation.

The specified ReadCondition must be attached to the DataReader; otherwise the operation will fail and return RETCODE_PRECONDITION_NOT_MET.

The samples are accessed with the same semantics as the 'take' operation.

This operation is especially useful in combination with QueryCondition to filter data samples based on the content.

If the DataReader has no samples that meet the constraints, the return value will be RETCODE_NO_DATA.

,out] data_values A LoanableCollection object where the received data samples will be returned. ,out] sample_infos A SampleInfoSeq object where the received sample info will be returned. max_samples The maximum number of samples to be returned. If the special value

'LENGTH_UNLIMITED' is provided, as many samples will be returned as are.

a_condition A ReadCondition that returned `data_values` must pass

Return type ReturnCode_t

Returns Any of the standard return codes.

property thisown

The membership flag

type()

Getter for the data type.

Return type *TypeSupport*

Returns TypeSupport associated to the DataReader.

wait_for_historical_data(*max_wait*)

NOT YET IMPLEMENTED

Method to block the current thread until an unread message is available.

max_wait Max blocking time for this operation. :rtype: ReturnCode_t :return: RETCODE_OK if there is new unread message, ReturnCode_t::RETCODE_TIMEOUT if timeout Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

wait_for_unread_message(*timeout*)

Method to block the current thread until an unread message is available.

Parameters **[in]** – timeout Max blocking time for this operation.

Return type boolean

Returns true if there is new unread message, false if timeout

DataReaderListener

class fastdds.**DataReaderListener**

Class DataReaderListener, it should be used by the end user to implement specific callbacks to certain actions.

on_data_available(*reader*)

Virtual function to be implemented by the user containing the actions to be performed when new Data Messages are received.

Parameters **reader** (*DataReader*) – DataReader

on_liveliness_changed(*reader*, *status*)

Method called when the liveliness status associated to a subscriber changes

Parameters

- **reader** (*DataReader*) – The DataReader
- **status** (*eprosima::fastrtps::LivelinessChangedStatus*) – The liveliness changed status

on_requested_deadline_missed(*reader*, *status*)

Virtual method to be called when a topic misses the deadline period

Parameters

- **reader** (*DataReader*) – DataReader
- **status** (*eprosima::fastrtps::RequestedDeadlineMissedStatus*) – The requested deadline missed status

on_requested_incompatible_qos(*reader*, *status*)

Method called an incompatible QoS was requested.

Parameters

- **reader** (*DataReader*) – The DataReader
- **status** (RequestedIncompatibleQosStatus) – The requested incompatible QoS status

on_sample_lost(*reader*, *status*)

Method called when a sample was lost.

Parameters

- **reader** (*DataReader*) – The DataReader
- **status** (SampleLostStatus) – The sample lost status

on_sample_rejected(*reader*, *status*)

Method called when a sample was rejected.

Parameters

- **reader** (*DataReader*) – The DataReader
- **status** (*eprosima::fastrtps::SampleRejectedStatus*) – The rejected status

on_subscription_matched(*reader*, *info*)

Virtual method to be called when the subscriber is matched with a new Writer (or unmatched); i.e., when a writer publishing in the same topic is discovered.

Parameters

- **reader** (*DataReader*) – DataReader

- **info** (*SubscriptionMatchedStatus*) – The subscription matched status

property thisown
The membership flag

DataReaderQos

class fastdds.**DataReaderQos**
Class DataReaderQos, containing all the possible Qos that can be set for a determined DataReader. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

data_sharing(**args*)
Overload 1:

Getter for DataSharingQosPolicy

Return type *DataSharingQosPolicy*

Returns DataSharingQosPolicy reference

Overload 2:

Getter for DataSharingQosPolicy

Return type *DataSharingQosPolicy*

Returns DataSharingQosPolicy reference

Overload 3:

Setter for DataSharingQosPolicy

Parameters **data_sharing** (*DataSharingQosPolicy*) – new value for the DataSharingQosPolicy

deadline(**args*)
Overload 1:

Getter for DeadlineQosPolicy

Return type *DeadlineQosPolicy*

Returns DeadlineQosPolicy reference

Overload 2:

Getter for DeadlineQosPolicy

Return type *DeadlineQosPolicy*

Returns DeadlineQosPolicy const reference

Overload 3:

Setter for DeadlineQosPolicy

Parameters **new_value** (*DeadlineQosPolicy*) – new value for the DeadlineQosPolicy

destination_order(**args*)

Overload 1:

Getter for DestinationOrderQosPolicy

Return type *DestinationOrderQosPolicy*

Returns DestinationOrderQosPolicy reference

Overload 2:

Getter for DestinationOrderQosPolicy

Return type *DestinationOrderQosPolicy*

Returns DestinationOrderQosPolicy const reference

Overload 3:

Setter for DestinationOrderQosPolicy

Parameters **new_value** (*DestinationOrderQosPolicy*) – new value for the DestinationOrderQosPolicy

durability(**args*)

Overload 1:

Getter for DurabilityQosPolicy

Return type *DurabilityQosPolicy*

Returns DurabilityQosPolicy reference

Overload 2:

Getter for DurabilityQosPolicy

Return type *DurabilityQosPolicy*

Returns DurabilityQosPolicy const reference

Overload 3:

Setter for DurabilityQosPolicy

Parameters **new_value** (*DurabilityQosPolicy*) – new value for the DurabilityQosPolicy

durability_service(**args*)

Overload 1:

Getter for DurabilityServiceQosPolicy

Return type *DurabilityServiceQosPolicy*

Returns DurabilityServiceQosPolicy reference

Overload 2:

Getter for DurabilityServiceQosPolicy

Return type *DurabilityServiceQosPolicy*

Returns DurabilityServiceQosPolicy const reference

Overload 3:

Setter for DurabilityServiceQosPolicy

Parameters **new_value** (*DurabilityServiceQosPolicy*) – new value for the Durability-ServiceQosPolicy

endpoint(**args*)

Overload 1:

Getter for RTPSEndpointQos

Return type *RTPSEndpointQos*

Returns RTPSEndpointQos reference

Overload 2:

Getter for RTPSEndpointQos

Return type *RTPSEndpointQos*

Returns RTPSEndpointQos const reference

Overload 3:

Setter for RTPSEndpointQos

Parameters **new_value** (*RTPSEndpointQos*) – new value for the RTPSEndpointQos

expects_inline_qos(**args*)

Overload 1:

Getter for expectsInlineQos

Return type boolean

Returns expectsInlineQos

Overload 2:

Setter for expectsInlineQos

Parameters **new_value** (*boolean*) – new value for the expectsInlineQos

get_readerqos(*sqos*)

history(**args*)

Overload 1:

Getter for HistoryQosPolicy

Return type *HistoryQosPolicy*

Returns HistoryQosPolicy reference

Overload 2:

Getter for HistoryQosPolicy

Return type *HistoryQosPolicy*

Returns HistoryQosPolicy const reference

Overload 3:

Setter for HistoryQosPolicy

Parameters **new_value** (*HistoryQosPolicy*) – new value for the HistoryQosPolicy

latency_budget(**args*)

Overload 1:

Getter for LatencyBudgetQosPolicy

Return type *LatencyBudgetQosPolicy*

Returns LatencyBudgetQosPolicy reference

Overload 2:

Getter for LatencyBudgetQosPolicy

Return type *LatencyBudgetQosPolicy*

Returns LatencyBudgetQosPolicy const reference

Overload 3:

Setter for LatencyBudgetQosPolicy

Parameters **new_value** (*LatencyBudgetQosPolicy*) – new value for the LatencyBudgetQosPolicy

lifespan(**args*)

Overload 1:

Getter for LifespanQosPolicy

Return type *LifespanQosPolicy*

Returns LifespanQosPolicy reference

Overload 2:

Getter for LifespanQosPolicy

Return type *LifespanQosPolicy*

Returns LifespanQosPolicy const reference

Overload 3:

Setter for LifespanQosPolicy

Parameters **new_value** (*LifespanQosPolicy*) – new value for the LifespanQosPolicy

liveliness(**args*)

Overload 1:

Getter for LivelinessQosPolicy

Return type *LivelinessQosPolicy*

Returns LivelinessQosPolicy reference

Overload 2:

Getter for LivelinessQosPolicy

Return type *LivelinessQosPolicy*

Returns LivelinessQosPolicy const reference

Overload 3:

Setter for LivelinessQosPolicy

Parameters **new_value** (*LivelinessQosPolicy*) – new value for the LivelinessQosPolicy

ownership(**args*)

Overload 1:

Getter for OwnershipQosPolicy

Return type *OwnershipQosPolicy*

Returns OwnershipQosPolicy reference

Overload 2:

Getter for OwnershipQosPolicy

Return type *OwnershipQosPolicy*

Returns OwnershipQosPolicy const reference

Overload 3:

Setter for OwnershipQosPolicy

Parameters **new_value** (*OwnershipQosPolicy*) – new value for the OwnershipQosPolicy

properties(**args*)

Overload 1:

Getter for PropertyPolicyQos

Return type PropertyPolicyQos

Returns PropertyPolicyQos reference

Overload 2:

Getter for PropertyPolicyQos

Return type PropertyPolicyQos

Returns PropertyPolicyQos const reference

Overload 3:

Setter for PropertyPolicyQos

Parameters **new_value** (PropertyPolicyQos) – new value for the PropertyPolicyQos

reader_data_lifecycle(**args*)

Overload 1:

Getter for ReaderDataLifecycleQosPolicy

Return type *ReaderDataLifecycleQosPolicy*

Returns ReaderDataLifecycleQosPolicy reference

Overload 2:

Getter for ReaderDataLifecycleQosPolicy

Return type *ReaderDataLifecycleQosPolicy*

Returns ReaderDataLifecycleQosPolicy const reference

Overload 3:

Setter for ReaderDataLifecycleQosPolicy

Parameters **new_value** (*ReaderDataLifecycleQosPolicy*) – new value for the ReaderDataLifecycleQosPolicy

reader_resource_limits(**args*)

Overload 1:

Getter for ReaderResourceLimitsQos

Return type *ReaderResourceLimitsQos*

Returns ReaderResourceLimitsQos reference

Overload 2:

Getter for ReaderResourceLimitsQos

Return type *ReaderResourceLimitsQos*

Returns ReaderResourceLimitsQos const reference

Overload 3:

Setter for ReaderResourceLimitsQos

Parameters **new_value** (*ReaderResourceLimitsQos*) – new value for the ReaderResourceLimitsQos

reliability(**args*)

Overload 1:

Getter for ReliabilityQosPolicy

Return type *ReliabilityQosPolicy*

Returns ReliabilityQosPolicy reference

Overload 2:

Getter for ReliabilityQosPolicy

Return type *ReliabilityQosPolicy*

Returns ReliabilityQosPolicy const reference

Overload 3:

Setter for ReliabilityQosPolicy

Parameters **new_value** (*ReliabilityQosPolicy*) – new value for the ReliabilityQosPolicy

reliable_reader_qos(**args*)

Overload 1:

Getter for RTPSReliableReaderQos

Return type *RTPSReliableReaderQos*

Returns RTPSReliableReaderQos reference

Overload 2:

Getter for RTPSReliableReaderQos

Return type *RTPSReliableReaderQos*

Returns RTPSReliableReaderQos const reference

Overload 3:

Setter for RTPSReliableReaderQos

Parameters **new_value** (*RTPSReliableReaderQos*) – new value for the RTPSReliableReaderQos

resource_limits(**args*)

Overload 1:

Getter for ResourceLimitsQosPolicy

Return type *ResourceLimitsQosPolicy*

Returns ResourceLimitsQosPolicy reference

Overload 2:

Getter for ResourceLimitsQosPolicy

Return type *ResourceLimitsQosPolicy*

Returns ResourceLimitsQosPolicy const reference

Overload 3:

Setter for ResourceLimitsQosPolicy

Parameters **new_value** (*ResourceLimitsQosPolicy*) – new value for the ResourceLimitsQosPolicy

property thisown
The membership flag

time_based_filter(**args*)
Overload 1:

Getter for TimeBasedFilterQosPolicy

Return type *TimeBasedFilterQosPolicy*

Returns TimeBasedFilterQosPolicy reference

Overload 2:

Getter for TimeBasedFilterQosPolicy

Return type *TimeBasedFilterQosPolicy*

Returns TimeBasedFilterQosPolicy const reference

Overload 3:

Setter for TimeBasedFilterQosPolicy

Parameters **new_value** (*TimeBasedFilterQosPolicy*) – new value for the TimeBasedFilterQosPolicy

type_consistency(**args*)
Overload 1:

Getter for TypeConsistencyQos

Return type *TypeConsistencyQos*

Returns TypeConsistencyQos reference

Overload 2:

Getter for TypeConsistencyQos

Return type *TypeConsistencyQos*

Returns TypeConsistencyQos const reference

Overload 3:

Setter for TypeConsistencyQos

Parameters **new_value** (*TypeConsistencyQos*) – new value for the TypeConsistencyQos

user_data(**args*)

Overload 1:

Getter for UserDataQosPolicy

Return type *UserDataQosPolicy*

Returns UserDataQosPolicy reference

Overload 2:

Getter for UserDataQosPolicy

Return type *UserDataQosPolicy*

Returns UserDataQosPolicy const reference

Overload 3:

Setter for UserDataQosPolicy

Parameters **new_value** (*UserDataQosPolicy*) – new value for the UserDataQosPolicy

class fastdds.**DATAREADER_QOS_DEFAULT**(**args: Any*, ***kwargs: Any*)

InstanceStateKind

class fastdds.**ALIVE_INSTANCE_STATE**(**args: Any*, ***kwargs: Any*)

class fastdds.**NOT_ALIVE_DISPOSED_INSTANCE_STATE**(**args: Any*, ***kwargs: Any*)

class fastdds.**NOT_ALIVE_NO_WRITERS_INSTANCE_STATE**(**args: Any*, ***kwargs: Any*)

ReaderResourceLimitsQos

class fastdds.**ReaderResourceLimitsQos**

Qos Policy to configure the limit of the reader resources

clear()

property matched_publisher_allocation

Matched publishers allocation limits.

property max_samples_per_read

Maximum number of samples to return on a single call to read / take.

This attribute is a signed integer to be consistent with the `max_samples` argument of 'DataReader' methods, but should always have a strict positive value. Bear in mind that a big number here may cause the creation of the DataReader to fail due to pre-allocation of internal resources.

Default value: 32.

property outstanding_reads_allocation

Loaned collections allocation limits.

property sample_infos_allocation

SampleInfo allocation limits.

property thisown

The membership flag

RTPSReliableReaderQos

class fastdds.**RTPSReliableReaderQos**

Qos Policy to configure the DisablePositiveACKsQos and the reader attributes

clear()

property disable_positive_ACKs

Control the sending of positive ACKs

property thisown

The membership flag

property times

Times associated with the Reliable Readers events.

SampleInfo

class fastdds.**SampleInfo**

SampleInfo is the information that accompanies each sample that is 'read' or 'taken.'

property absolute_generation_rank

the generation difference between the time the sample was received, and the time the most recent sample was received. The most recent sample used for the calculation may or may not be in the returned collection

property disposed_generation_count

number of times the instance had become alive after it was disposed

property generation_rank

the generation difference between the time the sample was received, and the time the most recent sample in the collection was received.

property instance_handle
identifies locally the corresponding instance

property instance_state
indicates whether the instance is currently in existence or, if it has been disposed, the reason why it was disposed.

property no_writers_generation_count
number of times the instance had become alive after it was disposed because no writers

property publication_handle
identifies locally the DataWriter that modified the instance Is the same InstanceHandle_t that is returned by the operation get_matched_publications on the DataReader

property reception_timestamp
time provided by the DataReader when the sample was added to its history

property related_sample_identity
Related Sample Identity (Extension for RPC)

property sample_identity
Sample Identity (Extension for RPC)

property sample_rank
number of samples related to the same instance that follow in the collection

property sample_state
indicates whether or not the corresponding data sample has already been read

property source_timestamp
time provided by the DataWriter when the sample was written

property thisown
The membership flag

property valid_data
whether the DataSample contains data or is only used to communicate of a change in the instance

property view_state
indicates whether the DataReader has already seen samples for the most-current generation of the related instance.

SampleStateKind

class fastdds.**READ_SAMPLE_STATE**(**args: Any*, ***kwargs: Any*)

class fastdds.**NOT_READ_SAMPLE_STATE**(**args: Any*, ***kwargs: Any*)

Subscriber

class fastdds.**Subscriber**(**args*, ***kwargs*)
Class Subscriber, contains the public API that allows the user to control the reception of messages. This class should not be instantiated directly. DomainRTPSParticipant class should be used to correctly create this element.

begin_access()
Indicates that the application is about to access the data samples in any of the DataReader objects attached to the Subscriber.

Return type ReturnCode_t

Returns RETCODE_OK

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

static copy_from_topic_qos(*reader_qos*, *topic_qos*)
Copies TopicQos into the corresponding DataReaderQos

, out] reader_qos topic_qos :rtype: ReturnCode_t :return: RETCODE_OK if successful, an error code otherwise Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

create_datareader(**args*)
This operation creates a DataReader. The returned DataReader will be attached and belong to the Subscriber.

Parameters

- **topic** (*TopicDescription*) – Topic the DataReader will be listening.
- **reader_qos** (*DataReaderQos*) – QoS of the DataReader.
- **listener** (*DataReaderListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all).

Return type *DataReader*

Returns Pointer to the created DataReader. nullptr if failed.

create_datareader_with_profile(**args*)
This operation creates a DataReader. The returned DataReader will be attached and belongs to the Subscriber.

Parameters

- **topic** (*TopicDescription*) – Topic the DataReader will be listening.
- **profile_name** (*string*) – DataReader profile name.
- **listener** (*DataReaderListener*) – Pointer to the listener (default: nullptr)
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all).

Return type *DataReader*

Returns Pointer to the created DataReader. nullptr if failed.

delete_contained_entities()
Deletes all contained DataReaders. If the DataReaders have any QueryCondition or ReadCondition, they are deleted before the DataReader itself.

Return type ReturnCode_t

Returns RETCODE_OK if successful, an error code otherwise

delete_datareader(*reader*)
This operation deletes a DataReader that belongs to the Subscriber.

The delete_datareader operation must be called on the same Subscriber object used to create the DataReader. If delete_datareader is called on a different Subscriber, the operation will have no effect and it will return an error.

Parameters **reader** (*DataReader*) – DataReader to delete

Return type ReturnCode_t

Returns RETCODE_PRECONDITION_NOT_MET if the datareader does not belong to this subscriber, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

enable()
This operation enables the Subscriber

Return type ReturnCode_t

Returns RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the participant creating this Subscriber is not enabled.

end_access()
Indicates that the application has finished accessing the data samples in DataReader objects managed by the Subscriber.

Return type ReturnCode_t

Returns RETCODE_OK

Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_datareader_qos_from_profile(*profile_name*, *qos*)
Fills the DataReaderQos with the values of the XML profile.

Parameters

- **profile_name** (*string*) – DataReader profile name.
- **qos** (*DataReaderQos*) – DataReaderQos object where the qos is returned.

Return type ReturnCode_t

Returns RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

get_datareaders(**args*)
Overload 1:

This operation allows the application to access the DataReader objects.

Parameters **readers** (std::vector< eprosima::fastdds::dds::DataReader *,std::allocator< eprosima::fastdds::dds::DataReader * > >) – Vector of DataReader where the list of existing readers is returned

Return type ReturnCode_t

Returns RETCODE_OK

Overload 2:

This operation allows the application to access the DataReader objects that contain samples with the specified sample_states, view_states, and instance_states.

readers Vector of DataReader where the list of existing readers is returned :type sample_states: std::vector< eprosima::fastdds::dds::SampleStateKind,std::allocator< eprosima::fastdds::dds::SampleStateKind > > :param sample_states: Vector of SampleStateKind :type view_states: std::vector< eprosima::fastdds::dds::ViewStateKind,std::allocator< eprosima::fastdds::dds::ViewStateKind > > :param view_states: Vector of ViewStateKind :type instance_states: std::vector<

eprosima::fastdds::dds::InstanceStateKind,std::allocator< eprosima::fastdds::dds::InstanceStateKind > > :param instance_states: Vector of InstanceStateKind :rtype: `ReturnCode_t` :return: RETCODE_OK Warning: Not supported yet. Currently returns RETCODE_UNSUPPORTED

get_default_datareader_qos(**args*)

Overload 1:

This operation returns the default value of the DataReader QoS, that is, the QoS policies which will be used for newly created DataReader entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Return type *DataReaderQos*

Returns Current default DataReaderQos.

Overload 2:

This operation returns the default value of the DataReader QoS, that is, the QoS policies which will be used for newly created DataReader entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Return type *DataReaderQos*

Returns Current default DataReaderQos.

Overload 3:

This operation retrieves the default value of the DataReader QoS, that is, the QoS policies which will be used for newly created DataReader entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Parameters **qos** (*DataReaderQos*) – DataReaderQos where the default_qos is returned

Return type `ReturnCode_t`

Returns RETCODE_OK

get_instance_handle()

Returns the Subscriber's handle.

Return type `InstanceHandle_t`

Returns InstanceHandle of this Subscriber.

get_listener()

Retrieves the attached SubscriberListener.

Return type *SubscriberListener*

Returns Pointer to the SubscriberListener

get_participant()
This operation returns the DomainParticipant to which the Subscriber belongs.

Return type *DomainParticipant*

Returns DomainParticipant Pointer

get_qos(**args*)
Overload 1:

Allows accessing the Subscriber Qos.

Return type *SubscriberQos*

Returns SubscriberQos reference

Overload 2:

Retrieves the Subscriber Qos.

Parameters **qos** (*SubscriberQos*) – SubscriberQos where the qos is returned

Return type ReturnCode_t

Returns RETCODE_OK

has_datareaders()
This operation checks if the subscriber has DataReaders

Return type boolean

Returns true if the subscriber has one or several DataReaders, false in other case

lookup_datareader(*topic_name*)
This operation retrieves a previously-created DataReader belonging to the Subscriber that is attached to a Topic with a matching topic_name. If no such DataReader exists, the operation will return nullptr.

If multiple DataReaders attached to the Subscriber satisfy this condition, then the operation will return one of them. It is not specified which one.

Parameters **topic_name** (*string*) – Name of the topic associated to the DataReader

Return type *DataReader*

Returns Pointer to a previously created DataReader created on a Topic with that topic_name

notify_datareaders()
This operation invokes the operation on_data_available on the DataReaderListener objects attached to contained DataReader entities.

This operation is typically invoked from the on_data_on_readers operation in the SubscriberListener. That way the SubscriberListener can delegate to the DataReaderListener objects the handling of the data.

Return type ReturnCode_t

Returns RETCODE_OK

set_default_datareader_qos(*qos*)

This operation sets a default value of the DataReader QoS policies which will be used for newly created DataReader entities in the case where the QoS policies are defaulted in the create_datareader operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value DATAREADER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_datareader_qos operation had never been called.

Parameters **qos** (*DataReaderQos*) – new value for DataReaderQos to set as default

Return type ReturnCode_t

Returns RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

set_listener(**args*)

Overload 1:

Modifies the SubscriberListener, sets the mask to StatusMask::all()

Parameters **listener** (*SubscriberListener*) – new value for SubscriberListener

Return type ReturnCode_t

Returns RETCODE_OK

Overload 2:

Modifies the SubscriberListener.

Parameters

- **listener** (*SubscriberListener*) – new value for the SubscriberListener
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to.

Return type ReturnCode_t

Returns RETCODE_OK

set_qos(*qos*)

Allows modifying the Subscriber Qos. The given Qos must be supported by the SubscriberQos.

Parameters **qos** (*SubscriberQos*) – new value for SubscriberQos

Return type ReturnCode_t

Returns RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

property thisown

The membership flag

SubscriberListener

class fastdds.**SubscriberListener**

Class SubscriberListener, it should be used by the end user to implement specific callbacks to certain actions. It also inherits all DataReaderListener callbacks.

on_data_on_readers(*sub*)

Virtual function to be implemented by the user containing the actions to be performed when a new Data Message is available on any reader.

Parameters **sub** (*Subscriber*) – Subscriber

property thisown

The membership flag

SubscriberQos

class fastdds.**SubscriberQos**

Class SubscriberQos, contains all the possible Qos that can be set for a determined Subscriber. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

entity_factory(**args*)

Overload 1:

Getter for EntityFactoryQosPolicy

Return type *EntityFactoryQosPolicy*

Returns EntityFactoryQosPolicy reference

Overload 2:

Getter for EntityFactoryQosPolicy

Return type *EntityFactoryQosPolicy*

Returns EntityFactoryQosPolicy reference

Overload 3:

Setter for EntityFactoryQosPolicy

Parameters **entity_factory** (*EntityFactoryQosPolicy*) – new value for the EntityFactoryQosPolicy

group_data(**args*)

Overload 1:

Getter for GroupDataQosPolicy

Return type *GroupDataQosPolicy*

Returns GroupDataQosPolicy reference

Overload 2:

Getter for GroupDataQosPolicy

Return type *GroupDataQosPolicy*

Returns GroupDataQosPolicy reference

Overload 3:

Setter for GroupDataQosPolicy

Parameters **group_data** (*GroupDataQosPolicy*) – new value for the GroupDataQosPolicy

partition(**args*)

Overload 1:

Getter for PartitionQosPolicy

Return type *PartitionQosPolicy*

Returns PartitionQosPolicy reference

Overload 2:

Getter for PartitionQosPolicy

Return type *PartitionQosPolicy*

Returns PartitionQosPolicy reference

Overload 3:

Setter for PartitionQosPolicy

Parameters **partition** (*PartitionQosPolicy*) – new value for the PartitionQosPolicy

presentation(**args*)

Overload 1:

Getter for PresentationQosPolicy

Return type *PresentationQosPolicy*

Returns PresentationQosPolicy reference

Overload 2:

Getter for PresentationQosPolicy

Return type *PresentationQosPolicy*

Returns PresentationQosPolicy reference

Overload 3:

Setter for PresentationQosPolicy

Parameters **presentation** (*PresentationQosPolicy*) – new value for the PresentationQosPolicy

property thisown
The membership flag

class fastdds.**SUBSCRIBER_QOS_DEFAULT**(**args: Any*, ***kwargs: Any*)

TypeConsistencyQos

class fastdds.**TypeConsistencyQos**
Qos Policy to configure the XTypes Qos associated to the DataReader

clear()
Clears the QosPolicy object

property representation
Data Representation Qos.

property thisown
The membership flag

property type_consistency
Type consistency enforcement Qos.

ViewStateKind

class fastdds.**NEW_VIEW_STATE**(**args: Any*, ***kwargs: Any*)

class fastdds.**NOT_NEW_VIEW_STATE**(**args: Any*, ***kwargs: Any*)

Topic

Topic

class fastdds.**Topic**(**args*, ***kwargs*)

Class Topic, represents the fact that both publications and subscriptions are tied to a single data-type

get_impl()

get_listener()

Retrieves the attached TopicListener.

Return type *TopicListener*

Returns pointer to TopicListener

get_participant()

Getter for the DomainParticipant

Return type *DomainParticipant*

Returns DomainParticipant pointer

get_qos(**args*)

Overload 1:

Allows accessing the Topic Qos.

Return type *TopicQos*

Returns reference to TopicQos

Overload 2:

Retrieves the Topic Qos.

Parameters **qos** (*TopicQos*) – TopicQos where the qos is returned

Return type ReturnCode_t

Returns RETCODE_OK

set_listener(**args*)

Modifies the TopicListener.

Parameters

- **listener** (*TopicListener*) – new value for the TopicListener
- **mask** (*StatusMask*) – StatusMask that holds statuses the listener responds to (default: all).

Return type ReturnCode_t

Returns RETCODE_OK

set_qos(*qos*)

Allows modifying the Topic Qos. The given Qos must be supported by the Topic.

Parameters **qos** (*TopicQos*) – new TopicQos value to set for the Topic.

property thisown
The membership flag

TopicDataType

class fastdds.**TopicDataType**(**args*, ***kwargs*)
Class TopicDataType used to provide the DomainRTPSParticipant with the methods to serialize, deserialize and get the key of a specific data type. The user should created a class that inherits from this one, where Serialize and deserialize methods MUST be implemented.

auto_fill_type_information(**args*)
Overload 1:

Get the type information auto-fill configuration

Return type boolean

Returns true if the type information should be auto-filled

Overload 2:

Set type information auto-fill configuration

Parameters **auto_fill_type_information** (*boolean*) – new value to set

auto_fill_type_object(**args*)
Overload 1:

Get the type object auto-fill configuration

Return type boolean

Returns true if the type object should be auto-filled

Overload 2:

Set the type object auto-fill configuration

Parameters **auto_fill_type_object** (*boolean*) – new value to set

construct_sample(*memory*)
Construct a sample on a memory location.

Parameters **memory** (*void*) – Pointer to the memory location where the sample should be constructed.

Return type boolean

Returns whether this type supports in-place construction or not.

createData()
Create a Data Type.

Return type void

Returns Void pointer to the created object.

deleteData(*data*)
Remove a previously created object.

Parameters **data** (*void*) – Pointer to the created Data.

deserialize(*payload*, *data*)
Deserialize method, it should be implemented by the user, since it is abstract.

payload Pointer to the payload data Pointer to the data :rtype: boolean :return: True if correct.

getKey(*data*, *ihandle*, *force_md5=False*)
Get the key associated with the data.

data Pointer to the data. ihandle Pointer to the Handle. force_md5 Force MD5 checking. :rtype: boolean :return: True if correct.

getName()
Get topic data type name

Return type string

Returns Topic data type name

getSerializedSizeProvider(**args*)
Overload 1:

Returns a function which can be used to calculate the serialized size of the provided data.

data Pointer to data. :rtype: std::function< uint32_t () > :return: Functor which calculates the serialized size of the data.

Overload 2:

Returns a function which can be used to calculate the serialized size of the provided data.

data Pointer to data. data_representation Representation that should be used for calculating the serialized size. :rtype: std::function< uint32_t () > :return: Functor which calculates the serialized size of the data.

is_bounded()
Checks if the type is bounded.

is_plain(**args*)
Overload 1:

Checks if the type is plain when using default encoding.

Overload 2:

Checks if the type is plain when using a specific encoding.

property m_isGetKeyDefined
Indicates whether the method to obtain the key has been implemented.

property m_typeSize
Maximum serialized size of the type in bytes. If the type has unbounded fields, and therefore cannot have a maximum size, use 0.

serialize(*args)
Overload 1:

Serialize method, it should be implemented by the user, since it is abstract. It is VERY IMPORTANT that the user sets the SerializedPayload length correctly.

data Pointer to the data payload Pointer to the payload :rtype: boolean :return: True if correct.

Overload 2:

Serialize method, it should be implemented by the user, since it is abstract. If not implemented, this method will call the serialize method in which the topic data representation is not considered. It is VERY IMPORTANT that the user sets the SerializedPayload length correctly.

data Pointer to the data payload Pointer to the payload data_representation Representation that should be used to encode the data into the payload. :rtype: boolean :return: True if correct.

setName(*nam*)
Set topic data type name

Parameters **nam** (*string*) – Topic data type name

property thisown
The membership flag

type_identifier(*args)
Overload 1:

Get the type identifier

Return type std::shared_ptr< eprosima::fastdds::dds::TypeIdV1 >

Returns TypeIdV1

Overload 2:

Set type identifier

Parameters **id** (*TypeIdV1*) – new value for TypeIdV1

Overload 3:

Set type identifier

Parameters **id** (*std::shared_ptr< eprosima::fastdds::dds::TypeIdV1 >*) – shared pointer to TypeIdV1

type_information(**args*)

Overload 1:

Get the type information

Return type std::shared_ptr< eprosima::fastdds::dds::xtypes::TypeInformation >

Returns TypeInformation

Overload 2:

Set type information

Parameters **info** (*TypeInformation*) – new value for TypeInformation

Overload 3:

Set type information

Parameters **info**(*std::shared_ptr< eprosima::fastdds::dds::xtypes::TypeInformation >*) – shared pointer to TypeInformation

type_object(**args*)

Overload 1:

Get the type object

Return type std::shared_ptr< eprosima::fastdds::dds::TypeObjectV1 >

Returns TypeObjectV1

Overload 2:

Set type object

Parameters **object** (*TypeObjectV1*) – new value for TypeObjectV1

Overload 3:

Set type object

Parameters **object** (*std::shared_ptr< eprosima::fastdds::dds::TypeObjectV1 >*) – shared pointer to TypeObjectV1

TopicDescription

class fastdds.**TopicDescription**(**args*, ***kwargs*)

Class TopicDescription, represents the fact that both publications and subscriptions are tied to a single data-type

get_impl()

get_name()

Get the name used to create this TopicDescription.

Return type string

Returns the name used to create this TopicDescription.

get_participant()

Get the DomainParticipant to which the TopicDescription belongs.

Return type *DomainParticipant*

Returns The DomainParticipant to which the TopicDescription belongs.

get_type_name()

Get the associated type name.

Return type string

Returns the type name.

property thisown

The membership flag

TopicListener

class fastdds.**TopicListener**

Class TopicListener, it should be used by the end user to implement specific callbacks to certain actions.

on_inconsistent_topic(*topic*, *status*)

Virtual function to be implemented by the user containing the actions to be performed when another topic exists with the same name but different characteristics.

Parameters

- **topic** (*Topic*) – Topic
- **status** (InconsistentTopicStatus) – The inconsistent topic status

property thisown

The membership flag

TopicQos

class fastdds.**TopicQos**

Class TopicQos, containing all the possible Qos that can be set for a determined Topic. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

deadline(**args*)

Overload 1:

Getter for DeadlineQosPolicy

Return type *DeadlineQosPolicy*

Returns DeadlineQos reference

Overload 2:

Getter for DeadlineQosPolicy

Return type *DeadlineQosPolicy*

Returns DeadlineQos reference

Overload 3:

Setter for DeadlineQosPolicy

Parameters **deadline** (*DeadlineQosPolicy*) – new value for the DeadlineQosPolicy

destination_order(**args*)

Overload 1:

Getter for DestinationOrderQosPolicy

Return type *DestinationOrderQosPolicy*

Returns DestinationOrderQos reference

Overload 2:

Getter for DestinationOrderQosPolicy

Return type *DestinationOrderQosPolicy*

Returns DestinationOrderQos reference

Overload 3:

Setter for DestinationOrderQosPolicy

Parameters **destination_order** (*DestinationOrderQosPolicy*) – new value for the DestinationOrderQosPolicy

durability(**args*)

Overload 1:

Getter for DurabilityQosPolicy

Return type *DurabilityQosPolicy*

Returns DurabilityQos reference

Overload 2:

Getter for DurabilityQosPolicy

Return type *DurabilityQosPolicy*

Returns DurabilityQos reference

Overload 3:

Setter for DurabilityQosPolicy

Parameters **durability** (*DurabilityQosPolicy*) – new value for the DurabilityQosPolicy

durability_service(**args*)

Overload 1:

Getter for DurabilityServiceQosPolicy

Return type *DurabilityServiceQosPolicy*

Returns DurabilityServiceQos reference

Overload 2:

Getter for DurabilityServiceQosPolicy

Return type *DurabilityServiceQosPolicy*

Returns DurabilityServiceQos reference

Overload 3:

Setter for DurabilityServiceQosPolicy

Parameters **durability_service** (*DurabilityServiceQosPolicy*) – new value for the DurabilityServiceQosPolicy

history(**args*)

Overload 1:

Getter for HistoryQosPolicy

Return type *HistoryQosPolicy*

Returns HistoryQos reference

Overload 2:

Getter for HistoryQosPolicy

Return type *HistoryQosPolicy*

Returns HistoryQos reference

Overload 3:

Setter for HistoryQosPolicy

Parameters **history** (*HistoryQosPolicy*) – new value for the HistoryQosPolicy

latency_budget(**args*)

Overload 1:

Getter for LatencyBudgetQosPolicy

Return type *LatencyBudgetQosPolicy*

Returns LatencyBudgetQos reference

Overload 2:

Getter for LatencyBudgetQosPolicy

Return type *LatencyBudgetQosPolicy*

Returns LatencyBudgetQos reference

Overload 3:

Setter for LatencyBudgetQosPolicy

Parameters **latency_budget** (*LatencyBudgetQosPolicy*) – new value for the LatencyBudgetQosPolicy

lifespan(**args*)

Overload 1:

Getter for LifespanQosPolicy

Return type *LifespanQosPolicy*

Returns LifespanQos reference

Overload 2:

Getter for LifespanQosPolicy

Return type *LifespanQosPolicy*

Returns LifespanQos reference

Overload 3:

Setter for LifespanQosPolicy

Parameters **lifespan** (*LifespanQosPolicy*) – new value for the LifespanQosPolicy

liveliness(**args*)

Overload 1:

Getter for LivelinessQosPolicy

Return type *LivelinessQosPolicy*

Returns LivelinessQos reference

Overload 2:

Getter for LivelinessQosPolicy

Return type *LivelinessQosPolicy*

Returns LivelinessQos reference

Overload 3:

Setter for LivelinessQosPolicy

Parameters **liveliness** (*LivelinessQosPolicy*) – new value for the LivelinessQosPolicy

ownership(**args*)

Overload 1:

Getter for OwnershipQosPolicy

Return type *OwnershipQosPolicy*

Returns OwnershipQos reference

Overload 2:

Getter for OwnershipQosPolicy

Return type *OwnershipQosPolicy*

Returns OwnershipQos reference

Overload 3:

Setter for OwnershipQosPolicy

Parameters **ownership** (*OwnershipQosPolicy*) – new value for the OwnershipQosPolicy

reliability(**args*)

Overload 1:

Getter for ReliabilityQosPolicy

Return type *ReliabilityQosPolicy*

Returns ReliabilityQos reference

Overload 2:

Getter for ReliabilityQosPolicy

Return type *ReliabilityQosPolicy*

Returns ReliabilityQos reference

Overload 3:

Setter for ReliabilityQosPolicy

Parameters **reliability** (*ReliabilityQosPolicy*) – new value for the ReliabilityQosPolicy

resource_limits(**args*)

Overload 1:

Getter for ResourceLimitsQosPolicy

Return type *ResourceLimitsQosPolicy*

Returns ResourceLimitsQos reference

Overload 2:

Getter for ResourceLimitsQosPolicy

Return type *ResourceLimitsQosPolicy*

Returns ResourceLimitsQos reference

Overload 3:

Setter for ResourceLimitsQosPolicy

Parameters **resource_limits** (*ResourceLimitsQosPolicy*) – new value for the ResourceLimitsQosPolicy

property thisown

The membership flag

topic_data(**args*)

Overload 1:

Getter for TopicDataQosPolicy

Return type *TopicDataQosPolicy*

Returns TopicDataQos reference

Overload 2:

Getter for TopicDataQosPolicy

Return type *TopicDataQosPolicy*

Returns TopicDataQos reference

Overload 3:

Setter for TopicDataQosPolicy

Parameters **value** (*TopicDataQosPolicy*) – new value for the TopicDataQosPolicy

transport_priority(**args*)

Overload 1:

Getter for TransportPriorityQosPolicy

Return type *TransportPriorityQosPolicy*

Returns TransportPriorityQos reference

Overload 2:

Getter for TransportPriorityQosPolicy

Return type *TransportPriorityQosPolicy*

Returns TransportPriorityQos reference

Overload 3:

Setter for TransportPriorityQosPolicy

Parameters **transport_priority** (*TransportPriorityQosPolicy*) – new value for the TransportPriorityQosPolicy

class fastdds.**TOPIC_QOS_DEFAULT**(**args: Any*, ***kwargs: Any*)

TypeIdV1

class fastdds.**TypeIdV1**(**args*)

Class TypeIdV1

clear()

Clears the QosPolicy object

get()

Getter for the TypeIndentifier

Return type eprosima::fastrtps::types::TypeIdentifier

Returns TypeIdentifier reference

property m_type_identifier

Type Identifier

property thisown
The membership flag

TypeInformation

class fastdds.TypeInformation(*args)
Class xtypes::TypeInformation

assigned(*args)
Overload 1:

Check if it is assigned

Return type boolean

Returns true if assigned, false if not

Overload 2:

Setter for assigned boolean

Parameters **value** (*boolean*) – Boolean to be set

clear()
Clears the QosPolicy object

property thisown
The membership flag

property type_information
Type Information

TypeObjectV1

class fastdds.TypeObjectV1(*args)
Class TypeObjectV1

clear()
Clears the QosPolicy object

get()
Getter for the TypeObject

Return type eprosima::fastrtps::types::TypeObject

Returns TypeObject reference

property m_type_object
Type Object

property thisown
The membership flag

TypeSupport

class fastdds.**TypeSupport**(**args*)

Notes: This class inherits from std::shared_ptr<TopicDataType>. Class TypeSupport used to provide the DomainRTPSParticipant with the methods to serialize, deserialize and get the key of a specific data type. The user should created a class that inherits from this one, where Serialize and deserialize methods MUST be implemented.

create_data()

Creates new data

Return type void

Returns Pointer to the data

delete_data(*data*)

Deletes data

Parameters **data** (*void*) – Pointer to the data to delete

deserialize(*payload*, *data*)

Deserializes the data

Parameters

- **payload** (*eprosima::fastrtps::rtps::SerializedPayload_t*) – Pointer to payload
- **data** (*void*) – Pointer to data

Return type boolean

Returns true if it is deserialized correctly, false if not

empty()

Check if the TypeSupport is empty

Return type boolean

Returns true if empty, false if not

get_key(*data*, *i_handle*, *force_md5=False*)

Getter for the data key

Parameters

- **data** (*void*) – Pointer to data
- **i_handle** (InstanceHandle_t) – InstanceHandle pointer to store the key
- **force_md5** (*boolean*) – boolean to force md5 (default: false)

Return type boolean

Returns true if the key is returned, false if not

get_serialized_size_provider(**args*)

Overload 1:

Returns a function which can be used to calculate the serialized size of the provided data.

data Pointer to data. :rtype: std::function< uint32_t () > :return: Functor which calculates the serialized size of the data.

Overload 2:

Returns a function which can be used to calculate the serialized size of the provided data.

data Pointer to data. data_representation Representation that should be used for calculating the serialized size. :rtype: std::function< uint32_t () > :return: Functor which calculates the serialized size of the data.

get_type_name()
Getter for the type name

Return type string

Returns name of the data type

is_bounded()
Checks if the type is bounded.

is_plain(**args*)
Overload 1:

Checks if the type is plain when using default encoding.

Overload 2:

Checks if the type is plain when using a specific encoding.

register_type(**args*)
Overload 1:

Registers the type on a participant

Parameters **participant** (*DomainParticipant*) – DomainParticipant where the type is going to be registered

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the type name is empty, RETCODE_PRECONDITION_NOT_MET if there is another type with the same name registered on the DomainParticipant and RETCODE_OK if it is registered correctly

Overload 2:

Registers the type on a participant

Parameters

- **participant** (*DomainParticipant*) – DomainParticipant where the type is going to be registered
- **type_name** (*string*) – Name of the type to register

Return type ReturnCode_t

Returns RETCODE_BAD_PARAMETER if the type name is empty, RETCODE_PRECONDITION_NOT_MET if there is another type with the same name registered on the DomainParticipant and RETCODE_OK if it is registered correctly

serialize(**args*)

Overload 1:

Serializes the data

Parameters

- **data** (*void*) – Pointer to data
- **payload** (*eprosima::fastrtps::rtps::SerializedPayload_t*) – Pointer to payload

Return type boolean

Returns true if it is serialized correctly, false if not

Overload 2:

Serializes the data

Parameters

- **data** (*void*) – Pointer to data
- **payload** (*eprosima::fastrtps::rtps::SerializedPayload_t*) – Pointer to payload data_representation Representation that should be used to encode the data into the payload.

Return type boolean

Returns true if it is serialized correctly, false if not

set(*ptr*)

property thisown

The membership flag

6.34 Introduction

eProsima Fast DDS-Gen is a Java application that generates *eProsima Fast DDS* source code using the data types defined in an IDL (Interface Definition Language) file. This generated source code can be used in any *Fast DDS* application in order to define the data type of a topic, which will later be used to publish or subscribe. *eProsima Fast DDS* defines the data type exchanged in a Topic through two classes: the *TypeSupport* and the *TopicDataType*. *TopicDataType* describes the data type exchanged between a publication and a subscription, i.e. the data corresponding to a Topic; while *TypeSupport* encapsulates an instance of TopicDataType, providing the functions needed to register the type and interact with the publication and subscription. Please refer to *Definition of data types* for more information on data types.

To declare the structured data, the IDL format must be used. IDL is a specification language, made by OMG (Object Management Group), which describes an interface in a language independent manner, allowing communication between software components that do not share the same language. The *eProsima Fast DDS-Gen* tool reads the IDL files and parses a subset of the OMG IDL specification to generate source code for data serialization. This subset includes the data type descriptions included in *Defining a data type via IDL*. The rest of the file content is ignored.

eProsima Fast DDS-Gen generated source code uses Fast CDR, a C++11 library that provides the data serialization and codification mechanisms. Therefore, as stated in the RTPS standard, when the data are sent, they are serialized

and encoded using the corresponding Common Data Representation (CDR). The CDR transfer syntax is a low-level representation for inter-agents transfer, mapping from OMG IDL data types to byte streams. Please refer to the official CDR specification for more information on the CDR transfer syntax (see PDF section 15.3).

The main feature of *eProsima Fast DDS-Gen* is to facilitate the implementation of DDS applications without the knowledge of serialization or deserialization mechanisms. With *Fast DDS-Gen* it is also possible to generate the C++ source code of a DDS application with a publisher and a subscriber that uses the *eProsima Fast DDS* library (see *Building a publish/subscribe application*). *Fast DDS-Gen* can also generate Python bindings for the data types in order to use them within a Python-based *Fast DDS* application (see *Building Python auxiliary libraries*).

For installing *Fast DDS-Gen*, please refer to *Linux installation of Fast DDS-Gen* or to *Windows installation of Fast DDS-Gen*.

6.35 Usage

This section explains the usage of *Fast DDS-Gen* tool and briefly describes the generated files.

6.35.1 Running the *Fast DDS-Gen* Java application

First, the steps outlined in *Linux installation of Fast DDS-Gen* or *Window installation of Fast DDS-Gen* must be accomplished for the installation of *Fast DDS-Gen*. According to this section, an executable file for Linux and Windows that runs the Java *Fast DDS-Gen* application is available in the `scripts` folder. If the `scripts` folder path is added to the `PATH` environment variable, *Fast DDS-Gen* can be executed running the following commands:

- Linux:

```
$ fastddsgen
```

- Windows:

```
> fastddsgen.bat
```

Note: In case the PATH has not been modified, these scripts can be found in the `<fastddsgen_directory>/scripts` directory.

6.35.2 Supported options

The expected argument list of the application is:

```
fastddsgen [<options>] <IDL file> [<IDL file> ...]
```

Where the option choices are:

Option	Description
-cdr	Sets the Fast CDR version used to generate types source code. Values: - v1. Generate source code for Fast CDR v1. - v2 (default). Generate source code for Fast CDR v2. - both. Generate source code for both Fast CDR v1 and Fast CDR v2.
-cs	Enables Case Sensitivity
-d <directory>	Sets the output directory where the generated files are created.
-default_extensibility <extensibility> -de <extensibility>	Sets the default extensibility for types without the @extensibility annotation. Values: - final - appendable (default) - mutable
-example <platform>	Generates an example and a solution to compile the generated source code for a specific platform. The help command shows the supported platforms.
-extrastg <template> <output>	Specifies a custom template used for generating source code. This option expects the location of the template and the location of the file where source code output will be stored. A custom template example can be found in this link
-help	Shows the help information
-I <directory>	Add directory to preprocessor include paths.
-ppDisable	Disables the preprocessor.
-ppPath	Specifies the preprocessor path.
-python	Generates source code and a CMake solution to compile a library containing the data types Python bindings required to run a *Fast DDS* Python-based application. This option is incompatible with the *-example* and *-typeobject* ones.
-replace	Replaces the generated source code files even if they exist.
-t <directory>	Sets a specific directory as a temporary directory.
-typeobject	Generates *TypeObject* files for the IDL provided and modifies MyType constructor to register the TypeObject representation into the factory.
-typeros2	Generates type naming compatible with ROS 2
-version	Shows the current version of eProsima *Fast DDS-Gen*.

Please refer to *Dynamic Topic Types* for more information on TypeObject representation.

6.36 Building a publish/subscribe application

Fast DDS-Gen can be used to build a fully functional publication/subscription application from an IDL file that defines the Topic under which messages are published and received. The application generated allows for the creation of as many publishers and subscribers as desired, all belonging to the same Domain and communicating using the same Topic.

- *Background*
- *Prerequisites*
- *Create the application workspace*
- *Import linked libraries and its dependencies*
  - *Installation from binaries*
  - *Colcon installation*
- *Creating the IDL file with the data type*
- *Generating a minimal functional example*

  - *Generate the Fast DDS source code*
  - *Build the Fast DDS application*
  - *Run the Fast DDS application*
- *Summary and next steps*

6.36.1 Background

eProsima Fast DDS-Gen is a Java application that generates eProsima Fast DDS source code using the data types defined in an IDL (Interface Definition Language) file. This generated source code can be used in any Fast DDS application in order to define the data type of a topic, which will later be used to publish or subscribe. Please refer to *Fast DDS-Gen introduction* for more information.

6.36.2 Prerequisites

First of all, follow the steps outlined in the Installation Manual for the installation of *eProsima Fast DDS* and all its dependencies. Moreover, perform the steps outlined in *Linux installation of Fast DDS-Gen* or in *Window installation of Fast DDS-Gen*, depending on the operating system, for the installation of the *eProsima Fast DDS-Gen* tool.

6.36.3 Create the application workspace

The application workspace will have the following structure at the end of the project. The file `build/HelloWorld` is the generated *Fast DDS* application.

```
.
└── FastDDSGenHelloWorld
    ├── build
    │   ├── CMakeCache.txt
    │   ├── CMakeFiles
    │   ├── cmake_install.cmake
    │   ├── HelloWorld
    │   ├── libHelloWorld_lib.a
    │   └── Makefile
    ├── CMakeLists.txt
    ├── HelloWorld.cxx
    ├── HelloWorld.h
    ├── HelloWorld.idl
    ├── HelloWorldCdrAux.hpp
    ├── HelloWorldCdrAux.ipp
    ├── HelloWorldPublisher.cxx
    ├── HelloWorldPublisher.h
    ├── HelloWorldPubSubMain.cxx
    ├── HelloWorldPubSubTypes.cxx
    ├── HelloWorldPubSubTypes.h
    ├── HelloWorldSubscriber.cxx
    └── HelloWorldSubscriber.h
```

Execute the following command to create the directory in which the files generated by *Fast DDS-Gen* will be saved.

```
mkdir FastDDSGenHelloWorld && cd FastDDSGenHelloWorld
mkdir build
```

6.36.4 Import linked libraries and its dependencies

The DDS application requires the *Fast DDS* and *Fast CDR* libraries. The way of making these accessible from the workspace depends on the installation procedure followed in the Installation Manual.

Installation from binaries

If the installation from binaries has been followed, these libraries are already accessible from the workspace.

- On Linux: The header files can be found in directories `/usr/include/fastrtps/` and `/usr/include/fastcdr/` for *Fast DDS* and *Fast CDR* respectively. The compiled libraries of both can be found in the directory `/usr/lib/`.
- On Windows: The header files can be found in directories `C:\Program Files\eProsima\fastrtps 2.0.0\include\fastrtps` and `C:\Program Files\eProsima\fastrtps 2.0.0\include\fastcdr\` for *Fast DDS* and *Fast CDR* respectively. The compiled libraries of both can be found in the directory `C:\Program Files\eProsima\fastrtps 2.0.0\lib\`.

Colcon installation

If the Colcon installation has been followed, there are several ways to import the libraries. To make these accessible only from the current shell session, run one of the following two commands.

- On Linux:

```
source <path/to/Fast-DDS/workspace>/install/setup.bash
```

- On Windows:

```
<path/to/Fast-DDS/workspace>/install/setup.bat
```

However, to make these accessible from any session, add the *Fast DDS* installation directory to the `$PATH` variable in the shell configuration files running the following command.

- On Linux:

```
echo 'source <path/to/Fast-DDS/workspace>/install/setup.bash' >> ~/.bashrc
```

- On Windows: Open the *Edit the system environment variables* control panel and add `<path/to/Fast-DDS/workspace>/install/setup.bat` to the `PATH`.

6.36.5 Creating the IDL file with the data type

To build a minimal application, the Topic must be defined by means of an IDL file. For this example the Topic data type defined by IDL is just a `string` message. Topics are explained in more detail in *Topic*, while the Topic data types to be defined using IDL are presented in *Definition of data types*. In the preferred text editor, create the *HelloWorld.idl* file with the following content and save it in the *FastDDSGenHelloWorld* directory.

```
// HelloWorld.idl
struct HelloWorld
{
    string message;
};
```

Then, this file is translated to something *Fast DDS* understands. For this, use the *Fast DDS-Gen* code generation tool, which can do two different things:

1. Generate C++ definitions for a custom topic.
2. Generate a functional example that uses the topic data.

The second option is the one used to create this publish/subscribe application, while the first option is applied in this other tutorial: *Writing a simple C++ publisher and subscriber application*.

6.36.6 Generating a minimal functional example

If the steps outlined in the Installation Manual have been followed, then *Fast DDS*, *Fast CDR*, and Fast-RTPS-Gen should be installed in the system.

Generate the Fast DDS source code

The application files are generated using the following command. The `-example` option creates an example application, and the CMake files needed to build it. In the workspace directory (*FastDDSGenHelloWorld* directory), execute one of the following commands according to the installation followed and the operating system.

- On Linux:
  - For an **installation from binaries** or a **colcon installation**:

```
<path-to-Fast-DDS-workspace>/src/fastddsgen/scripts/fastddsgen -example␣
↪CMake HelloWorld.idl
```

  - For a **stand-alone installation**, run:

```
<path-to-Fast-DDS-Gen>/scripts/fastddsgen -example CMake HelloWorld.idl
```

- On Windows:
  - For a **colcon installation**:

```
<path-to-Fast-DDS-workspace>/src/fastddsgen/scripts/fastddsgen.bat -
↪example CMake HelloWorld.idl
```

  - For a **stand-alone installation**, run:

```
<path-to-Fast-DDS-Gen>/scripts/fastddsgen.bat -example CMake HelloWorld.idl
```

– For an **installation from binaries**, run:

```
fastddsgen.bat -example CMake HelloWorld.idl
```

Warning: The colcon installation does not build the `fastddsgen.jar` file although it does download the Fast DDS-Gen repository. The following commands must be executed to build the Java executable:

```
cd <path-to-Fast-DDS-workspace>/src/fastddsgen
gradle assemble
```

Build the Fast DDS application

Then, compile the generated code executing the following commands from the *FastDDSGenHelloWorld* directory.

- On Linux:

```
cd build
cmake ..
make
```

- On Windows:

```
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
cmake --build .
```

Run the Fast DDS application

The application build can be used to spawn any number of publishers and subscribers associated with the topic.

- On Linux:

```
./HelloWorld publisher
./HelloWorld subscriber
```

- On Windows:

```
HelloWorld.exe publisher
HelloWorld.exe subscriber
```

Each time <Enter> is pressed on the Publisher, a new datagram is generated, sent over the network and received by Subscribers currently online. If more than one subscriber is available, it can be seen that the message is equally received on all listening nodes.

The values on the custom IDL-generated data type can also be modified as indicated below.

```
HelloWorld sample; //Auto-generated container class for topic data from Fast DDS-Gen
sample.msg("Hello there!"); // Add contents to the message
publisher->write(&sample); //Publish
```

Warning: It may be necessary to set up a special rule in the Firewall for *eprosima Fast DDS* to work correctly on Windows.

6.36.7 Summary and next steps

In this tutorial, a publisher/subscriber DDS application using *Fast DDS-Gen* has been built. The tutorial also describes how to generate IDL files that contain the description of the Topic data type.

To continue developing DDS applications please take a look at the eProsima Fast DDS examples on github for ideas on how to improve this basic application through different configuration options, and also for examples of advanced *Fast DDS* features.

6.37 Building Python auxiliary libraries

eProsima Fast DDS-Gen can generate the required source files and CMake project to build the Python modules that allow the use of the IDL defined data types within a *Fast DDS* Python-based application. Each IDL file will result in a new Python module that will contain all the data types defined in the file. The Python binding is generated building the provided solution using SWIG.

Calling *eProsima Fast DDS-Gen* with the option *-python* will generate these files. *eProsima Fast DDS-Gen* will generate a *.i* file which will be processed by SWIG and a CMake project to call SWIG first generating C++ files (for connecting C++ and Python) and Python files (Python module for your type) and then compiling the C++ sources.

Before calling CMake, the *Building* process needs several *Dependencies* to be met.

6.37.1 Dependencies

SWIG

SWIG is a development tool that allows connecting programs written in C/C++ with a variety of other programming languages, among them Python. SWIG 4.0 is required in order to build Fast DDS Python bindings.

Note: More recent SWIG releases are not yet supported. Please, ensure to be using SWIG 4.0.

SWIG can be installed directly from the package manager of the appropriate Linux distribution. For Ubuntu, please run:

```
sudo apt install swig
```

Header files and static library for Python

Python static libraries and header files are needed to compile C++ source code generated by SWIG. They can be installed directly from the package manager of the appropriate Linux distribution. For Ubuntu, please run:

```
sudo apt install libpython3-dev
```

6.37.2 Building

Call CMake:

```
mkdir build
cd build
cmake ..
cmake --build .
```

This will create the Python files (*.py*) with the modules (one per each IDL file) that have to be imported within the Python script.

6.38 Defining a data type via IDL

This section describes the data types that can be defined using IDL files, as well as other mechanisms for building data types using IDL files.

- *Supported IDL types*
  - *Primitive types*
  - *Arrays*
  - *Sequences*
  - *Maps*
  - *Structures*
  - *Unions*
  - *Bitsets*
  - *Enumerations*
  - *Bitmasks*
  - *Modules*
  - *Data types with a key*
- *Including other IDL files*
- *Annotations*
- *Forward declaration*
- *IDL 4.2 aliases*
- *IDL 4.2 comments*

6.38.1 Supported IDL types

Be aware that *Fast DDS-Gen* is not case sensitive as it is specified in the IDL specification. To activate case sensitivity use option `-cs` when running *Fast DDS-Gen* (see *Supported options*).

Primitive types

The following table shows the basic IDL types supported by *Fast DDS-Gen* and how they are mapped to C++11.

IDL	C++11
char	`char`
octet	`uint8_t`
short	`int16_t`
unsigned short	`uint16_t`
long	`int32_t`
unsigned long	`uint32_t`
long long	`int64_t`
unsigned long long	`uint64_t`
float	`float`
double	`double`
long double	`long double`
boolean	`bool`
string	`std::string`

Arrays

Fast DDS-Gen supports unidimensional and multidimensional arrays. Arrays are always mapped to `std::array` containers. The following table shows the array types supported and their mapping.

IDL	C++11
char a[5]	`std::array<char,5> a`
octet a[5]	`std::array<uint8_t,5> a`
short a[5]	`std::array<int16_t,5> a`
unsigned short a[5]	`std::array<uint16_t,5> a`
long a[5]	`std::array<int32_t,5> a`
unsigned long a[5]	`std::array<uint32_t,5> a`
long long a[5]	`std::array<int64_t,5> a`
unsigned long long a[5]	`std::array<uint64_t,5> a`
float a[5]	`std::array<float,5> a`
double a[5]	`std::array<double,5> a`

Sequences

Fast DDS-Gen supports sequences, which map into the `std::vector` container. The following table represents how the map between IDL and C++11 is handled.

IDL	C++11
sequence<char>	`std::vector<char>`
sequence<octet>	`std::vector<uint8_t>`
sequence<short>	`std::vector<int16_t>`
sequence<unsigned short>	`std::vector<uint16_t>`
sequence<long>	`std::vector<int32_t>`
sequence<unsigned long>	`std::vector<uint32_t>`
sequence<long long>	`std::vector<int64_t>`
sequence<unsigned long long>	`std::vector<uint64_t>`
sequence<float>	`std::vector<float>`
sequence<double>	`std::vector<double>`

Maps

Fast DDS-Gen supports maps, which are equivalent to the `std::map` container. The equivalence between types is handled in the same way as for *sequences*.

IDL	C++11
map<char, unsigned long long>	`std::map<char, uint64_T>`

Note: Only *Primitive types* are currently supported.

Structures

You can define an IDL structure with a set of members with multiple types. It will be converted into a C++ class in which the members of the structure defined via IDL are mapped to private data members of the class. Furthermore, `set()` and `get()` member functions are created to access these private data members.

The following IDL structure:

```
struct Structure
{
    octet octet_value;
    long long_value;
    string string_value;
};
```

Would be converted to:

```
class Structure
{
public:

    Structure();
```

(continues on next page)

(continued from previous page)

```
    ~Structure();
    Structure(
            const Structure& x);
    Structure(
            Structure&& x);
    Structure& operator =(
            const Structure& x);
    Structure& operator =(
            Structure&& x);

    void octet_value(
            uint8_t _octet_value);
    uint8_t octet_value() const;
    uint8_t& octet_value();
    void long_value(
            int64_t _long_value);
    int64_t long_value() const;
    int64_t& long_value();
    void string_value(
            const std::string
            & _string_value);
    void string_value(
            std::string&& _string_value);
    const std::string& string_value() const;
    std::string& string_value();

private:

    uint8_t m_octet_value;
    int64_t m_long_value;
    std::string m_string_value;
};
```

Structures can inherit from other structures, extending their member set.

```
struct ParentStruct
{
    octet parent_member;
};

struct ChildStruct : ParentStruct
{
    long child_member;
};
```

In this case, the resulting C++ code will be:

```
class ParentStruct
{
    octet parent_member;
};
```

(continues on next page)

(continued from previous page)

```
class ChildStruct : public ParentStruct
{
    long child_member;
};
```

Optional members

A member of a structure can be optional. This is achieved by writing the `@optional` annotation before the member.

```
struct StructWithOptionalMember
{
    @optional octet octet_opt;
};
```

An optional member is converted into a template class `eprosima::fastcdr::optional<T>`, where `T` is the member's type.

```
class StructWithOptionalMember
{
    eprosima::fastcdr::optional<octet> octet_opt;
};
```

Before reading the value of the optional member, it should be checked the optional contains a value using `has_value()` function. Accessing a *null* optional throws a `eprosima::fastcdr::exception::BadOptionalAccessException` exception.

```
if (octet_opt.has_value())
{
    octet oc = octet_opt.value();
}
```

Extensibility

In order to support evolving types without breaking interoperability, the concept of type extensibility is supported by *Fast DDS-Gen*. There are three extensibility kinds: *final*, *appendable* and *mutable*.

- *FINAL* extensibility indicates that the type is strictly defined. It is not possible to add members while maintaining type assignability.
- *APPENDABLE* extensibility indicates that two types, where one contains all of the members of the other plus additional members appended to the end, may remain assignable.
- *MUTABLE* extensibility indicates that two types may differ from one another in the additional, removal, and/or transposition of members while remaining assignable.

```
@extensibility(FINAL)
struct FinalStruct
{
    octet octet_opt;
};
```

(continues on next page)

(continued from previous page)

```
@extensibility(APPENDABLE)
struct AppendableStruct
{
    octet octet_opt;
};

@extensibility(MUTABLE)
struct MutableStruct
{
    octet octet_opt;
};
```

Note: XCDRv1 encoding algorithm is not able to manage correctly the deserialization of an appendable structure when it is used as a member of another one.

Unions

In IDL, a union is defined as a sequence of members with their own types and a discriminant that specifies which member is in use. An IDL union type is mapped as a C++ class with member functions to access the union members and the discriminant.

The following IDL union:

```
union Union switch(long)
{
   case 1:
    octet octet_value;
  case 2:
    long long_value;
  case 3:
    string string_value;
};
```

Would be converted to:

```
class Union
{
public:

    Union();
    ~Union();
    Union(
            const Union& x);
    Union(
            Union&& x);
    Union& operator =(
            const Union& x);
    Union& operator =(
            Union&& x);
```

(continues on next page)

(continued from previous page)

```
    void d(
            int32_t __d);
    int32_t _d() const;
    int32_t& _d();

    void octet_value(
            uint8_t _octet_value);
    uint8_t octet_value() const;
    uint8_t& octet_value();
    void long_value(
            int64_t _long_value);
    int64_t long_value() const;
    int64_t& long_value();
    void string_value(
            const std::string
            & _string_value);
    void string_value(
            std:: string&& _string_value);
    const std::string& string_value() const;
    std::string& string_value();

private:

    int32_t m__d;
    uint8_t m_octet_value;
    int64_t m_long_value;
    std::string m_string_value;
};
```

Bitsets

Bitsets are a special kind of structure, which encloses a set of bits. A bitset can represent up to 64 bits. Each member is defined as *bitfield* and eases the access to a part of the bitset.

For example:

```
bitset MyBitset
{
    bitfield<3> a;
    bitfield<10> b;
    bitfield<12, long> c;
};
```

The type `MyBitset` will store a total of 25 bits (3 + 10 + 12) and will require 32 bits in memory (lowest primitive type to store the bitset's size).

- The bitfield 'a' allows us to access to the first 3 bits (0..2).
- The bitfield 'b' allows us to access to the next 10 bits (3..12).
- The bitfield 'c' allows us to access to the next 12 bits (13..24).

The resulting C++ code will be similar to:

```
class MyBitset
{
public:

    void a(
            char _a);
    char a() const;

    void b(
            uint16_t _b);
    uint16_t b() const;

    void c(
            int32_t _c);
    int32_t c() const;

private:

    std::bitset<25> m_bitset;
};
```

Internally, it is stored as a `std::bitset`. For each bitfield, `get()` and `set()` member functions are generated with the smaller possible primitive unsigned type to access it. In the case of bitfield 'c', the user has established that this accessing type will be `long`, so the generated code uses `int32_t` instead of automatically use `uint16_t`.

Bitsets can inherit from other bitsets, extending their member set.

```
bitset ParentBitset
{
    bitfield<3> parent_member;
};

bitset ChildBitset : ParentBitset
{
    bitfield<10> child_member;
};
```

In this case, the resulting C++ code will be:

```
class ParentBitset
{
    std::bitset<3> parent_member;
};

class ChildBitset : public ParentBitset
{
    std::bitset<10> child_member;
};
```

Note that in this case, `ChildBitset` will have two `std::bitset` data members, one belonging to `ParentBitset` and the other belonging to `ChildBitset`.

Enumerations

An enumeration in IDL format is a collection of identifiers that have an associated numeric value. An IDL enumeration type is mapped directly to the corresponding C++11 enumeration definition.

The following IDL enumeration:

```
enum Enumeration
{
    RED,
    GREEN,
    BLUE
};
```

Would be converted to:

```
enum Enumeration : uint32_t
{
    RED,
    GREEN,
    BLUE
};
```

Bitmasks

Bitmasks are a special kind of Enumeration to manage masks of bits. It allows defining bit masks based on their position.

The following IDL bitmask:

```
@bit_bound(8)
bitmask MyBitMask
{
    @position(0) flag0,
    @position(1) flag1,
    @position(4) flag4,
    @position(6) flag6,
    flag7
};
```

Would be converted to:

```
enum MyBitMask : uint8_t
{
    flag0 = 0x01 << 0,
    flag1 = 0x01 << 1,
    flag4 = 0x01 << 4,
    flag6 = 0x01 << 6,
    flag7 = 0x01 << 7
};
```

The annotation `bit_bound` defines the width of the associated enumeration. It must be a positive number between 1 and 64. If omitted, it will be 32 bits. For each `flag`, the user can use the annotation `position` to define the position of the flag. If omitted, it will be auto incremented from the last defined flag, starting at 0.

Modules

In order to avoid collision between variable names, modules can be defined within the IDL file. A module would be converted into a namespace in C++.

Data types with a key

In order to use keyed topics, the user should define some key members inside the structure. This is achieved by writing the `@key` annotation before the members of the structure that are used as keys. For example in the following IDL file the *id* and *type* field would be the keys:

```
struct MyType
{
    @key long id;
    @key string type;
    long positionX;
    long positionY;
};
```

Fast DDS-Gen automatically detects these tags and correctly generates the serialization methods for the key generation function in TopicDataType (`getKey()`). This function will obtain the 128-bit MD5 digest of the big-endian serialization of the Key Members.

6.38.2 Including other IDL files

Other IDL files can be included in addition to the current IDL file. *Fast DDS-Gen* uses a C/C++ preprocessor for this purpose, and `#include` directive can be used to include an IDL file. Preprocessor directives guarding against multiple inclusion of the same IDL file are also advisable.

```
#include "OtherFile.idl"
#include <AnotherFile.idl>
```

If *Fast DDS-Gen* does not find a C/C++ preprocessor in default system paths, the preprocessor path can be specified using parameter `-ppPath`. The parameter `-ppDisable` can be used to disable the usage of the C/C++ preprocessor.

6.38.3 Annotations

The application allows the user to define and use their own annotations as defined in the OMG IDL 4.2 specification. User annotations will be passed to TypeObject generated code if the `-typeobject` argument was used.

```
@annotation MyAnnotation
{
    long value;
    string name;
};
```

Additionally, the following standard annotations are builtin (recognized and passed to TypeObject when unimplemented).

Annotation	Implemented behavior
@id	[Unimplemented] Assign a 32-bit integer identifier to an element.
@autoid	[Unimplemented] Automatically allocate identifiers to the elements.
@optional	Setting an element as optional. More info in *Optional Members*.
@extensibility	Applied to any element which is constructed. Allow specifying how the element is allowed to evolve. More info in *Extensibility*.
@final	Shortcut for *@extensibility(FINAL)*
@appendable	Shortcut for *@extensibility(APPENDABLE)*
@mutable	Shortcut for *@extensibility(MUTABLE)*
@position	Setting a position to an element or group of elements. Used by *bitmasks*.
@value	[Unimplemented] Allow setting a constant value to any element.
@key	Alias for eProsima's @Key annotation. Indicate that a data member is part of the key (please refer to *Topics, keys and instances* for more information).
@must_understand	[Unimplemented] Indicate that the data member must be understood by any application making use of that piece of data.
@default_literal	[Unimplemented] Allow selecting one member as the default within a collection.
@default	Allow specifying the default value of the annotated element.
@range	[Unimplemented] Allow specifying a range of allowed values for the annotated element.
@min	[Unimplemented] Allow specifying a minimum value for the annotated element.
@max	[Unimplemented] Allow specifying a maximum value for the annotated element.
@unit	[Unimplemented] Allow specifying a unit of measurement for the annotated element.
@bit_bound	Allow setting a size to a *bitmasks*.
@external	[Unimplemented] Force the annotated element to be placed in a dedicated data space.
@nested	[Unimplemented] Allow indicating that the objects from the type under annotation will always be nested within another one.
@verbatim	[Unimplemented] Allow injecting some user-provided information into what the compiler will generate.
@service	[Unimplemented] Allow indicating that an interface is to be treated as a service.
@oneway	[Unimplemented] Allow indicating that an operation is one way only, meaning that related information flow will go from client to server but not back.
@ami	[Unimplemented] Allow indicating that an interface or an operation is to be made callable asynchronously.
@non_serialized	The annotated member will be omitted from serialization.

Most unimplemented annotations are related to Extended Types.

6.38.4 Forward declaration

Fast DDS-Gen supports forward declarations. This allows declaring inter-dependant structures, unions, etc.

```
struct ForwardStruct;

union ForwardUnion;

struct ForwardStruct
{
    ForwardUnion fw_union;
};
```

(continues on next page)

(continued from previous page)

```
union ForwardUnion switch (long)
{
    case 0:
        ForwardStruct fw_struct;
    default:
        string empty;
};
```

6.38.5 IDL 4.2 aliases

IDL 4.2 allows using the following names for primitive types:

int8
uint8
int16
uint16
int32
uint32
int64
uint64

6.38.6 IDL 4.2 comments

There are two ways to write IDL comments:

- The characters `/*` start a comment, which terminates with the characters `*/`.
- The characters `//` start a comment, which terminates at the end of the line on which they occur.

Please refer to the IDL 4.2 specification (*Section 7.2 Lexical Conventions*) for more information on IDL conventions.

```
/* MyStruct definition */
struct MyStruc
{
    string mymessage;   // mymessage data member.
};
```

6.39 CLI

The *Fast DDS* command line interface provides a set commands and sub-commands to perform, *Fast DDS* related, maintenance and configuration tasks.

- *discovery*
  - *How to run*
  - *Examples*
- *shm*

- *xml*

An executable file for Linux and Windows that runs the *Fast DDS CLI* application is available in the *tools* folder. If the *tools/fastdds* folder path is added to the `PATH`, or by sourcing the *<path/to/fastdds>/install/setup.bash* configuration file, *Fast DDS CLI* can be executed running the following commands:

- Linux:

```
$ fastdds <command> [<command-args>]
```

- Windows:

```
> fastdds.bat <command> [<command-args>]
```

There are three verbs whose functionality is described in the following table:

Verbs	Description
`discovery`	Launches a server for *Discovery Server*.
`shm`	Allows manual cleaning of garbage files that may be generated by *Shared Memory Transport*
`xml`	Checks if a xml profile is well formed.

6.39.1 discovery

This command launches a *SERVER* (or *BACKUP*) for *Discovery Server*. This *server* will manage the discovery phases of the *CLIENTS* which are connected to it. *Clients* must know how to reach the *server*, which is accomplished by specifying an IP address, the *servers* GUID prefix, and a transport protocol like UDP or TCP. *Servers* do not need any prior knowledge of their *clients*, but require a GUID prefix, and the listening IP address where they may be reached. For more information on the different *Fast DDS* discovery mechanisms and how to configure them, please refer to *Discovery*.

Important: It is possible to interconnect *servers* (or *backup* servers) instantiated with `fastdds discovery` using environment variable `ROS_DISCOVERY_SERVER` (see *ROS_DISCOVERY_SERVER*) or a XML configuration file.

How to run

On a shell, execute:

```
fastdds discovery -i {0-255} [optional parameters]
```

Where the parameters are:

Option	Description
`-h` `-help`	Produce help message.
`-i` `--server-id`	Unique server identifier. Specifies zero based server position in `ROS_DISCOVERY_SERVER` environment variable. Must be an integer in range [0, 255] If not specified, it must be defined using a XML configuration file.
`-l` `--udp-address`	IPv4/IPv6 address chosen to listen the clients using UDP transport. Defaults to any (0.0.0.0/::0). Instead of an address, a DNS domain name can be specified.
`-p` `--udp-port`	UDP port chosen to listen the clients. Defaults to '11811'.
`-t` `--tcp-address`	IPv4/IPv6 address chosen to listen the clients using TCP transport. Instead of an address, a DNS domain name can be specified. Defaults to localhost (127.0.0.1).
`-q` `--tcp-port`	TCP port chosen to listen the clients. Defaults to '42100'. Only one server can be configured using the default port.
`-b` `--backup`	Creates a BACKUP *server* (see *Discovery Protocol*)
`-x` `--xml-file`	XML configuration file (see *XML profiles*). In this case, the default configuration file is not loaded. The CLI options override XML configuration for that specific parameter. The default profile in the XML file is loaded except if a specific profile name is specified: `profile_name@xml_file`

The output is:

```
### Server is running ###
  Participant Type:   <SERVER|BACKUP>
  Security:           <YES|NO>
  Server ID:          <server-id>
  Server GUID prefix: 44.53.<server-id-in-hex>.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[<ip-address>]:<port>
                      UDPv6:[<ip-address>]:<port>
                      TCPv4:[<ip-address>]:<physical-port>-<logical-port>
                      TCPv6:[<ip-address>]:<physical-port>-<logical-port>
```

Once the *server* is instantiated, the *clients* can be configured either programmatically or by XML (see *Discovery Server Settings*), or using environment variable `ROS_DISCOVERY_SERVER` (see *ROS_DISCOVERY_SERVER*)

Note: The *Security* configuration of the discovery server should be done through XML. See example below.

Examples

1. Launch a **default server** with id 0 (first on `ROS_DISCOVERY_SERVER`) listening on all available interfaces on UDP port '11811'. Only one server can use default values per machine.

   ```
   fastdds discovery -i 0
   ```

   Output:

   ```
   ### Server is running ###
     Participant Type:   SERVER
     Security:           NO
     Server ID:          0
   ```

(continues on next page)

(continued from previous page)

```
  Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[0.0.0.0]:11811
```

2. Launch a default server with id 1 (second on ROS_DISCOVERY_SERVER) listening on localhost with UDP port 14520. Only localhost clients can reach the server defining as *ROS_DISCOVERY_SERVER=;127.0.0.1:14520* .

```
fastdds discovery -i 1 -l 127.0.0.1 -p 14520
```

Output:

```
### Server is running ###
  Participant Type:   SERVER
  Security:           NO
  Server ID:          1
  Server GUID prefix: 44.53.01.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[127.0.0.1]:14520
```

This same output can be obtained loading the following XML configuration file `DiscoveryServerCLI.xml`:

```
<participant profile_name="participant_profile_discovery_server_cli" is_default_
↪profile="true">
    <rtps>
        <prefix>44.53.01.5f.45.50.52.4f.53.49.4d.41</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>localhost</address>
                        <port>14520</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>

<participant profile_name="second_participant_profile_discovery_server_cli">
    <rtps>
        <prefix>44.53.02.5f.45.50.52.4f.53.49.4d.41</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.36.34</address>
                        <port>8783</port>
                    </udpv4>
                </locator>
```

(continues on next page)

(continued from previous page)

```
                <locator>
                    <udpv4>
                        <address>172.20.96.1</address>
                        <port>51083</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>

<participant profile_name="secure_discovery_server_cli">
    <rtps>
        <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>0.0.0.0</address>
                        <port>11811</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
        <propertiesPolicy>
            <properties>
                <!-- Activate Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.plugin</name>
                    <value>builtin.PKI-DH</value>
                </property>

                <!-- Configure Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_ca</name>
                    <value>file://maincacert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_certificate</name>
                    <value>file://appcert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.private_key</name>
                    <value>file://appkey.pem</value>
                </property>

                <!-- Activate Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.plugin</name>
```

(continues on next page)

(continued from previous page)

```
                    <value>builtin.Access-Permissions</value>
                </property>

                <!-- Configure Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.permissions_ca</
↪name>
                    <value>file://maincacet.pem</value>
                </property>
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.governance</
↪name>
                    <value>file://governance.smime</value>
                </property>
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.permissions</
↪name>
                    <value>file://permissions.smime</value>
                </property>

                <!-- Activate Crypto:AES-GCM-GMAC plugin -->
                <property>
                    <name>dds.sec.crypto.plugin</name>
                    <value>builtin.AES-GCM-GMAC</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

```
fastdds discovery -x [PATH_TO_FILE]/DiscoveryServerCLI.xml
```

3. Launch a default server with id 1 (second on `ROS_DISCOVERY_SERVER`) listening on IPv6 address `2a02:ec80:600:ed1a::3` with UDP port 14520.

```
fast-discovery-serverd-1.0.1.exe -i 1 -l 2a02:ec80:600:ed1a::3 -p 14520
```

Output:

```
### Server is running ###
  Participant Type:   SERVER
  Security:           NO
  Server ID:          1
  Server GUID prefix: 44.53.01.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv6:[2a02:ec80:600:ed1a::3]:14520
```

4. Launch a default server with id 2 (third on `ROS_DISCOVERY_SERVER`) listening on WiFi (192.168.36.34) and Ethernet (172.20.96.1) local interfaces with UDP ports 8783 and 51083 respectively (addresses and ports are made up for the example).

```
fastdds discovery -i 2 -l 192.168.36.34 -p 8783 -l 172.20.96.1 -p 51083
```

Output:

```
### Server is running ###
  Participant Type    SERVER
  Security:           NO
  Server ID:          2
  Server GUID prefix: 44.53.02.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[192.168.36.34]:8783
                      UDPv4:[172.20.96.1]:51083
```

Using the same XML configuration file from the second example, the same output can be obtained loading a specific profile: *second_participant_profile_discovery_server_cli*.

```
fastdds discovery -x second_participant_profile_discovery_server_cli@[PATH_TO_FILE]/
↪DiscoveryServerCLI.xml
```

5. Launch a default server with id 3 (fourth on ROS_DISCOVERY_SERVER) listening on 172.30.144.1 with UDP port 12345 and provided with a backup file. If the server crashes it will automatically restore its previous state when re-enacted.

```
fastdds discovery -i 3 -l 172.30.144.1 -p 12345 -b
```

Output:

```
### Server is running ###
  Participant Type    BACKUP
  Security:           NO
  Server ID:          3
  Server GUID prefix: 44.53.03.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[172.30.144.1]:12345
```

6. Launch a default server with id 0 (first on ROS_DISCOVERY_SERVER) listening on localhost with UDP port 14520. Only localhost clients can reach the server defining as *ROS_DISCOVERY_SERVER=localhost:14520*.

```
fastdds discovery -i 0 -l localhost -p 14520
```

Output:

```
### Server is running ###
  Participant Type:   SERVER
  Security:           NO
  Server ID:          0
  Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[127.0.0.1]:14520
```

7. Launch a secure server with id 0 (first on ROS_DISCOVERY_SERVER) listening on all available interfaces on UDP port '11811'.

```
fastdds discovery -x secure_discovery_server_cli@[PATH_TO_FILE]/DiscoveryServerCLI.
↪xml
```

Output:

```
### Server is running ###
  Participant Type:   SERVER
  Security:           YES
```

(continues on next page)

(continued from previous page)

```
  Server ID:          0
  Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[0.0.0.0]:11811
```

8. Launch a server with id 0 (first on ROS_DISCOVERY_SERVER) reading specific *profile_name* configuration from XML file.

```
fastdds discovery -i 0 -x profile_name@[PATH_TO_FILE]/config.xml
```

Output:

```
### Server is running ###
  Participant Type:   SERVER
  Security:           NO
  Server ID:          0
  Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   UDPv4:[127.0.0.1]:56542
```

9. Launch a server with id 0 (first on ROS_DISCOVERY_SERVER) listening on localhost on default TCP port '42100'.

```
fastdds discovery -i 0 -t 127.0.0.1
```

Output:

```
### Server is running ###
  Participant Type:   SERVER
  Security:           NO
  Server ID:          0
  Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   TCPv4:[127.0.0.1]:42100-42100
```

10. Launch a server with id 0 (first on ROS_DISCOVERY_SERVER) listening on localhost and WiFi (192.163.6.34). Two TCP ports need to be specified because TCP transports cannot share ports.

```
fastdds discovery -i 0 -t 127.0.0.1 -q 42100 -t 192.163.6.34 -q 42101
```

Output:

```
### Server is running ###
  Participant Type:   SERVER
  Security:           NO
  Server ID:          0
  Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
  Server Addresses:   TCPv4:[127.0.0.1]:42100-42100
                      TCPv4:[192.163.6.34]:42101-42101
```

Note: When using Discovery Server over TCP, the first port shown in the output refers to the TCP Physical port and the second one to the TCP Logical port (see *TCP Transport*).

Note: A server can be instantiated just by passing the port arguments `-p` and `-q`. Fast DDS CLI will use the default values of the IP addresses, that is, `0.0.0.0` for UDP and `127.0.0.1` for TCP.

6.39.2 shm

Provides maintenance tasks related with *Shared Memory Transport*. Shared Memory transport creates *Segments*, blocks of memory accessible from different processes. Zombie files are memory blocks that were reserved by shared memory and are no longer in use which take up valuable memory resources. This tool finds and frees those memory allocations.

```
fastdds shm [<shm-command>]
```

Sub-command	Description
`clean`	Cleans SHM zombie files.

Option	Description
`-h -help`	Produce help message.

6.39.3 xml

Checks if a given xml profile is well formed, by matching it against a XSD schema. If the given input to the command is a path to a folder instead of a path to a file,all xml files contained in the folder will be validated.

This validation consists in checking the lack of parameters, values bounds, expected values data types and main profile structure. For further information see *Creating an XML profiles file*.

```
fastdds xml [<xml-command>]
```

Sub-command	Description
`validate`	Checks a xml profile by matching it against a XSD schema.

Option	Description
`-h --help`	Produce help message.
`-d --debug`	Print debug information (disabled by default)
`-x --xsd_file`	XSD schema for validation (not required, Fast DDS schema is used by default)

Example

```
fastdds xml validate my_profile.xml
```

6.40 Docker Images

eProsima provides the Fast DDS and the Fast DDS Suite Docker images for those who want a quick demonstration of Fast DDS running on an Ubuntu platform. They can be downloaded from eProsima's downloads page.

This Docker images were built for Ubuntu 22.04 (Jammy Jellyfish).

To run a container you need Docker installed. From a terminal, run:

```
sudo apt install docker.io
```

6.40.1 Leveraging Fast DDS SHM in Docker deployments

By default, *Fast DDS* enables both a *Shared Memory Transport* and *Data-sharing delivery* (when the configuration allows it, see *Constraints*). The way *Fast DDS* utilizes to find out whether a remote *DomainParticipant* is running on the same host as the local one is through a hashing of the network interfaces, which entails that two Docker container which are not sharing the same network stack will be detected as two different hosts, and therefore *Fast DDS* will defer to communicating through the *UDP Transport* instead.

To enable the use of both the *Shared Memory Transport* and *Data-sharing* in Docker deployments, two additional options need to be passed to the *docker run* command, those are:

- –network=host: This option shares the host's network stack with the containers, and therefore *Fast DDS* will be able to identify them as the same host.
- –ipc=host: This option shares the host's shared memory mechanism with the containers. Without it, *Fast DDS* will identify both containers as the same host, but since they will have separate shared memory spaces, they will not be able to communicate with one another. The use of this option is the Docker recommended way of sharing shared memory between containers, in opposition of for instance sharing the */dev/shm* volume in Linux machines. A more advanced user could set the flag to *shared*, thus sharing the shared memory mechanism of one of the container with the others.

```
docker run -it --rm --network=host --ipc=host [OPTIONS] <docker-image>
```

6.40.2 Fast DDS Image

This Docker image contains the Fast DDS library and its dependencies, ready to be used in a final user application. This includes:

- *eProsima Fast DDS libraries and examples*: Fast DDS libraries bundled with several examples that showcase a variety of capabilities of eProsima's Fast DDS implementation.

To load this image into your Docker repository, from a terminal, run:

```
docker load -i "ubuntu-fastdds <FastDDS-Version>.tar"
```

You can run this Docker container as follows:

```
docker run -it ubuntu-fastdds:<FastDDS-Version>
```

From the resulting Bash Shell you can run each feature.

Fast DDS Examples

Included in this Docker container is a set of binary examples that showcase several functionalities of the Fast DDS libraries. These examples' path can be accessed from a terminal by typing:

```
goToExamples
```

From this folder, you can access all examples, both for DDS and RTPS layers.

Hello World Example

This is a minimal example that will perform a Publisher/Subscriber match and start sending samples.

```
goToExamples
cd dds/HelloWorldExample/bin
tmux new-session "./DDSHelloWorldExample publisher 0 1000" \; \
     split-window "./DDSHelloWorldExample subscriber" \; \
     select-layout even-vertical
```

This example is not constrained to the current instance. It's possible to run several instances of this container to check the communication between them by running the following from each container.

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample publisher
```

or

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample subscriber
```

Benchmark Example

This example creates either a Publisher or a Subscriber and on a successful match starts sending samples. After a few seconds the process that launched the Publisher will show a report with the number of samples transmitted.

On the subscriber side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark subscriber udp
```

On the publisher side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark publisher udp
```

6.40.3 Fast DDS Suite Image

This Docker image contains the complete Fast DDS suite. This includes:

- *eProsima Fast DDS libraries and examples*: Fast DDS libraries bundled with several examples that showcase a variety of capabilities of eProsima's Fast DDS implementation.
- *Shapes Demo*: eProsima Shapes Demo is an application in which Publishers and Subscribers are shapes of different colors and sizes moving on a board. Each shape refers to its own topic: Square, Triangle or Circle. A single instance of the eProsima Shapes Demo can publish on or subscribe to several topics at a time.

  You can read more about this application on the Shapes Demo documentation page.

- *Fast DDS Monitor*: eProsima Fast DDS Monitor is a graphical desktop application aimed at monitoring DDS environments deployed using the *eProsima Fast DDS* library. Thus, the user can monitor in real time the status of publication/subscription communications between DDS entities. They can also choose from a wide variety of communication parameters to be measured (latency, throughput, packet loss, etc.), as well as record and compute in real time statistical measurements on these parameters (mean, variance, standard deviation, etc.).

  You can read more about this application on the Fast DDS Monitor documentation page.

- *DDS Router*: eProsima DDS Router is an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of eProsima DDS Router. This is achieved by deploying a DDS Router on an edge device of each local network so that the DDS Router routes DDS traffic from one network to the other through WAN communication.

  You can read more about this application on the DDS Router documentation website.

- *Plotjuggler eProsima Edition*: eProsima Fast DDS Visualizer Plugin is a plugin for the *PlotJuggler* application. PlotJuggler is a graphical desktop application providing visualization features of data series, time series, X-Y plots. It also adds data management features, such as data import and export, custom and built-in data manipulation functions, data series merges, etc. Also, this software supports many different layouts, with dynamic, rich and user-friendly customization.

  You can read more about this application on the Plotjuggler eProsima Edition documentation website.

To load this image into your Docker repository, from a terminal run

```
docker load -i "ubuntu-fastdds-suite <FastDDS-Version>.tar"
```

You can run this Docker container as follows

```
xhost local:root
docker run -it --privileged -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
ubuntu-fastdds-suite:<FastDDS-Version>
```

From the resulting Bash Shell you can run each feature.

Fast DDS Examples

Included in this Docker container is a set of binary examples that showcase several functionalities of the Fast DDS libraries. These examples' path can be accessed from a terminal by typing

```
goToExamples
```

From this folder you can access all examples, both for DDS and RTPS. We detail the steps to launch two such examples below.

Hello World Example

This is a minimal example that will perform a Publisher/Subscriber match and start sending samples.

```
goToExamples
cd dds/HelloWorldExample/bin
tmux new-session "./DDSHelloWorldExample publisher 0 1000" \; \
    split-window "./DDSHelloWorldExample subscriber" \; \
    select-layout even-vertical
```

This example is not constrained to the current instance. It's possible to run several instances of this container to check the communication between them by running the following from each container.

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample publisher
```

or

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample subscriber
```

Benchmark Example

This example creates either a Publisher or a Subscriber and on a successful match starts sending samples. After a few seconds the process that launched the Publisher will show a report with the number of samples transmitted.

On the subscriber side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark subscriber udp
```

On the publisher side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark publisher udp
```

Shapes Demo

To launch the Shapes Demo, from a terminal run

```
ShapesDemo
```

eProsima Shapes Demo usage information can be found on the Shapes Demo documentation.

Fast DDS Monitor

To launch the Fast DDS Monitor, from a terminal run

```
fastdds_monitor
```

eProsima Fast DDS Monitor user manual can be found on the Fast DDS Monitor documentation.

DDS Router

This example configures a DDS Router to communicate a publisher and subscriber running in different DDS Domains.

Run the following command to create the DDS Router *yaml* configuration file (`/config.yml`).

```
echo "version: v2.0
participants:
  - name: simple_dds_participant_0
    kind: local
    domain: 0
  - name: simple_dds_participant_1
    kind: local
    domain: 1" > /config.yml
```

Then execute the following command to run the Publisher in Domain 0, the Subscriber in Domain 1, and the DDS Router communicating both Domains.

```
goToExamples
cd dds/BasicConfigurationExample/bin
tmux new-session \
    "ddsrouter --config-path /config.yml" \; \
    split-window -h "./BasicConfigurationExample publisher --domain 0 --interval 1000 --
↪transport udp" \; \
    split-window -v "./BasicConfigurationExample subscriber --domain 1 --transport udp"
```

eProsima DDS Router usage information can be found on the DDS Router documentation.

PlotJuggler eProsima Edition

To launch the PlotJuggler eProsima Edition, from a terminal run

```
plotjuggler
```

eProsima PlotJuggler eProsima Edition usage information can be located on the PlotJuggler eProsima Edition User Manual. Information about the release lifecycle can be found here.

6.41 Version 2.12.2

This release includes the following **features**:

1. Methods to configure *transport scenarios*
2. Support `Autofill port` (*automatically set the port*) for TCP Transport
3. Support *TCP for Discovery Server* CLI and environment variable
4. Define a *super client by environment variable*
5. Change serialize function default behaviour to omit the data representation
6. `LARGE_DATA` Participants logic with same listening ports
7. TCP Client&Server Participant Decision-Making logic
8. Expose *Authentication Handshake Properties*
9. Enabling multiple interfaces through whitelist in TCP servers
10. Add macOS and Ubuntu Github CI

This release includes the following **improvements**:

1. Improve environment variable substitution algorithm
2. Upgrade dependency version to last patch version in .repos file
3. Rerun failed tests with ctest option instead of colcon's
4. Remove unnecessary TCP warning
5. Update PR template to include check for PR description, title and backports
6. Improvements in GitHub CI

This release includes the following **fixes**:

1. Fix TCP reconnection after open logical port failure
2. TCP unique client announced local port
3. TCP non-blocking send
4. Fix wrong log info messages on TCP
5. Improve `IgnoreNonExistentSegment` test
6. Use `SO_EXCLUSIVEADDRUSE` for Win32 unicast listening sockets
7. Fix dns filter in CMakeLists file for tests
8. Fix bad-free when receiving malformed DATA_FRAG submessage
9. Fix memory problem related to ciphering payload
10. Fix CVE-2023-50257
11. Fix build with TLS, but not security
12. Fix comparison in `remove_from_pdp_reader_history`
13. Fix data race in `PDPListener` and `SecurityManager`
14. Discard already processed samples on `PDPListener`
15. Fix .repos versions

16. Fix the shared memory cleaning script
17. Fix data race on writer destruction while sending heartbeat
18. Return `const` reference to the shared pointer instead of a copy in `get_log_resources`
19. Ignore `0x8007` if coming from other vendor
20. Fix Doxygen docs warnings and prepare for compiling with Doxygen 1.10.0
21. Include variety of terminate process signals handler in discovery server
22. Add missing `TypeLookup` listeners
23. Add a keyed fragmented change to the reader data instance only when its completed
24. Fix data race on PDP
25. Check History QoS inconsistencies
26. Make DataWriters always send the key hash on keyed topics
27. Prevent index overflow and correctly assert the end iterator in DataSharing
28. Fix uninitialized member in `RTPSParticipantAttributes`
29. Remove unnecessary `std::move` in `FileWatch.hpp` causing warning
30. Add missing thread include
31. Add missing virtual destructor for `StatisticsAncillary`
32. Protect asio exception
33. `TCPSendResources` cleanup
34. Downgrade CMake version to 3.20

6.42 Previous versions

6.42.1 Version 2.12.1

Note: This release upgrades the following Fast DDS dependencies:

- Fast CDR v2.0.0
- Fast DDS-Gen v3.0.1

As Fast DDS dependencies have been upgraded to new major releases, depending on the types defined in the IDL files, it might be required to modify the user application source code besides recompiling it (more information can be found in the corresponding release notes).

This release includes the following **improvements**:

1. Support for linking with Fast CDR v1.
2. The period for the timer within the *DisablePositiveACKsQosPolicy* is now updatable.
3. Log error message upon receiver resource creation failure.
4. CI and repository improvements.
5. Simplify code in CDRMessage.

This release includes the following **fixes**:

1. **Fast DDS bugfixes**
    1. Fix transient local durability for reliable readers using intra-process and data-sharing.
    2. Use STL implementation of Timed/RecursiveTimedMutex when MSVC >= 19.36.
    3. Fix updatability of immutable DataWriterQos.
    4. Fix the clang build for clang 14.
    5. Fix remote locators filtering when whitelist provided.
    6. Fix Data Race when updating liveliness changed in WLP.
    7. Add XML parser bit_bound bounds check.
    8. Fix missing mandatory attribute check in XML parser struct type.
    9. SHM transport: ignore non-existing segment on pop.
    10. Fix: mac address overflow on Windows.
2. CI fixes:
    1. Fix flow controllers unit tests compilation when using Fast CDR from thirdparty.
    2. PubSubAsReliable test fix.
    3. FileWatchTest fix for github windows CI.

Note: When upgrading to version 2.12.1 it is **advisable** to regenerate generated source from IDL files using Fast DDS-Gen v3.1.0.

6.42.2 Version 2.12.0

Note: This release upgrades the following Fast DDS dependencies:

- Fast CDR v2.0.0
- Fast DDS-Gen v3.0.1

Please, read also the release notes of Fast DDS-Gen v3.0.0 to be aware of every possible break in the application code.

As Fast DDS dependencies have been upgraded to new major releases, depending on the types defined in the IDL files, it might be required to modify the user application source code besides recompiling it (more information can be found in the corresponding release notes).

Note: There is a minor API break with previous v2.x versions: `MEMBER_INVALID` identifier was declared using `#define`. In order to prevent polluting the user workspace, it has been transformed into a `constexpr` within `eprosima::fastrtps::types` namespace.

This release includes the following **features**:

1. *New participant property* to configure SHM transport metatraffic behavior.
2. Exposed custom payload pool on DDS *DataWriter* and *DataReader* declaration.
    1. Feature example.

3. *Processing environment variables in XML text*.
4. Dependencies
    1. Upgrade internal type supports using latest Fast DDS-Gen release v3.0.0. This release introduces the following features:
        1. *Support for @optional builtin annotation <optional_members>*.
        2. *Support for @extensibility builtin annotation <extensibility>*.
    2. Upgrade Fast CDR submodule to v2.0.0 introducing XCDR encoding version 2.

This release includes the following **improvements**:

1. `fixed_string` comparison operators.
2. Remove mutex from *TimedEventImpl* (#3745, #3760)
3. Performance improvements on intraprocess and datasharing.
4. Improve Shared Memory resilience to crashing participants.
5. Improve scripts shebang portability.
6. Use `foonathan_memory` to reduce allocations in SharedMemManager.

This release includes the following **fixes**:

1. **Fast DDS bugfixes**
    1. Fixed XMLParser null-dereference when parsing log configuration.
    2. Allow participant XML profiles with no <rtps> tag.
    3. Fix encapsulation format in Writer Liveliness Protocol.
    4. Fix *DomainParticipant::register_remote_type* return when negotiating type.
    5. Fix strict real-time feature when using Flow Controller feature.
    6. Fix ParameterPropertyList increment operators.
    7. Fix bad-free when receiving malformed DATA submessage.
    8. Fix asymmetric whitelist matching.
    9. Fix heap-use-after-free on XMLElementParser.
    10. Fix History remove change return statement.
2. **CI fixes**
    1. Fix RemoteBuiltinEndpointHonoring blackbox test.
    2. Improve repository workflows.
    3. Use *FASTRTPS_NO_LIB* on unittest root folder.
    4. Fix Windows workflow.
3. **Tools**
    1. Remove C++11 check in `fastdds-discovery-server` CLI tool.
4. **Examples**
    1. Fix HelloWorldDataSharing data type.
5. **Documentation**

    1. Doxygen typos.

6. **Repository**
    1. Remove 2.9.x as active branch.
7. **Non Tier 1 support**
    1. Fixed SHM in 32-bit architectures.
    2. Fix warning on Win32 architecture.

Note: Upgrading to version 2.12.0 **requires** to regenerate generated source from IDL files using Fast DDS-Gen v3.0.1.

6.42.3 Version 2.11.2 (EOL)

This release includes the following **improvements**:

1. Improve Shared Memory resilience to crashing participants
2. User configuration for *Shared Memory metatraffic*
3. Performance improvements on intraprocess and data-sharing

This release includes the following **fixes**:

1. Remove Mutex from TimedEventImpl
2. Replace uint64_t by 8 in `alignas` specifier
3. Fix XMLParser null-dereference in `parseLogConfig`

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.4 Version 2.11.1 (EOL)

This release includes the following **improvements**:

1. Correct CONTRIBUTING.md typo
2. Improve validation on PID_PROPERTY_LIST deserialization
3. Apply eProsima brand style to Fast DDS repository
4. Fix spelling mistake: SUBSTRACTION to SUBTRACTION

This release includes the following **fixes**:

1. Fixed long-standing reconnection issue on SHM transport
2. Added missing include
3. Fixed Boost handle usage regression
4. Fix StatelessWriter locators filtering
5. Avoid double definition of FASTDDS_ENFORCE_LOG_INFO
6. Explicitly register type object in ContentFilteredTopicExample

7. Properly handle zero-sized payloads on dynamic memory payload pools

---

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.5 Version 2.11.0 (EOL)

This release includes the following **features**:

1. *Ignore every local endpoint within the DomainParticipant preventing local matching*.
2. Extend DynamicDataHelper API providing a `print` overload with `std::ostream` as parameter.
3. *TypeLookup Service Configuration*.
4. Static Discovery XSD Schema.

This release includes the following **improvements**:

1. **Fast DDS improvements**
    1. Assign minimum available participant ID to new participants.
    2. Export symbols correctly on ContentFilteredTopic.
    3. Improve content filter expression parameters check and verbosity.
    4. Check TCP headers endianness.
    5. Security module: distinguished names (DN) comparison.
2. **Fast DDS deprecation**
    1. DDS:Crypto:AES-GCM-GMAC configuration using Property Policy QoS (security vulnerability).
3. **CI improvements**
    1. Include BitmapRange unit tests.
    2. Support for running some tests in parallel.
    3. Windows workflow.
4. **Build system**
    1. Improve CMake target loading. Removal of `FASTDDS_STATIC` CMake option.
    2. Avoid auto-linkage using CMake.
5. **Dependencies**
    1. Upgrade internal type supports using latest Fast DDS-Gen release v2.5.1.
    2. Upgrade Fast CDR submodule to v1.1.0.
6. **Examples**
    1. Admit XML configuration files in AdvanceConfigurationExample.
    2. New Discovery Server example.

This release includes the following **fixes**:

1. **Fast DDS bugfixes**
    1. Fix crash when creating two participants with the same fixed participant ID.

2. Fix crash when calling *on_requested_deadline_missed()* callback.
3. Fix crashes caused by not capturing every Fast CDR exception.
4. Correctly resolve aliases in DDSSQLFilter.
5. Wait for log background thread initialization on the first queued entry.
6. Fix data race when accessing `WRITE_PARAM_DEFAULT` static variable.
7. Fix partition copy in QoS.
8. Fix Data-Sharing delivery when data_count is zero.
9. Fix API Fast DDS v2.10.0 API break calling correctly. *on_participant_discovery()* callbacks.
10. Security module: Honor *Allow Unauthenticated Participants* flag.
11. Fix concurrent access to *load_profiles()*.
12. Fix UBSan (Undefined Behavior Sanitizer) issues.
13. Improve Doxygen documentation about DomainParticipantListener discovery callbacks.

2. **XSD fixes**
    1. Set TransportDescriptor kind parameter as optional.
    2. Correctly assign QoS to the proper endpoint.
    3. Add missing tags.
3. **CI fixes**
    1. Fix null dereference in fuzzer code.
    2. Limit Thread Sanitizer memory usage to prevent runner shutdown.
    3. Use correct time unit in latency tests.
    4. Run communication tests.
4. **Examples**
    1. Correct DDS entity deletion order.
5. **Installer generation**
    1. Add documentation fallback when the documentation tag is not found.
6. **Repository**
    1. Remove 2.1.x as active branch.
    2. Remove 2.8.x as active branch.
7. **Non Tier 1 support**
    1. Fix build on MSVC 19.36.
    2. Forward compatibility with Boost inter-process 1.74+.
    3. Include missing header files required for compiling with GCC 13.
    4. QNX build fixes.
    5. Fix build issues in RPM systems.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.6 Version 2.10.2

This release includes the following **improvements**:

1. Fix Data-Sharing delivery when data_count is zero
2. Improve performance of intraprocess plus data-sharing
3. Improve content filter expression parameters checks and verbosity
4. Improve validation on PID_PROPERTY_LIST deserialization
5. Participant ignore local endpoints
6. Pick smallest available participant ID for new participants
7. Improve endpoint QoS XML tags
8. Forward compatibility with boost interprocess 1.74+
9. Cap Thread Sanitizer memory usage to prevent runner shutdown
10. Allow participant XML profile with no <rtps> tag
11. Add unsupported note in API documentation to new ignore DomainParticipantListener callbacks
12. Add documentation version fallback

This release includes the following **bugfixes**:

1. Fixed long-standing reconnection issue on SHM transport
2. Fix null dereference when fuzzing
3. Fix segfault when creating two participant with same fixed id
4. Fix UBSan (Undefined Behavior Sanitizer) issues
5. Fix listener selection for on_requested_deadline_missed
6. Fix build on msvc 19.36.32528
7. Fix XML schema to set Transport descriptor kind as NOT mandatory
8. Fix missing includes
9. Fix overhead time unit
10. Fix request reply example spelling typo
11. Fix topic deletion after endpoint in examples
12. Fix Data-Sharing delivery when data_count is zero
13. Wait for log background thread initialization on the first queued entry
14. Fix alias resolve in DDSSQLFilter
15. Fix partition copy in QoS
16. Fix StatelessWriter locators filtering
17. Fix XMLParser null-dereference in parseLogConfig

18. Fix encapsulation format in WLP
19. Replace uint64_t by 8 in alignas specifier
20. Capture all Fast CDR exceptions
21. Security module: Honor allow_unauthenticated_participants flag
22. Explicitly register type object in ContentFilteredTopicExample
23. Avoid double definition of FASTDDS_ENFORCE_LOG_INFO
24. Fix API Fast DDS v2.10.0 API break calling correctly on_participant_discovery callbacks
25. Remove mutex from TimedEventImpl

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.7 Version 2.10.1

This release includes the following **feature** in an ABI compatible way:

1. *ignore_participant()* implementation

This release includes the following **bugfixes**:

1. Fix repeated matched event notification.
2. Fix regression introduced by #3396.
3. Initial acknack backoff.
4. Fix regular expression in XML schema.
5. Correctly assign multicast port to multicast initial peers.

This release includes the following **CI improvements**:

1. Update runner and GCC version for Thread Sanitizer job.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.8 Version 2.10.0

This minor release includes several new features, improvements and bugfixes.

Note: Mind that, even though this release is API compatible with previous v2.x versions, it is *NOT* ABI compatible with previous versions. This means that applications upgrading Fast DDS to v2.10.0 will require recompilation, though not source code modification.

Note: It is also advisable to regenerate the type support from the IDL files using Fast DDS-Gen v2.4.0. Furthermore, if upgrading to v2.10.0, it is also recommended to upgrade Fast CDR to v1.0.27.

This release includes the following **features**:

1. New *on_unacknowledged_sample_removed callback* in `DataWriterListener`.
2. *Secure Discovery Server*.
3. DomainParticipant ignore empty API.
4. RTPS `ReaderListener::on_incompatible_type` and `WriterListener::on_incompatible_type` empty API.

This release includes the following **improvements**:

1. **Fast DDS improvements**
    1. Improve behavior when `STRICT_REALTIME` *CMake option* is not enabled.
    2. Using functors for `for_matched_readers` parameter.
    3. Improve auto GAPs in Data Sharing.
    4. Use standard value for `PID_RELATED_SAMPLE_IDENTITY`.
2. **Contributions and repository quality**
    1. Update Pull Request template.
    2. Update foonathan_memory quality declaration.
    3. Update XSD schema.
    4. Make network headers private avoiding exposing non-public API.
    5. Improve Doxygen documentation for `ResourceLimitsQosPolicy`.
3. **Examples**
    1. New *Request-Reply example*.
4. **CI improvements**
    1. New workflow to check documentation build.
    2. ASAN workflow updated to use Ubuntu 22.04.
5. **Dependencies**
    1. Upgrade internal type supports using latest Fast DDS-Gen release v2.4.0.
    2. Upgrade Fast CDR submodule to v1.0.27.
6. **Fast DDS CLI**
    1. Handle `SIGTERM` signal.
7. **Community supported platforms**
    1. *QNX 7.1 build infrastructure*.

This release includes the following **fixes**:

1. **Security vulnerability**
    1. Fix chain of trust issues with a single CA certificate.
2. **Bugfixes**
    1. Fix RTPS StatelessWriter ACK check.
    2. ASAN (Address Sanitizer) fixes.

3. UBSan (Undefined Behavior Sanitizer) fixes.
4. Export public API correctly in Windows.
5. Correctly handle builtin endpoints mask.
6. Fix backwards compatibility using SHM communication.
7. Protect against uncaught exception in SHM segment creation.
8. Fix build for GCC 5.
9. Validity check for first sequence number.
10. Fix crash when enabling DisablePositiveACKsQoSPolicy with remote best-effort readers.

3. **Synchronization fixes**
    1. Take mutex when removing local reader in WLP.
    2. Fix data races in SecurityManager authentication process.
4. **CI fixes**
    1. Fix test building when using `GTEST_INDIVIDUAL` *CMake option*.
    2. Fix overflow in received samples in performance tests.
5. **Example fixes**
    1. Avoid creating entities within callbacks in DynamicHelloWorldExample.
    2. Remove Asio dependency from DeadlineQoSExample.
6. **Repository fixes**
    1. Remove 2.7.x as active branch.
7. **Community supported platforms**
    1. Include right header when building for iOS.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.9 Version 2.9.2 (EOL)

This release includes the following ABI compatible **improvements**:

1. **Library improvements**
    1. Improve behavior when `STRICT_REALTIME` *CMake option* is not enabled.
    2. Using functors for `for_matched_readers` parameter.
    3. Improve auto GAPs in Data Sharing.
    4. Improve content filter expression parameters check and verbosity.
    5. Improve validation on `PID_PROPERTY_LIST` deserialization.
2. **Fast DDS CLI**
    1. Handle `SIGTERM` signal.

This release includes the following **bugfixes**:

1. **Security vulnerability**
    1. Fix chain of trust issues with a single CA certificate.
2. **Library bugfixes**
    1. Fix RTPS StatelessWriter ACK check.
    2. UBSan (Undefined Behavior Sanitizer) fixes.
    3. Fix backwards compatibility using SHM communication.
    4. Correctly handle builtin endpoint mask.
    5. Fix crash when enabling DisablePositiveACKsQosPolicy with remote best-effort readers.
    6. Validity check for first sequence number.
    7. ASAN (Address Sanitizer) fixes.
    8. Correctly assign multicast port to multicast initial peers.
    9. Protect against uncaught exception in SHM segment creation.
    10. Initial acknack backoff.
    11. Fix crash when calling *on_requested_deadline_missed()* callback.
    12. Security module: Honor `allow_unauthenticated_participants` flag.
    13. Fix crashes caused by not capturing every Fast CDR exception.
    14. Correctly resolve aliases in DDSSQLFilter.
    15. Wait for log background thread initialization on the first queued entry.
    16. Fix partition copy in QoS.
    17. Fix Data-Sharing delivery when `data_count` is zero.
    18. Fix StatelessWriter locators filtering.
    19. Avoid double definition of `FASTDDS_ENFORCE_LOG_INFO`.
    20. Fixed long-standing reconnection issue on SHM transport.
3. **CI fixes**
    1. Fix test building when using `GTEST_INDIVIDUAL` *CMake* option.
    2. Use correct time unit in latency tests.
4. **Synchronization fixes**
    1. Take mutex when removing local reader in WLP.
    2. Fix data races in SecurityManager authentication process.
5. **Example fixes**
    1. Avoid creating entities within callbacks in DynamicHelloWorldExample.
    2. Remove Asio dependency from DeadlineQosExample.
    3. Correct DDS entity deletion order.
    4. Explicitly register TypeObject in ContentFilteredTopicExample.
6. **Installer generation**
    1. Add documentation fallback when the documentation tag is not found.

7. **Non Tier 1 Support**
    1. Fix build for GCC5.
    2. Fix build on MSVC 19.36.
    3. Include right header when building for iOS.
    4. Forward compatibility with Boost inter-process 1.74+.
    5. Include missing header files required for compiling with GCC13.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.10 Version 2.9.1 (EOL)

This release includes the following **bugfixes**:

1. **Synchronization fixes:**
    1. Fix deadlock in Writer Liveliness Protocol (WLP) using intraprocess.
    2. Fix data race in *DomainParticipant::set_listener()*.
    3. Fix deadlock on TLS closure.
2. **Other fixes:**
    1. Fix notification lost.
    2. Fix `total_read_` to be consistent with Reader's History after *DataReader::get_first_untaken_info()*.
    3. Use shared pointers for internal singletons.
    4. Support CCache on Windows.
    5. Avoid null dereference on fuzzer.
    6. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.11 Version 2.9.0 (EOL)

This minor release includes several new features (new log macros to avoid conflicts with external libraries), behavior change (default memory management policy is no longer *PREALLOCATED_MEMORY_MODE*, enable by default Fast DDS Statistics module but only taking statistics metrics if the corresponding Statistics DataWriter is enabled), performance improvements, CI improvements (including address sanitizer jobs), and several bug fixes.

Note: Mind that, even though this release is API compatible with previous v2.x versions, it is *NOT* ABI compatible with previous versions. This means that applications upgrading Fast DDS to v2.9.0 will require recompilation, though not source code modification.

Note: It is also advisable to regenerate the type support from the IDL files using Fast DDS-Gen v2.3.0. Furthermore, if upgrading to v2.9.0, it is also recommended to upgrade Fast CDR to v1.0.26.

This release includes the following **features**:

1. New log macros *EPROSIMA_LOG_INFO*, *EPROSIMA_LOG_WARNING* and *EPROSIMA_LOG_ERROR*.
2. Add ENABLE_OLD_LOG_MACROS CMake option to support *disabling the compilation of previous log macros*.

This release includes the following **behavior changes**:

1. Default *memory management policy* set to *PREALLOCATED_WITH_REALLOC_MEMORY_MODE*.
2. Statistics metrics are only calculated/accumulated when their corresponding DataWriter is enabled in *Fast DDS Statistics Module*.
3. Enable FASTDDS_STATISTICS *CMake option* by default.

This release includes the following **improvements**:

1. **CI improvements:**
    1. Add address sanitizer job for Fast DDS library.
    2. Add address sanitizer job for Discovery Server test suite.
2. Upgrade Fast CDR submodule.

This release includes the following **bugfixes**:

1. **Synchronization fixes:**
    1. Fix deadlock when removing remote DomainParticipants by expired liveliness when using Security.
    2. Protect *DomainParticipant::set_listener()* avoiding null reference.
    3. Fix data race on WriterProxy::stop while TimedEvent is being triggered.
    4. Protect creation/destruction of Boost's named_mutex.
2. **CI fixes:**
    1. Fix Statistics Module test suite.
    2. Fix recurring data races in test suite.
    3. Fix thread sanitizer job keeping Ubuntu 20.04.
3. **Other:**
    1. Fix Topic creation when registering a type name different from the internal TypeSupport name.
    2. Fix communication with asymmetric *Ignore Participant flags*.
    3. Several dependencies fixes upgrading to Ubuntu 22.04.
    4. Disable error logged when DomainParticipant profile is not found.
    5. Fix CMake for Fast DDS use as submodule.
    6. Upgrade internal type supports generated with Fast DDS-Gen v2.3.0.
    7. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.12 Version 2.8.2 (EOL)

This release includes the following **improvements**:

1. Add ASAN CI tests for Fast DDS and Discovery Server
2. Mirror master onto both 2.8.x & 2.9.x
3. Doxygen documentation: add deprecation notice to *ThroughputControllerDescriptor*
4. Several fixes to remove warnings in Ubuntu Jammy (22.04)
5. Improve behavior when *HAVE_STRICT_REALTIME* is not set
6. Using functors in *StatefulWriter.cpp* for_matched_readers
7. Fix build on old compilers
8. Avoid creation of DynamicTypes on example
9. Implement a validity check for *firstSN*

This release includes the following **bugfixes**:

1. Fix bug in Topic creation with different Type Name
2. Fix tests failing with subprocess aborted error
3. Fix communication with asymmetric ignoreParticipantFlags
4. Added *ignore_participant_flags()* to *Blackbox_FastRTPS PubSubReader*.
5. Fix Deadlock in *remove_participant* (ResourceEvent thread) when compiled WITH_SECURITY
6. Fix failed tests when compiling with statistics enabled
7. Fix Windows *StatistisQosTests.cpp* linkage and Failed test
8. Fixing deadlock in WLP
9. Fix notification lost
10. Fix *StatelessWriter* ACK check
11. Fix *total_unread_* consistent with reader's history upon *get_first_untaken_info()*
12. Fix chain of trust issues
13. Fixed StatisticsSubmessageData unaligned access
14. Fix build error when *GTEST_INDIVIDUAL* is OFF
15. Correctly handle builtin endpoints mask
16. Added missing mutex to *WLP::remove_local_reader*
17. Handle SIGTERM in fast discovery server
18. Improve auto gaps in data sharing
19. Replaced *SecurityManager* temporary *ProxyDatas* with *ProxyPools*
20. Fix crash when *disable_positive_acks* is enable and the remote reader is best-effort

21. Protect from uncaught exception during SHM Segment creation
22. Fix asio dependency
23. Include the right header when building for iOS

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.13 Version 2.8.1 (EOL)

This release includes the following **bugfixes**:

1. **Statistics module fixes:**
    1. Increase statistics DataWriter history.
    2. Fix Statistics module CI.
    3. Install Statistics IDL file.
    4. Fix for building in old compilers.
    5. Fix core dumped in *delete_contained_entities()*.
1. **Address sanitizer fixes:**
    1. Add ASAN CI job and `SANITIZE` *CMake option*.
    2. Fixes reported by address sanitizer.
1. **Synchronization fixes:**
    1. Fix data races when creating DataWriters.
    2. Ensure `shared_mutex` implementation is consistent throughout supported platforms.
1. **Other fixes:**
    1. Include missing ReadCondition header.
    2. Fix selection of output locators.
    3. Fix null-dereference on `parseXMLEnumDynamicType`.
    4. Include `2.8.x` branch release support.
    5. Send GAPs correctly when using separate sending.
    6. Fixes for building in old compilers.
    7. Fix *DataReader::read_next_instance()* and *DataReader::take_next_instance()* implementation.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.14 Version 2.8.0 (EOL)

This minor release includes several new features, performance improvements (especially in the case of topics with many instances), CI improvements (including the ability to run the CI in Android emulators or devices), and several bug fixes.

Note: Mind that, even though this release is API compatible with previous v2.x versions, it is *NOT* ABI compatible with previous versions. This means that applications upgrading Fast DDS to v2.8.0 will require recompilation, though not source code modification.

Note: It is also advisable to regenerate the type support from the IDL files using Fast DDS-Gen v2.2.0. Furthermore, if upgrading to v2.8.0, it is also recommended to upgrade Fast CDR to v1.0.25.

This release includes the following **features**:

1. Full *Ownership* and *OwnershipStrength* QoS support
2. *External locators*
3. UDPv6 support for *fast-discovery-server tool* and *ROS_DISCOVERY_SERVER*
4. *XML configuration support for statistics DataWriters QoS*
5. *SNI support*
6. *Propagate PropertyQoS properties when explicitly set*
7. Add API to createRTPSWriter with a custom pool
8. Add `std::string::compare` API to `fixed_string`
9. Get WAN address API in TCPv4 transport descriptors
10. Adding `DomainParticipantFactory::get_shared_instance()` API

This release includes the following **improvements**:

1. **Performance improvements:**
    1. Skip writer_removed processing for unaccounted instances
    2. Improve GUID_t operator< performance
2. **CI improvements:**
    1. Add optional parameters to thread-sanitizer job
    2. Enable Android testing on device
3. **Examples:**
    1. Update BasicConfigurationExample to allow set up TTL
    2. Add Guid info to BasicConfiguration Example `cout`
4. **Internal implementation improvements:**
    1. Add script to generate idl files
    2. Group set_qos_from_attributes free functions into a separate file
    3. Update script for generating idl files
    4. Set `last_heartbeat_count_` private member of WriterProxy as atomic

5. Android Improvements
6. Upgrade Fast CDR submodule

This release includes the following **bugfixes**:

1. **Synchronization fixes:**
    1. Fix datarace using writer's locator selectors
    2. Add lock guard at changing SHM port listener status members
    3. Add atomic variable to prevent datarace in FlowController
    4. Disable RTPSParticipantImpl after removing it from RTPSDomain participants list
    5. Fixing datarace on listener callbacks
    6. Protect access to reader listeners
    7. Use thread-safe localtime function in unix distributions
    8. Fixed usage of uninitialised ifreq
    9. Adding protection to id_counter access
2. **Repository fixes:**
    1. Fix spelling mistake
    2. Add python3 dependency to package.xml
3. **Other:**
    1. Fix null dereference on parseXMLBitsetDynamicType
    2. Change internal include path of nlohmann/json header file
    3. Instance allocation consistency
    4. Fix complex member printing for DynamicDataHelper
    5. Fix initialization order in mock
    6. Upgraded internal type supports

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.15 Version 2.7.2 (EOL)

This release includes the following **improvements**:

1. Skip `writer_removed` processing for unaccounted instances.
2. Improve `GUID_t` operator < performance.

This release also includes the following **bugfixes**:

1. Fix complex member printing for DynamicDataHelper.
2. Add python3 dependency.
3. Fix selection of output locators.
4. Fix data races when creating DataWriters.

5. Fix null dereferences on XML parser.
6. Send GAPs correctly when using separate sending.
7. Install Statistics IDL file.
8. Fixes for building in old compilers.
9. Fix several deadlocks.
10. Fix communication with asymmetric *Ignore Participant flags*.
11. Fix notification lost.
12. Fix StatelessWriter ACK check.
13. Fix `total_read_` to be consistent with Reader's History after *DataReader::get_first_untaken_info()*.
14. Fix doxygen documentation adding deprecated to `ThroughputControllerDescriptor`.
15. Several dependencies fixes upgrading to Ubuntu 22.04.
16. Ensure `shared_mutex` implementation is consistent throughout supported platforms.
17. Fix StatisticsSubmessageData unaligned access.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.16 Version 2.7.1 (EOL)

This release includes the following **features** in an ABI compatible way:

1. *Checking STATIC EDP XML Files* by means of *DomainParticipantFactory::check_xml_static_discovery()*.
2. *ReadCondition* implementation.

This release includes the following **improvements**:

1. Thread sanitizer CI.
2. Overload *get_unread_count()*.
3. Improve read/take performance when using topic with a great number of keys.
4. Improve rediscovery on lossy environments.
5. New *CMake option* *USE_THIRDPARTY_SHARED_MUTEX*.
6. Notify changes in bulk in RTPS readers.

This release includes the following **bugfixes**:

1. Fix Fast CDR submodule update to v1.0.24.
2. Fix access to some pointers.
3. Fixed validation on `ParameterPropertyList_t`.
4. Fixed acknowledgement in DataSharing.
5. Fixed wrong usage of `std::remove_if`.
6. Suppress OpenSSL 3.0 warnings.
7. Fixed race condition in Logging module.

8. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.17 Version 2.7.0 (EOL)

This release includes the following **improvements**:

1. Support for *DDS SampleRejectedStatus API*
2. Support for DDS DataWriter methods:
1. *DataWriter::write_w_timestamp()*
2. *DataWriter::register_instance_w_timestamp()*
3. *DataWriter::unregister_instance_w_timestamp()*
4. *DataWriter::dispose_w_timestamp()*
3. Support for DDS *find_topic()*
4. Support for GCC 12
5. Upgrade CMake minimum requirement to 3.16.3
6. Add Windows DLL support to Dynamic Types API

Some **bugfixes** are also included:

1. Deadlocks and data races
2. Move deprecated OpenSSL cleanup function to match the right version

---

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.18 Version 2.6.6

This release includes the following **improvements**:

1. Improve validation on PID_PROPERTY_LIST deserialization.
2. Improved CPU usage of timed events thread.
3. Improved performance on intraprocess + data-sharing.
4. Explicitly register type object in ContentFilteredTopicExample.
5. Improve installer generation with documentation version fallback.
6. Improve content filter expression parameters checks and verbosity.

This release includes the following **bugfixes**:

1. Fixed long-standing reconnection issues on SHM transport.
2. Correctly resolve alias in DDSQLFilter.
3. Fixed partition copy in QoS.

4. Added length checks to prevent nullptr memory copy calls.
5. Fixed XMLParser null-dereference when parsing log configuration.
6. Fixed SHM in 32-bit architectures.
7. Added missing include.
8. Avoid double definition of FASTDDS_ENFORCE_LOG_INFO.
9. Fixed statistics data_count with data-sharing.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.19 Version 2.6.5

This release includes the following **improvements**:

1. Improve behavior when `STRICT_REALTIME` *CMake option* is not enabled.
2. Using functors for `for_matched_readers` parameter.
3. Improve auto GAPs in Data Sharing.

This release includes the following **bugfixes**:

1. Fix RTPS StatelessWriter ACK check.
2. Fix `total_read_` to be consistent with Reader's History after *`DataReader::get_first_untaken_info()`*.
3. Add deprecation notice to `ThroughputControllerDescriptor`.
4. UBSan (Undefined Behavior Sanitizer) fixes.
5. Several dependencies fixes upgrading to Ubuntu 22.04.
6. Fix chain of trust issues with a single CA certificate.
7. Correctly handle builtin endpoints mask.
8. Take mutex when removing local reader in WLP.
9. Handle `SIGTERM` signal in Fast DDS CLI.
10. Fix data races in SecurityManager authentication process.
11. Avoid creating entities within callbacks in DynamicHelloWorldExample.
12. Remove Asio dependency from DeadlineQoSExample.
13. Validity check for first sequence number.
14. Include right header when building for iOS.
15. Fix build on MSVC 19.
16. Correctly assign multicast port to multicast initial peers.
17. Select correct listener for *`on_requested_deadline_missed()`*.
18. Forward compatibility with boost inter-process 1.74+.
19. Fix missing includes when building with GCC 13.
20. Honor `allow_unauthenticated_participants` flag.

21. Capture all Fast CDR exceptions.
22. Fix example to delete Topic after deleting the corresponding Endpoint.
23. Protect against uncaught exception in SHM segment creation.
24. Initial acknack backoff.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.20 Version 2.6.4

This release includes the following **bugfixes**:

1. Fix communication with asymmetric *Ignore Participant flags*.
2. Fix deadlock in Writer Liveliness Protocol when using intraprocess.
3. Fix notification lost.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.21 Version 2.6.3

This release includes the following **improvements**:

1. Upgrade CMake minimum requirement to 3.13.
2. Improve `Guid_t` operator < performance.

This release includes the following **bugfixes**:

1. Add python3 dependency to package.xml.
2. Fix complex member printing for DynamicDataHelper.
3. Fix selection of output locators.
4. Fix null references on XML parser.
5. Fix data races when creating DataWriters.
6. Send GAPs correctly when using separate sending.
7. Install Statistics IDL file.
8. Fixes for building in older compilers.
9. Fix deadlock when removing DomainParticipant when using SECURITY.
10. Ensure `shared_mutex` implementation is consistent throughout supported platforms.
11. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.22 Version 2.6.2

This release includes the following **improvements**:

1. Support for GCC 12.
2. Overload *DataReader::get_unread_count()*.
3. Improve read/take performance when using topic with a great number of keys.
4. Improve rediscovery on lossy environments.

This release includes the following **bugfixes**:

1. Fixed several deadlocks and data races.
2. Fixed validation on `ParameterPropertyList_t`.
3. Fixed wrong usage of `std::remove_if`.
4. Fixed acknowledgement in DataSharing.
5. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.23 Version 2.6.1

This release includes the following **improvements**:

1. Support for *writer side content filtering*
2. Support hexadecimal values on SQL filter
3. Support for *DataWriter::get_key_value()*
4. Support for *DataReader::lookup_instance()*
5. Support for SampleLostStatus on DataReader
6. Improved doxygen documentation

Some **bugfixes** are also included:

1. Fixed several lock order inversion issues
2. Fixed data race when closing UDP channels
3. Fixed empty partition validation checks
4. Fixed corner case with reliable writers and samples with a huge number of fragments
5. Other minor fixes and improvements

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.24 Version 2.6.0

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed. Some API is also being deprecated.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This minor release includes the following **features**:

1. *Allow modifying remote server locators at runtime*
2. *Add statistics physical information to DATA[p] using properties*
3. *Content filter discovery information RTPS API*
4. Endpoint discovery RTPS API
5. `on_sample_lost` RTPS API
6. Transport layer API extension
7. *XML support for Fast DDS CLI*
8. *New exchange format to reduce bandwidth in Static Discovery*

It also includes the following **improvements**:

1. Support lowercase keywords on SQL filter
2. Separate initialization and enabling of BuiltinProtocols
3. Add `disable_positive_acks` to Static Discovery XML
4. Several updates in the DDS-PIM API
5. Support for octet vectors on XML parser
6. Update README and roadmap
7. Update Fast-CDR submodule to v1.0.24
8. Add new CMake option `APPEND_PROJECT_NAME_TO_INCLUDEDIR`

Some **bugfixes** are also included:

1. Fix MatchedStatus `last_*_handle`
2. Fix recommended statistics DataReaderQos to enable backwards compatibility
3. Fixes for supporting Python bindings in Windows platforms
4. Fix publishing physical data on statistics topic
5. Other minor fixes and improvements

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.25 Version 2.5.2 (EOL)

This release includes the following **improvements**:

1. Support lowercase keywords and hexadecimal values on SQL filter.
2. Support for GCC 12.

This release includes the following **bugfixes**:

1. Fix MatchedStatus *last_*_handle*.
2. Fix recommended statistics DataReaderQos to enable backwards compatibility.
3. Fix deadlocks and data races.
4. Fix empty partition validation checks.
5. Fix corner case with reliable writers and samples with a huge number of fragments.
6. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.26 Version 2.5.1 (EOL)

This release includes the following **improvements**:

1. *ContentFilterTopic filtering at the DataReader side*.
2. Release lifecycle.

This release includes the following **bugfixes**:

1. XML parser fixes.
2. Discovery Server fixes.
3. Fix DataSharing sample validation.
4. PKCS#11 support fixes.
5. Test fixes.
6. Doxygen documentation fixes.
7. GAP message fix.
8. Enable memory protection on DataSharing readers.
9. TCP reconnection issues.
10. Fix dynamic network interfaces feature.
11. Several Security module fixes.
12. STRICT_REALTIME fix.
13. Suppress OpenSSL 3.0 warnings.
14. Move optionparser to thirdparty.
15. Thread-safe access to endpoints collections.

16. MemberDescriptor fully qualified name.
17. Setting QoS fix.
18. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.27 Version 2.5.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This minor release includes the following **features**:

1. *Support for PKCS#11* format URIs for private keys
2. Added interfaces for content filter APIs
3. Allow new *network interfaces to be detected at runtime*
4. New API on *DataWriter* to wait for a specific instance to be acknowledged
5. Added interfaces for *concatenation of transports*
6. Allow *XML profiles to be loaded from a string*
7. Allow *disabling piggyback heartbeat* on XML and DataWriter QoS
8. New basic configuration example

It also includes the following **improvements**:

1. Working implementation of instance_state and view_state
2. Allow zero-valued keys
3. Made some type aliases public to ease python bindings integration
4. Improved performance by avoiding unnecessary payload copies for samples that are going to be rejected
5. Removed unnecessary headers from Log module public headers
6. Add support for Key annotation in TypeObjectFactory
7. Only export public symbols on non-windows platforms
8. Some documentation improvements

Some important **bugfixes** are also included:

1. Fixed payload pool handling on EDPSimple destructor
2. Fixed null dereference on XML parser
3. Correctly export XTypes related methods on Windows

4. Ensure correct boost singleton destruction order
5. Avoid warning when environment file filename is empty
6. Correctly set GUID of DataWriter and DataReader upon creation

---

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.28 Version 2.4.2 (EOL)

This release includes the following **improvements**:

1. Enable memory protection on *DataSharing* readers
2. Add const overload of `DataReader::guid()`
3. Set recommended statistics *DataReaderQos* to `PREALLOCATED_WITH_REALLOC`
4. Allow fully qualified name on *MemberDescriptor*

This release includes the following bugfixes:

1. Fix and refactor EDPSimple destructor
2. Fix several build warnings on certain platforms
3. Fix OSS fuzz issues
4. Fix *TCP* synchronization issues
5. Correct reporting of *MatchedStatus* `last_*_handle`
6. Ensure correct boost singleton destruction order
7. Fix inserting minimum CacheChange_t in GAP message

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.29 Version 2.4.1 (EOL)

This release includes the following **improvements**:

1. Fixed several flaky tests
2. Improved bandwidth usage of GAPs and HEARTBEATs
3. Correctly implement delete_contained_entities
4. Use native inter-process on Windows
5. Improved performance of unregister_instance
6. Improved OSS-fuzz integration
7. Support for partitions on DataWriterQoS and DataReaderQoS
8. Some documentation improvements
9. Removed unused macro to avoid naming clashes

This release includes the following **bugfixes**:

1. Avoid bad_node_size exception when cross building
2. Fixed build on old compilers
3. Fixed buffers exhaustion when compiled with statistics
4. Fixed runtime addition of Discovery Servers
5. Fixed dangling sample references with big data
6. Fixed history record issues with persistence
7. Correctly disable DataReader on destruction
8. Fixed alignment issues on XTypes QoS policies serialization
9. Fixed reconnection to Discovery Server
10. Correctly use builtin publisher for statistics DataWriters
11. Fixed various GCC-11 warnings
12. Use only public APIs from foonathan::memory
13. Fixed installation directories for DDS examples
14. Fixed read after free on security code

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.30 Version 2.4.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This minor release includes the following **features**:

- *Conditions and Wait-sets* implementation.
- *Flow controllers*.
- *Configure Discovery Server locators using names*.
- *Modifying remote servers list at run time*.
- *Environment file override*.

It also includes the following **improvements**:

- Allow setting custom folder for *data-sharing* files.
- Allow setting persistence guid with static discovery.
- Check for NDEBUG in logInfo.
- Removed old unused CMake code.

- Fixed TLS behavior on TCP example.
- Prepare API for easy integration of python bindings.
- Improved statistics performance.

Some important **bugfixes** are also included:

- Fixed order of returned samples on topics with keys.
- Allow updating partitions to an empty set.
- Correctly propagate DomainParticipantQos updates.
- Avoid a volatile data-sharing reader to block a writer.
- Correctly give priority to intra-process over data-sharing.
- Fixed reallocation issue on LivelinessManager.
- Fixed deadline issue on volatile DataWriter
- Fixed STRICT_REALTIME silently not active with Unix POSIX systems.
- Fixed build errors with OpenSSL 3.0

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.31 Version 2.3.6 (EOL)

This release includes the following **improvements**:

1. Improve rediscovery on lossy environments.
2. Upgrade CMake minimum requirement to 3.13.
3. Improve `Guid_t` operator < performance.

This release includes the following **bugfixes**:

1. Fixed validation on `ParameterPropertyList_t`.
2. Add python3 dependency to package.xml.
3. Fix null references and uncaught exceptions on XML parser.
4. Install Statistics IDL file.
5. Fix data races when creating DataWriters.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.32 Version 2.3.5 (EOL)

This release includes the following **improvements**:

1. Fixed several flaky tests.
2. Use native inter-process on Windows.
3. Support for partitions on DataWriterQoS and DataReaderQoS.
4. Support for GCC 12.
5. Correctly implement delete_contained_entities.

This release also includes the following **bugfixes**:

1. Fixed deadline issue on volatile DataWriter.
2. Allow updating partitions to an empty set.
3. Fixed order of returned samples on topics with keys.
4. Fixed issues in LivelinessManager.
5. Correctly give priority to intra-process over data-sharing.
6. Avoid bad_node_size exception when cross-building.
7. Fixed build errors with OpenSSL 3.0.
8. Avoid a volatile data-sharing reader to block a writer.
9. Fixed history record issues with persistence.
10. Correctly disable DataReader on destruction.
11. Fixed various GCC 11 warnings.
12. Fixed payload pool handling on EDPSimple destructor.
13. Fixed read after free on security code.
14. Fixed null dereference on XML parser.
15. Ensure correct boost singleton destruction order.
16. Enable memory protection on DataSharing readers.
17. TCP reconnection issues.
18. MemberDescriptor fully qualified name.
19. Fix recommended statistics DataReaderQos to enable backwards compatibility.
20. Fixed dangling sample references with big data.
21. Fixed deadlocks and data races.
22. Fixed reconnection to Discovery Server.
23. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.33 Version 2.3.4 (EOL)

This release includes the following **improvements**:

1. Support of googletest using colcon
2. Network latency reports source participant
3. Update Fast DDS Gen to v2.0.2

This release includes the following **bugfixes**:

1. Fix mutex lock count on `PDPListener`
2. Limit `SequenceNumberSet` number of bits on deserialization
3. Fix segmentation fault on discovery server
4. Fix deadlock with security and timers
5. Fix bug using not protected code in a test
6. Fix deadlock with `LivelinessManager`
7. Fix interval loop on events
8. Fix run event when was cancelled
9. Validate sequence range on `CDRMessage::readSequenceNumberSet`
10. Fix subscription throughput data generation
11. Allow examples to build on QNX
12. Fix code on SHM clean
13. Accept Statistics DataWriters in Discovery Server
14. Fix read/take behavior when a future change is found
15. Correctly handle deserialization errors on `read_next_sample()` / `take_next_sample()`
16. Fixing `SequenceNumberSet_t` deserialization
17. Proper history clean up when a reader unmatches a writer
18. Unprotected code loaning samples
19. Fix publication throughput statistic on volatile writers
20. Fix Fast DDS CLI server name
21. Several fixes in examples and tests

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.34 Version 2.3.3 (EOL)

This release includes the following **improvements**:

- Added more durability kinds in Static Discovery xml parser
- Explicitly enable/disable data-sharing on performance tests
- Allow fully qualified name in TypeDescriptor
- Added missing DynamicData::get_union_id() method
- Change log severity in DiscoveryServer first announcement
- Several corrections to README

This release includes the following **bugfixes**:

- Fixed warnings and segfaults on 32-bit platforms
- Fixed UDPv6 behavior
- Fixed persistence guid issue on statistics writers
- Fixed static linking with open SSL
- Fixed statistics header file inclusion
- Fixed build on RedHat systems

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.35 Version 2.3.2 (EOL)

This release includes the following **feature**:

- Statistics Module

It also includes the following **improvements**:

- Update Asio submodule and avoid exporting Asio API
- Improve Windows installers
- Ease Google Fuzz integration
- Improve Doxygen documentation on lifetime of pointers created with `RTPSDomain`
- Update Fast CDR to v1.0.21

This release includes the following **bugfixes**:

- Add a correct multicast address for UDPv6
- Recover from out-of-sync TCP datagrams

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.36 Version 2.3.1 (EOL)

This release includes several **bugfixes** and **improvements**:

- Added *Fast DDS Statistics Module* implementation
- Fixed alignment issues on generated code calculation of maximum serialized size
- Fixed calculation of data-sharing domain id
- Fixed issues on data-sharing with volatile writers
- Fixed build issues on old compilers
- Fixed some tests when the library is built without security
- Fixed and exposed pull mode on writers
- Fixed handling of *–data_sharing* on latency test
- Fixed calculation of memory pools sizes on debug builds
- Correctly update memory policy on writers and readers

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.37 Version 2.3.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This release adds the following **features**:

- *Unique network flows*
- *Discovery super-client*
- *Statistics module API*
- *New flow controller API*
- *Static discovery configuration from raw string*
- *Added reception timestamp to SampleInfo*
- *Exposing get_unread_count on DataReader*

It also includes the following **improvements**:

- Data-sharing delivery internal refactor
- Additional metadata on persistence databases
- Refactor on ReturnCode_t to make it switch friendly

- Performance tests refactored to use DDS-PIM high-level API
- Receive const pointers on delete_xxx methods
- Discovery server improvements
- Made SOVERSION follow major.minor

Some important **bugfixes** are also included:

- Fixed shared memory usage on QNX
- Fixed reference counting on internal pools
- Fixed singleton destruction order
- Fixed interoperability issues with x-types information
- Fixed recovery of shared memory buffers
- Lifespan support in persistent writers

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.38 Version 2.2.1 (EOL)

This release includes the following **improvements**:

1. Data-sharing delivery internal refactor.
2. Performance tests refactored to use DDS-PIM high-level API.
3. Discovery server improvements.

This release includes the following **bugfixes**:

1. Fixed reference counting on internal pools.
2. Fixed singleton destruction order.
3. Fixed default multicast locators.
4. Fixed interoperability issues with x-types information.
5. Fixed Reader history issues.
6. Fixed data races issues.
7. Fixed shared memory issues.
8. Fixed heartbeat and ACK issues.
9. Fixed LivelinessManager issues.
10. Fixed TCP reception synchronization.
11. Fixed build issues on old compilers.
12. Allow modifying Partition QoS in enabled entities.
13. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.39 Version 2.2.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This release adds the following **features**:

- Data Sharing delivery (avoids transport encapsulation for localhost communications)
- Complete DDS-PIM high-level API declarations
- Extension APIs allowing zero-copy delivery (both intra-process and inter-process)
- Upgrade to Quality Level 1

It also includes the following **improvements**:

- Code coverage policy
- Added several tests to increase coverage
- Increased GUID uniqueness
- Allow logInfo messages to be compiled on build types other than debug

Some important **bugfixes** are also included:

- Fixed timed events manager race condition
- Fixed payload protection issues with SHM transport
- Writers correctly handle infinite resource limits on keyed topics
- Fixed unsafe code on AESGCMGMAC plugin
- Several fixes for IPv6 (whitelists, address parser)
- Fixes on liveliness timing handling
- Fixed warnings building on C++20

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

6.42.40 Version 2.1.4 (EOL)

This release includes the following **improvements**:

1. Improve auto GAPs in Data Sharing
2. Improve behavior when `STRICT_REALTIME` *CMake option* is not enabled
3. Handle SIGTERM in fast discovery server CLI

This release includes the following **bugfixes**:

1. Select correct listener for on_requested_deadline_missed
2. Correctly assign multicast port to multicast initial peers
3. Fix chain of trust issues with a single CA certificate
4. Correctly handle builtin endpoints mask
5. Fix build on MSVC 19

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.41 Version 2.1.3 (EOL)

This release includes the following **improvements**:

1. Improve rediscovery on lossy environments.

This release includes the following **bugfixes**:

1. Fixed validation on `ParameterPropertyList_t`.
2. Fixed `SequenceNumberSet_t` deserialization.
3. Add python3 dependency to package.xml.
4. Fix data races when creating DataWriters.
5. Fix deadlock when removing remote DomainParticipants by expired liveliness when using Security.
6. Fix communication with asymmetric *Ignore Participant flags*.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.42 Version 2.1.2 (EOL)

This release includes the following **improvements**:

1. Allow fully qualified name in TypeDescriptor.
2. Use native inter-process on Windows.
3. Support for GCC 12.
4. Support of googletest using colcon.

This release also includes the following **bugfixes**:

1. Fixed recovery of shared memory buffers.
2. Fixed issues in LivelinessManager.
3. Fixed default multicast locators.
4. Fixed TCP issues.
5. Fixed deadlocks and data races.
6. Fixed deadline issue on volatile DataWriter.
7. Avoid bad_node_size exception when cross-building.
8. Fixed order of returned samples on topics with keys.
9. Allow updating partitions to an empty set.
10. Suppress OpenSSL 3.0 warnings.
11. MemberDescriptor fully qualified name.
12. Fixed history record issues with persistence.
13. Fixed reconnection to Discovery Server.
14. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.43 Version 2.1.1 (EOL)

This release includes the following **bugfixes**:

- Fixed race condition on security handshake
- Fixed SHM data corruption when using both payload and sub-message protection
- Fixed some interoperability issues
- Fixed race condition on timed-events thread
- Fixed usage of SHM on QNX systems

It also includes the following **improvements**:

- Increased uniqueness of GUID prefix
- Discovery server improvements

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.44 Version 2.1.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

Users of the RTPS low-level API should also be aware of the following **API deprecations**:

- History::reserve_Cache has been deprecated
  - Methods RTPSWriter::new_change or RTPSReader::reserveCache should be used instead
- History::release_Cache has been deprecated
  - Methods RTPSWriter::release_change or RTPSReader::releaseCache should be used instead

This release adds the following **features**:

- Support persistence for large data
- Added support for *on_requested_incompatible_qos* and *on_offered_incompatible_qos*
- SKIP_DEFAULT_XML environment variable
- Added FORCE value to THIRDPARTY cmake options
- New log consumer (StdOutErrConsumer)
- Added methods to get qos defined in XML Profile
- Support for persistence on TRANSIENT_LOCAL

It also includes the following **improvements**:

- Internal refactor for intra-process performance boost
- Allow usage of foonathan/memory library built without debug tool
- Large data support on performance tests
- Reduced flakiness of several tests

Some important **bugfixes** are also included:

- Fixed behavior of several DDS API methods
- Fixed interoperability issues with RTI connext
- Fixed DLL export of some methods
- Avoid redefinition of compiler defined macros
- Fixed some intra-process related segmentation faults and deadlocks
- Fixed large data payload protection issues on intra-process
- Fixed C++17 and VS 2019 warnings
- Fixed linker problems on some platforms
- Fixed transient local retransmission after participant drop
- Fixed assertion failure on persistent writers

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.45 Version 2.0.3 (EOL)

It also includes the following **improvements**:

1. Increased uniqueness of GUID prefix (#1648)
2. Upgrade Fast CDR to v1.0.20 (#1793)

This release includes the following **bugfixes**:

1. Fixed some race conditions (#1540, #2023)
2. Fixed SHM issues (#1644, #1895, #2266)
3. Fixed some interoperability issues (#1624, #1752, #1849)
4. Fixed Discovery Server 2.0 issues (#1639, #1651, #1761, #1796)
5. Fixed several issues on QNX systems (#1744, #1773, #1776)
6. Fix singleton destruction order (#1758)
7. Fix heartbeat and ACK issues (#1865)
8. Fix issues in LivelinessManager (#1872, #2147)
9. Fix multicast issues (#1966, #1905)
10. Fix TCP reception synchronization (#1981)
11. XTypes standard compliance and fixes (#2006, #2278)
12. Other minor fixes (#1558, #1734, #1814, #1935, #1978, #2121)

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.46 Version 2.0.2 (EOL)

This release includes the following improvements:

- Improve QNX support
- Security improvements
- Fast DDS Quality Declaration (QL 2)
- Large traffic reduction when using Discovery Server (up to 85-90% for large deployments)
- Configuration of Clients of Discovery Server using an environment variable
- A CLI for Fast DDS:
  - This can be used to launch a discovery server
  - Clean SHM directories with one command
- Shared memory transport enabled by default

- Solved edge-case interoperability issue with CycloneDDS
- Add package.xml

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.47 Version 2.0.1 (EOL)

This release includes the following bug fixes:

- Fixed sending GAPs to late joiners
- Fixed asserting liveliness on data reception
- Avoid calling `OpenSSL_add_all_algorithms()` when not required

Other improvements:

- Fixing warnings

PRs in merge order: #1295, #1300, #1304, #1290, #1307.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.48 Version 2.0.0 (EOL)

This release has the following **API breaks**:

- eClock API, which was deprecated on v1.9.1, has been removed
- *eprosima::fastrtps::rtps::RTPSDomain::createParticipant* methods now have an additional first argument *domain_id*
- Data member *domainId* has been removed from *eprosima::fastrtps::rtps::RTPSParticipantAttributes* and added to *eprosima::fastrtps::ParticipantAttributes*

Users should also be aware of the following **deprecation announcement**:

- All classes inside the namespace *eprosima::fastrtps* should be considered deprecated. Equivalent functionality is offered through namespace *eprosima::fastdds*.
- Namespaces beneath *eprosima::fastrtps* are not included in this deprecation, i.e. *eprosima::fastrtps::rtps* can still be used)

This release adds the following **features**:

- Added support for register/unregister/dispose instance
- Added DDS compliant API. This new API exposes all the functionality of the Publisher-Subscriber Fast RTPS API adhering to the Data Distribution Service (DDS) version 1.4 specification
- Added Security Logging Plugin (contributed by Cannonical Ltd.)
- Bump to FastCDR v1.0.14

It also includes the following bug fixes and improvements:

- Support for OpenSSL 1.1.1d and higher
- Support for latest versions of gtest
- Support for FreeBSD
- Fault tolerance improvements to Shared Memory transport
- Fixed segfault when no network interfaces are detected
- Correctly ignoring length of *PID_SENTINEL* on parameter list
- Improved traffic on PDP simple mode
- Reduced CPU and memory usage

6.42.49 Version 1.10.1 (EOL)

This release includes the following **improvements**:

1. Add new CMake option: SHM_TRANSPORT_DEFAULT. Shared Memory (SHM) Transport disabled by default.
2. Comply with the RTPS standard concerning PID_SENTINEL.
3. Support for OpenSSL 1.1.1d.

This release includes the following bugfixes:

1. Fix crash when there are no network interfaces.
2. Several Shared Memory Transport fixes.
3. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.50 Version 1.10.0 (EOL)

This release adds the following features:

- New built-in *Shared Memory Transport*
- Transport API refactored to support locator iterators
- Added subscriber API to retrieve info of first non-taken sample
- Added parameters to fully avoid dynamic allocations
- History of built-in endpoints can be configured
- Bump to FastCDR v1.0.13.
- Bump to Fast-RTPS-Gen v1.0.4.
- Require CMake 3.5 but use policies from 3.13

It also includes the following bug fixes and improvements:

- Fixed alignment on parameter lists
- Fixed error sending more than 256 fragments.
- Fix handling of STRICT_REALTIME.

- Fixed submessage_size calculation on last data_frag.
- Solved an issue when recreating a publishing participant with the same GUID.
- Solved an issue where a publisher could block on write for a long time when a new subscriber (late joiner) is matched, if the publisher had already sent a large number of messages.
- Correctly handling the case where lifespan expires at the same time on several samples.
- Solved some issues regarding liveliness on writers with no readers.
- Correctly removing changes from histories on keyed topics.
- Not reusing cache change when sample does not fit.
- Fixed custom wait_until methods when time is in the past.
- Several data races and ABBA locks fixed.
- Reduced CPU and memory usage.
- Reduced flakiness of liveliness tests.
- Allow for more use cases on performance tests.

Several bug fixes on discovery server:

- Fixed local host communications.
- Correctly trimming server history.
- Fixed backup server operation.
- Fixed timing issues.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.42.51 Version 1.9.5 (EOL)

This release includes the following **improvements**:

1. Propagate serialization error when reading samples from Subscriber History.
2. Improvements in test suite.
3. Improvements to reduce memory consumption.
4. Update CMake (3.5) using newer policies.
5. Improve PDP StatefulWriter announcement.
6. Performance improvements.
7. Message receiver improvements.
8. QoS Policies improvements.

This release includes the following bugfixes:

1. Fix compiler warnings in Windows when building the test suite.
2. Fix several data races.
3. Fix History issues.
4. Fix errors when sending data fragments.
5. Fix strict real-time behavior.

6. Fix in Discovery Server.
7. Fix CMake option.
8. Fix interoperability issues.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.52 Version 1.9.4 (EOL)

This release adds the following features:

- Intra-process delivery mechanism is now active by default.
- Synchronous writers are now allowed to send fragments.
- New memory mode DYNAMIC_RESERVE on history pool.
- Performance tests can now be run on Windows and Mac.
- XML profiles for requester and replier.

It also includes the following bug fixes and improvements:

- Bump to FastCDR v1.0.12.
- Bump to Fast-RTPS-Gen v1.0.3.
- Fixed deadlock between PDP and StatefulReader.
- Improved CPU usage and allocations on timed events management.
- Performance improvements on reliable writers.
- Fixing bugs when Intra-process delivery is activated.
- Reducing dynamic allocations and memory footprint.
- Improvements and fixes on performance tests.
- Other minor bug fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.53 Version 1.9.3 (EOL)

This release adds the following features:

- Participant discovery filtering flags.
- Intra-process delivery mechanism opt-in.

It also includes the following bug fixes and improvements:

- Bump to Fast-RTPS-Gen v1.0.2.
- Bring back compatibility with XTypes 1.1 on PID_TYPE_CONSISTENCY.
- Ensure correct alignment when reading a parameter list.
- Add CHECK_DOCUMENTATION *cmake* option.
- EntityId_t and GuidPrefix_t have now their own header files.

- Fix potential race conditions and deadlocks.
- Improve the case where *check_acked_status* is called between reader matching process and its acknack reception.
- RTPSMessageGroup_t instances now use the thread-local storage.
- FragmentedChangePitStop manager removed.
- Remove the data fragments vector on CacheChange_t.
- Only call find_package for TinyXML2 if third-party options are off
- Allow XMLProfileManager methods to not show error log messages if a profile is not found.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.54 Version 1.9.2 (EOL)

This release includes the following feature:

- Multiple initial PDP announcements.
- Flag to avoid builtin multicast.

It also adds the following bug fixes and improvements:

- Bump to Fast-RTPS-Gen v1.0.1.
- Bump to IDL-Parser v1.0.1.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.55 Version 1.9.1 (EOL)

This release includes the following features:

- Fast-RTPS-Gen is now an independent project.
- Header **eClock.h** is now marked as deprecated.

It also adds the following bug fixes and improvements:

- Bump to FastCDR v1.0.11.
- Installation from sources documentation fixed.
- Fixed assertion on WriterProxy.
- Fixed potential fall through while parsing Parameters.
- Removed deprecated guards causing compilation errors in some 32 bits platforms.
- *addTOCDRMessage* method is now exported in the DLL, fixing issues related with Parameters' constructors.
- Improve windows performance by avoiding usage of *_Cnd_timedwait* method.
- Fixed reported communication issues by sending multicast through *localhost* too.
- Fixed potential race conditions and deadlocks.
- Eliminating use of *acceptMsgDirectTo*.
- Discovery Server framework reconnect/recreate strategy.

- Removed unused folders.
- Restored subscriber API.
- SequenceNumber_t improvements.
- Added STRICT_REALTIME *cmake* option.
- SubscriberHistory improvements.
- Assertion of participant liveliness by receiving RTPS messages from the remote participant.
- Fixed error while setting next deadline event in *create_new_change_with_params*.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.56 Version 1.9.0 (EOL)

This release includes the following features:

- Partial implementation of allocation QoS.
- Implementation of Discovery Server.
- Implementation of non-blocking calls.

It also adds the following bug fixes and improvements:

- Added sliding window to BitmapRange.
- Modified default behavior for unknown writers.
- A *Flush()* method was added to the logger to ensure all info is logged.
- A test for loading *Duration_t* from XML was added.
- Optimized WLP when removing local writers.
- Some liveliness tests were updated so that they are more stable on Windows.
- A fix was added to *CMakeLists.txt* for installing static libraries.
- A fix was added to performance tests so that they can run on the RT kernel.
- Fix for race condition on built-in protocols creation.
- Fix for setting *nullptr* in a *fixed_string*.
- Fix for v1.8.1 not building with -DBUILD_JAVA=ON.
- Fix for GAP messages not being sent in some cases.
- Fix for coverity report.
- Several memory issues fixes.
- *fastrtps.repos* file was updated.
- Documentation for building with Colcon was added.
- Change CMake configuration directory if INSTALLER_PLATFORM is set.
- IDL sub-module updated to current version.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.57 Version 1.8.5 (EOL)

This release includes the following **bugfixes**:

1. Fix Subscriber History to correctly notify late-joiners in case of KEEP_LAST, RELIABLE, and TRANSIENT_LOCAL.
2. Fix Writer History behavior when there are no matched readers.
3. Fix heartbeat and ACK issues.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.58 Version 1.8.4 (EOL)

This release adds the following **feature**:

- XML profiles for *requester* and *replier*

It also has the following **important bug fixes**:

- Solved an issue when recreating a publishing participant with the same GUID (either on purpose or by chance)
- Solved an issue where a publisher could block on *write* for a long time when, after a large number of samples have been sent, a new subscriber is matched.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*

6.42.59 Version 1.8.3 (EOL)

This release adds the following bug fixes and improvements:

- Fix serialization of TypeConsistencyEnforcementQosPolicy.
- Bump to Fast-RTPS-Gen v1.0.2.
- Bump to IDL-Parser v1.0.1.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*

6.42.60 Version 1.8.2 (EOL)

This release includes the following features:

- Modified unknown writers default behavior.
- Multiple initial PDP announcements.
- Flag to avoid builtin multicast.
- *STRICT_REALTIME* compilation flag.

It also adds the following bug fixes and improvements:

- Fix for setting *nullptr* in a fixed string.
- Fix for not sending GAP in several cases.
- Solve *Coverity* report issues.

- Fix issue of *fastddsgen* failing to open *IDL.g4* file.
- Fix unnamed lock in *AESGCMGMAC_KeyFactory.cpp*.
- Improve *XMLProfiles* example.
- Multicast is now sent through *localhost* too.
- *BitmapRange* now implements sliding window.
- Improve *SequenceNumber_t* struct.
- Participant's liveliness is now asserted when receiving any RTPS message.
- Fix leak on RemoteParticipantLeaseDuration.
- Modified default values to improve behavior in *Wi-Fi* scenarios.
- *SubscriberHistory* improvements.
- Removed use of *acceptMsgDirectTo*.
- *WLP* improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*

6.42.61 Version 1.8.1 (EOL)

This release includes the following features:

- Implementation of *LivelinessQosPolicy* QoS.

It also adds the following bug fixes and improvements:

- Fix for get_change on history, which was causing issues during discovery.
- Fix for announcement of participant state, which was sending ParticipantBuiltinData twice.
- Fix for closing multicast UDP channel.
- Fix for race conditions in SubscriberHistory, UDPTransportInterface and StatefulReader.
- Fix for lroundl error on Windows in Time_t.
- CDR & IDL submodules update.
- Use of java 1.8 or greater for fastddsgen.jar generation.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.62 Version 1.8.0 (EOL)

This release included the following features:

- Implementation of IDL 4.2.
- Implementation of *DeadlineQosPolicy* QoS.
- Implementation of *LifespanQosPolicy* QoS.
- Implementation of *DisablePositiveACKsQosPolicy* QoS.
- Secure sockets on TCP transport (*TLS over TCP*).

It also adds the following improvements and bug fixes:

- Real-time improvements: non-blocking write calls for best-effort writers, addition of fixed size strings, fixed size bitmaps, resource limited vectors, etc.
- Duration parameters now use nanoseconds.
- Configuration of participant mutation tries.
- Automatic calculation of the port when a value of 0 is received on the endpoint custom locators.
- Non-local addresses are now filtered from whitelists.
- Optimization of check for acked status for stateful writers.
- Linked libs are now not exposed when the target is a shared lib.
- Limitation on the domain ID has been added.
- UDP non-blocking send is now optional and configurable via XML.
- Fix for non-deterministic tests.
- Fix for ReaderProxy history being reloaded incorrectly in some cases.
- Fix for RTPS domain hostid being potentially not unique.
- Fix for participants with different lease expiration times failing to reconnect.

Known issues

- When using TPC transport, sometimes callbacks are not invoked when removing a participant due to a bug in ASIO.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.63 Version 1.7.3 (EOL)

This release includes the following **bugfixes**:

1. Remove inline specifier from public method not defined in header file.
2. Fix FastRTPS-Gen version generation

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.64 Version 1.7.2 (EOL)

This release fixes an important bug:

- Allocation limits on subscribers with a KEEP_LAST QoS was taken from resource limits configuration and didn't take history depth into account.

It also has the following improvements:

- Vendor FindThreads.cmake from CMake 3.14 release candidate to help with sanitizers.
- Fixed format of gradle file.

Some other minor bugs and performance improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.65 Version 1.7.1 (EOL)

This release included the following features:

- LogFileConsumer added to the logging system.
- Handle FASTRTPS_DEFAULT_PROFILES_FILE environment variable indicating the default profiles XML file.
- XML parser made more restrictive and with better error messages.

It also fixes some important bugs: * Fixed discovery issues related to the selected network interfaces on Windows. * Improved discovery times. * Workaround ASIO issue with multicast on QNX systems. * Improved TCP transport performance. * Improved handling of key-only data submessages.

Some other minor bugs and performance improvements.

KNOWN ISSUES

- Allocation limits on subscribers with a KEEP_LAST QoS is taken from resource limits configuration and doesn't take history depth into account.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.66 Version 1.7.0 (EOL)

This release included the following features:

- *TCP Transport*.
- *Dynamic Topic Types*.
- Security 1.1 compliance.

Also bug fixing, allocation and performance improvements.

Note: If you are upgrading from an older version, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

6.42.67 Version 1.6.0 (EOL)

This release included the following features:

- Persistence.
- Security access control plugin API and builtin *Access control plugin: DDS:Access:Permissions* plugin.

Also bug fixing.

Note: If you are upgrading from an older version than 1.4.0, it is advisable to regenerate generated source from IDL files using *fastddsgen*.

6.42.68 Version 1.5.0 (EOL)

This release included the following features:

- Configuration of Fast RTPS entities through XML profiles.
- Added heartbeat piggyback support.

Also bug fixing.

Note: If you are upgrading from an older version than 1.4.0, it is advisable to regenerate generated source from IDL files using *fastddsgen*.

6.42.69 Version 1.4.0 (EOL)

This release included the following:

- Added secure communications.
- Removed all Boost dependencies. Fast RTPS is not using Boost libraries anymore.
- Added compatibility with Android.
- Bug fixing.

Note: After upgrading to this release, it is advisable to regenerate generated source from IDL files using *fastddsgen*.

6.42.70 Version 1.3.1 (EOL)

This release included the following:

- New examples that illustrate how to tweak Fast RTPS towards different applications.
- Improved support for embedded Linux.
- Bug fixing.

6.42.71 Version 1.3.0 (EOL)

This release introduced several new features:

- Unbound Arrays support: Now you can send variable size data arrays.
- Extended Fragmentation Configuration: It allows you to setup a Message/Fragment max size different to the standard 64Kb limit.
- Improved logging system: Get even more introspection about the status of your communications system.
- Static Discovery: Use XML to map your network and keep discovery traffic to a minimum.
- Stability and performance improvements: A new iteration of our built-in performance tests will make benchmarking easier for you.
- ReadTheDocs Support: We improved our documentation format and now our installation and user manuals are available online on ReadTheDocs.

6.42.72 Version 1.2.0 (EOL)

This release introduced two important new features:

- Flow Controllers: A mechanism to control how you use the available bandwidth avoiding data bursts. The controllers allow you to specify the maximum amount of data to be sent in a specific period of time. This is very useful when you are sending large messages requiring fragmentation.
- Discovery Listeners: Now the user can subscribe to the discovery information to know the entities present in the network (Topics, Publishers & Subscribers) dynamically without prior knowledge of the system. This enables the creation of generic tools to inspect your system.

But there is more:

- Full ROS 2 Support: Fast RTPS is used by ROS 2, the upcoming release of the Robot Operating System (ROS).
- Better documentation: More content and examples.
- Improved performance.
- Bug fixing.

INDEX

A

absolute_generation_rank (*fastdds.SampleInfo property*), 974
access_scope (*fastdds.PresentationQosPolicy property*), 876
add_domain_id() (*fastdds.DataSharingQosPolicy method*), 864
alive_count (*fastdds.LivelinessChangedStatus property*), 886
alive_count_change (*fastdds.LivelinessChangedStatus property*), 886
ALIVE_INSTANCE_STATE (*class in fastdds*), 973
all() (*fastdds.StatusMask static method*), 889
allocated_samples (*fastdds.ResourceLimitsQosPolicy property*), 880
allocation() (*fastdds.DomainParticipantQos method*), 915
ALLOW_TYPE_COERCION (*class in fastdds*), 884
announcement_period (*fastdds.LivelinessQosPolicy property*), 872
assert_liveliness() (*fastdds.DataWriter method*), 920
assert_liveliness() (*fastdds.DomainParticipant method*), 893
assigned() (*fastdds.TypeInformation method*), 997
ASYNCHRONOUS_PUBLISH_MODE (*class in fastdds*), 877
AUTO (*class in fastdds*), 866
auto_fill_type_information() (*fastdds.TopicDataType method*), 985
auto_fill_type_object() (*fastdds.TopicDataType method*), 985
autodispose_unregistered_instances (*fastdds.WriterDataLifecycleQosPolicy property*), 885
autoenable_created_entities (*fastdds.EntityFactoryQosPolicy property*), 869
automatic() (*fastdds.DataSharingQosPolicy method*), 864
AUTOMATIC_LIVELINESS_QOS (*class in fastdds*), 873
autopurge_disposed_samples_delay (*fastdds.ReaderDataLifecycleQosPolicy property*), 879
autopurge_no_writer_samples_delay (*fastdds.ReaderDataLifecycleQosPolicy property*), 879

B

BaseStatus (*class in fastdds*), 885
begin() (*fastdds.PartitionQosPolicy method*), 874
begin_access() (*fastdds.Subscriber method*), 975
begin_coherent_changes() (*fastdds.Publisher method*), 939
BEST_EFFORT_RELIABILITY_QOS (*class in fastdds*), 880
BIT (*C macro*), 782
BIT0 (*C macro*), 781
BIT1 (*C macro*), 781
BIT2 (*C macro*), 781
BIT3 (*C macro*), 781
BIT4 (*C macro*), 781
BIT5 (*C macro*), 781
BIT6 (*C macro*), 781
BIT7 (*C macro*), 781
buffer() (*fastdds.LoanableCollection method*), 891
builtin (*fastdds.WireProtocolConfigQos property*), 884
builtin_controllers_sender_thread() (*fastdds.DomainParticipantQos method*), 915
BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_READER (*C macro*), 800
BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_WRITER (*C macro*), 800
BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_SECURE_DATA_READER (*C macro*), 800
BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_SECURE_DATA_WRITER (*C macro*), 800
BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REPLY_DATA_READER (*C macro*), 800
BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REPLY_DATA_WRITER (*C macro*), 800
BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REQUEST_DATA_READER (*C macro*), 800
BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REQUEST_DATA_WRITER (*C macro*), 800
BUILTIN_PARTICIPANT_DATA_MAX_SIZE (*C macro*), 800

builtin_transports_reception_threads_ (*fastdds.TransportConfigQos property*), 882
BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS (*class in fastdds*), 867
BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS (*class in fastdds*), 867

C

C_B_BLUE (*C macro*), 851
C_B_CYAN (*C macro*), 851
C_B_GREEN (*C macro*), 851
C_B_MAGENTA (*C macro*), 851
C_B_RED (*C macro*), 851
C_B_WHITE (*C macro*), 852
C_B_YELLOW (*C macro*), 852
C_BLUE (*C macro*), 851
C_BRIGHT (*C macro*), 851
C_CYAN (*C macro*), 852
C_DEF (*C macro*), 852
C_GREEN (*C macro*), 852
C_MAGENTA (*C macro*), 852
C_RED (*C macro*), 852
C_WHITE (*C macro*), 852
C_YELLOW (*C macro*), 853
CDR_BE (*C macro*), 773
CDR_LE (*C macro*), 773
check_xml_static_discovery() (*fastdds.DomainParticipantFactory method*), 907
clear() (*fastdds.DataRepresentationQosPolicy method*), 863
clear() (*fastdds.DataSharingQosPolicy method*), 864
clear() (*fastdds.DeadlineQosPolicy method*), 866
clear() (*fastdds.DestinationOrderQosPolicy method*), 867
clear() (*fastdds.DisablePositiveACKsQosPolicy method*), 867
clear() (*fastdds.DurabilityQosPolicy method*), 868
clear() (*fastdds.DurabilityServiceQosPolicy method*), 868
clear() (*fastdds.EntityFactoryQosPolicy method*), 869
clear() (*fastdds.GenericDataQosPolicy method*), 870
clear() (*fastdds.HistoryQosPolicy method*), 871
clear() (*fastdds.LatencyBudgetQosPolicy method*), 872
clear() (*fastdds.LifespanQosPolicy method*), 872
clear() (*fastdds.LivelinessQosPolicy method*), 873
clear() (*fastdds.OwnershipQosPolicy method*), 873
clear() (*fastdds.OwnershipStrengthQosPolicy method*), 874
clear() (*fastdds.PartitionQosPolicy method*), 875
clear() (*fastdds.PresentationQosPolicy method*), 876
clear() (*fastdds.PublishModeQosPolicy method*), 877
clear() (*fastdds.QosPolicy method*), 877
clear() (*fastdds.ReaderDataLifecycleQosPolicy method*), 879
clear() (*fastdds.ReaderResourceLimitsQos method*), 974
clear() (*fastdds.ReliabilityQosPolicy method*), 879
clear() (*fastdds.ResourceLimitsQosPolicy method*), 880
clear() (*fastdds.RTPSReliableReaderQos method*), 974
clear() (*fastdds.TimeBasedFilterQosPolicy method*), 881
clear() (*fastdds.TransportConfigQos method*), 882
clear() (*fastdds.TransportPriorityQosPolicy method*), 883
clear() (*fastdds.TypeConsistencyEnforcementQosPolicy method*), 883
clear() (*fastdds.TypeConsistencyQos method*), 983
clear() (*fastdds.TypeIdV1 method*), 996
clear() (*fastdds.TypeInformation method*), 997
clear() (*fastdds.TypeObjectV1 method*), 997
clear() (*fastdds.WireProtocolConfigQos method*), 884
clear_history() (*fastdds.DataWriter method*), 920
close() (*fastdds.Entity method*), 862
coherent_access (*fastdds.PresentationQosPolicy property*), 876
construct_sample() (*fastdds.TopicDataType method*), 985
contains_entity() (*fastdds.DomainParticipant method*), 893
copy_from_topic_qos() (*fastdds.Publisher method*), 939
copy_from_topic_qos() (*fastdds.Subscriber static method*), 976
count (*fastdds.QosPolicyCount property*), 888
create_contentfilteredtopic() (*fastdds.DomainParticipant method*), 893
create_data() (*fastdds.TypeSupport method*), 998
create_datareader() (*fastdds.Subscriber method*), 976
create_datareader_with_profile() (*fastdds.Subscriber method*), 976
create_datawriter() (*fastdds.Publisher method*), 939
create_datawriter_with_profile() (*fastdds.Publisher method*), 939
create_multitopic() (*fastdds.DomainParticipant method*), 894
create_participant() (*fastdds.DomainParticipantFactory method*), 907
create_participant_with_profile() (*fastdds.DomainParticipantFactory method*), 908
create_publisher() (*fastdds.DomainParticipant method*), 895
create_publisher_with_profile() (*fastdds.DomainParticipant method*), 895

create_querycondition() (*fastdds.DataReader method*), 946
create_readcondition() (*fastdds.DataReader method*), 947
create_subscriber() (*fastdds.DomainParticipant method*), 895
create_subscriber_with_profile() (*fastdds.DomainParticipant method*), 895
create_topic() (*fastdds.DomainParticipant method*), 895
create_topic_with_profile() (*fastdds.DomainParticipant method*), 896
createData() (*fastdds.TopicDataType method*), 985
current_count (*fastdds.MatchedStatus property*), 887
current_count_change (*fastdds.MatchedStatus property*), 887

D

data_available() (*fastdds.StatusMask static method*), 889
data_on_readers() (*fastdds.StatusMask static method*), 889
data_sharing() (*fastdds.DataReaderQos method*), 963
data_sharing() (*fastdds.DataWriterQos method*), 928
data_sharing_listener_thread() (*fastdds.DataSharingQosPolicy method*), 864
data_vec() (*fastdds.GenericDataQosPolicy method*), 870
DataReader (*class in fastdds*), 946
DATAREADER_QOS_DEFAULT (*class in fastdds*), 973
DataReaderListener (*class in fastdds*), 962
DataReaderQos (*class in fastdds*), 963
DATAREPRESENTATION_QOS_POLICY_ID (*class in fastdds*), 878
DataRepresentationQosPolicy (*class in fastdds*), 863
DataSharingQosPolicy (*class in fastdds*), 864
dataVec() (*fastdds.GenericDataQosPolicy method*), 870
DataWriter (*class in fastdds*), 920
DATAWRITER_QOS_DEFAULT (*class in fastdds*), 939
DataWriterListener (*class in fastdds*), 927
DataWriterQos (*class in fastdds*), 928
deadline() (*fastdds.DataReaderQos method*), 963
deadline() (*fastdds.DataWriterQos method*), 929
deadline() (*fastdds.TopicQos method*), 990
DEADLINE_QOS_POLICY_ID (*class in fastdds*), 878
DeadlineMissedStatus (*class in fastdds*), 886
DeadlineQosPolicy (*class in fastdds*), 866
DEFAULT_DATA_REPRESENTATION (*class in fastdds*), 863
default_external_unicast_locators (*fastdds.WireProtocolConfigQos property*), 884
default_multicast_locator_list (*fastdds.WireProtocolConfigQos property*), 884
default_unicast_locator_list (*fastdds.WireProtocolConfigQos property*), 884
delete_contained_entities() (*fastdds.DataReader method*), 947
delete_contained_entities() (*fastdds.DomainParticipant method*), 896
delete_contained_entities() (*fastdds.Publisher method*), 940
delete_contained_entities() (*fastdds.Subscriber method*), 976
delete_contentfilteredtopic() (*fastdds.DomainParticipant method*), 896
delete_data() (*fastdds.TypeSupport method*), 998
delete_datareader() (*fastdds.Subscriber method*), 976
delete_datawriter() (*fastdds.Publisher method*), 940
delete_multitopic() (*fastdds.DomainParticipant method*), 896
delete_participant() (*fastdds.DomainParticipantFactory method*), 909
delete_publisher() (*fastdds.DomainParticipant method*), 897
delete_readcondition() (*fastdds.DataReader method*), 947
delete_subscriber() (*fastdds.DomainParticipant method*), 897
delete_topic() (*fastdds.DomainParticipant method*), 897
deleteData() (*fastdds.TopicDataType method*), 986
depth (*fastdds.HistoryQosPolicy property*), 871
deserialize() (*fastdds.TopicDataType method*), 986
deserialize() (*fastdds.TypeSupport method*), 998
destination_order() (*fastdds.DataReaderQos method*), 964
destination_order() (*fastdds.DataWriterQos method*), 929
destination_order() (*fastdds.TopicQos method*), 990
DESTINATIONORDER_QOS_POLICY_ID (*class in fastdds*), 878
DestinationOrderQosPolicy (*class in fastdds*), 867
disable_heartbeat_piggyback (*fastdds.RTPSReliableWriterQos property*), 946
disable_positive_ACKs (*fastdds.RTPSReliableReaderQos property*), 974
disable_positive_acks (*fastdds.RTPSReliableWriterQos property*), 946
DISABLEPOSITIVEACKS_QOS_POLICY_ID (*class in fastdds*), 878
DisablePositiveACKsQosPolicy (*class in fastdds*), 867
DISALLOW_TYPE_COERCION (*class in fastdds*), 884
DISC_BUILTIN_ENDPOINT_PARTICIPANT_ANNOUNCER

(*C macro*), 800
DISC_BUILTIN_ENDPOINT_PARTICIPANT_DETECTOR (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_ANNOUNCER (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_DETECTOR (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PARTICIPANT_SECURE_ANNOUNCER (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PARTICIPANT_SECURE_DETECTOR (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_ANNOUNCER (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_DETECTOR (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PUBLICATION_ANNOUNCER (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PUBLICATION_DETECTOR (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PUBLICATION_SECURE_ANNOUNCER (*C macro*), 800
DISC_BUILTIN_ENDPOINT_PUBLICATION_SECURE_DETECTOR (*C macro*), 800
DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_ANNOUNCER (*C macro*), 800
DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_DETECTOR (*C macro*), 800
DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_SECURE_ANNOUNCER (*C macro*), 800
DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_SECURE_DETECTOR (*C macro*), 800
discovery_server_thread() (*fastdds.DomainParticipantQos method*), 916
dispose() (*fastdds.DataWriter method*), 920
dispose_w_timestamp() (*fastdds.DataWriter method*), 921
disposed_generation_count (*fastdds.SampleInfo property*), 974
domain_ids() (*fastdds.DataSharingQosPolicy method*), 865
DomainEntity (*class in fastdds*), 863
DomainParticipant (*class in fastdds*), 893
DomainParticipantFactory (*class in fastdds*), 907
DomainParticipantFactoryQos (*class in fastdds*), 912
DomainParticipantListener (*class in fastdds*), 913
DomainParticipantQos (*class in fastdds*), 915
durability() (*fastdds.DataReaderQos method*), 964
durability() (*fastdds.DataWriterQos method*), 930
durability() (*fastdds.TopicQos method*), 991
DURABILITY_QOS_POLICY_ID (*class in fastdds*), 878
durability_service() (*fastdds.DataReaderQos method*), 965
durability_service() (*fastdds.DataWriterQos method*), 930
durability_service() (*fastdds.TopicQos method*), 991
durabilityKind() (*fastdds.DurabilityQosPolicy method*), 868
DurabilityQosPolicy (*class in fastdds*), 868
DURABILITYSERVICE_QOS_POLICY_ID (*class in fastdds*), 878
DurabilityServiceQosPolicy (*class in fastdds*), 868
duration (*fastdds.DisablePositiveACKsQosPolicy property*), 867
duration (*fastdds.LatencyBudgetQosPolicy property*), 872
duration (*fastdds.LifespanQosPolicy property*), 872

E

empty() (*fastdds.PartitionQosPolicy method*), 875
empty() (*fastdds.TypeSupport method*), 998
enable() (*fastdds.DataReader method*), 947
enable() (*fastdds.DataWriter method*), 921
enable() (*fastdds.DomainParticipant method*), 897
enable() (*fastdds.Entity method*), 862
enable() (*fastdds.Publisher method*), 940
enable() (*fastdds.Subscriber method*), 977
enabled (*fastdds.DisablePositiveACKsQosPolicy property*), 867
end() (*fastdds.PartitionQosPolicy method*), 875
end_access() (*fastdds.Subscriber method*), 977
end_coherent_changes() (*fastdds.Publisher method*), 940
endpoint() (*fastdds.DataReaderQos method*), 965
endpoint() (*fastdds.DataWriterQos method*), 931
Entity (*class in fastdds*), 862
entity_factory() (*fastdds.DomainParticipantFactoryQos method*), 912
entity_factory() (*fastdds.DomainParticipantQos method*), 916
entity_factory() (*fastdds.PublisherQos method*), 944
entity_factory() (*fastdds.SubscriberQos method*), 981
entity_id (*fastdds.RTPSEndpointQos property*), 881
ENTITYFACTORY_QOS_POLICY_ID (*class in fastdds*), 878
EntityFactoryQosPolicy (*class in fastdds*), 869
ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_READER (*C macro*), 745
ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_WRITER (*C macro*), 745
ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_READER (*C macro*), 745
ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_WRITER (*C macro*), 745
ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_R (*C macro*), 745

ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_READER (*C macro*), 745
ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_READER (*C macro*), 745
ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_WRITER (*C macro*), 745
ENTITYID_RTPSParticipant (*C macro*), 744
ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER (*C macro*), 744
ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_READER (*C macro*), 745
ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_WRITER (*C macro*), 745
ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER (*C macro*), 744
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER (*C macro*), 744
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_READER (*C macro*), 745
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_WRITER (*C macro*), 745
ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER (*C macro*), 744
ENTITYID_SEDP_BUILTIN_TOPIC_READER (*C macro*), 744
ENTITYID_SEDP_BUILTIN_TOPIC_WRITER (*C macro*), 744
ENTITYID_SPDP_BUILTIN_RTPSParticipant_READER (*C macro*), 745
ENTITYID_SPDP_BUILTIN_RTPSParticipant_WRITER (*C macro*), 745
ENTITYID_SPDP_RELIABLE_BUILTIN_PARTICIPANT_SECURE_READER (*C macro*), 745
ENTITYID_SPDP_RELIABLE_BUILTIN_PARTICIPANT_SECURE_WRITER (*C macro*), 745
ENTITYID_TL_SVC_REPLY_READER (*C macro*), 745
ENTITYID_TL_SVC_REPLY_WRITER (*C macro*), 745
ENTITYID_TL_SVC_REQ_READER (*C macro*), 745
ENTITYID_TL_SVC_REQ_WRITER (*C macro*), 745
ENTITYID_UNKNOWN (*C macro*), 744
eprosima::fastdds::dds::BaseStatus (*C++ struct*), 611
eprosima::fastdds::dds::BaseStatus::total_count (*C++ member*), 611
eprosima::fastdds::dds::BaseStatus::total_count_change (*C++ member*), 611
eprosima::fastdds::dds::Condition (*C++ class*), 619
eprosima::fastdds::dds::Condition::get_trigger_value (*C++ function*), 620
eprosima::fastdds::dds::ConditionSeq (*C++ type*), 620
eprosima::fastdds::dds::ContentFilteredTopic (*C++ class*), 712
eprosima::fastdds::dds::ContentFilteredTopic::get_expressi (*C++ function*), 712
eprosima::fastdds::dds::ContentFilteredTopic::get_filter_e (*C++ function*), 712
eprosima::fastdds::dds::ContentFilteredTopic::get_particip (*C++ function*), 713
eprosima::fastdds::dds::ContentFilteredTopic::get_related_ (*C++ function*), 712
eprosima::fastdds::dds::ContentFilteredTopic::set_expressi (*C++ function*), 712
eprosima::fastdds::dds::ContentFilteredTopic::set_filter_e (*C++ function*), 713
eprosima::fastdds::dds::DataReader (*C++ class*), 670
eprosima::fastdds::dds::DataReader::~DataReader (*C++ function*), 681
eprosima::fastdds::dds::DataReader::create_querycondition (*C++ function*), 685
eprosima::fastdds::dds::DataReader::create_readcondition (*C++ function*), 685
eprosima::fastdds::dds::DataReader::delete_contained_entit (*C++ function*), 686
eprosima::fastdds::dds::DataReader::delete_readcondition (*C++ function*), 685
eprosima::fastdds::dds::DataReader::enable (*C++ function*), 681
eprosima::fastdds::dds::DataReader::get_first_untaken_info (*C++ function*), 682
eprosima::fastdds::dds::DataReader::get_instance_handle (*C++ function*), 683
eprosima::fastdds::dds::DataReader::get_key_value (*C++ function*), 682
eprosima::fastdds::dds::DataReader::get_listener (*C++ function*), 684
eprosima::fastdds::dds::DataReader::get_listening_locators (*C++ function*), 686
eprosima::fastdds::dds::DataReader::get_liveliness_changed (*C++ function*), 684
eprosima::fastdds::dds::DataReader::get_matched_publicatio (*C++ function*), 684
eprosima::fastdds::dds::DataReader::get_matched_publicatio (*C++ function*), 684
eprosima::fastdds::dds::DataReader::get_qos (*C++ function*), 683
eprosima::fastdds::dds::DataReader::get_requested_deadline (*C++ function*), 683
eprosima::fastdds::dds::DataReader::get_requested_incompat (*C++ function*), 683
eprosima::fastdds::dds::DataReader::get_sample_lost_status (*C++ function*), 684
eprosima::fastdds::dds::DataReader::get_sample_rejected_st (*C++ function*), 684
eprosima::fastdds::dds::DataReader::get_subscriber (*C++ function*), 685

eprosima::fastdds::dds::DataReader::get_subscription_matched_status (C++ function), 684
eprosima::fastdds::dds::DataReader::get_topicdescription (C++ function), 683
eprosima::fastdds::dds::DataReader::get_unread_count (C++ function), 682, 683
eprosima::fastdds::dds::DataReader::guid (C++ function), 683
eprosima::fastdds::dds::DataReader::is_sample_valid (C++ function), 686
eprosima::fastdds::dds::DataReader::lookup_instance (C++ function), 682
eprosima::fastdds::dds::DataReader::read (C++ function), 670
eprosima::fastdds::dds::DataReader::read_instance (C++ function), 674
eprosima::fastdds::dds::DataReader::read_next_instance (C++ function), 675
eprosima::fastdds::dds::DataReader::read_next_instance_with_condition (C++ function), 676
eprosima::fastdds::dds::DataReader::read_next_sample (C++ function), 676
eprosima::fastdds::dds::DataReader::read_w_condition (C++ function), 673
eprosima::fastdds::dds::DataReader::return_loan (C++ function), 681
eprosima::fastdds::dds::DataReader::set_listener (C++ function), 683, 684
eprosima::fastdds::dds::DataReader::set_qos (C++ function), 683
eprosima::fastdds::dds::DataReader::take (C++ function), 677
eprosima::fastdds::dds::DataReader::take_instance (C++ function), 678
eprosima::fastdds::dds::DataReader::take_next_instance (C++ function), 679
eprosima::fastdds::dds::DataReader::take_next_instance_with_condition (C++ function), 680
eprosima::fastdds::dds::DataReader::take_next_sample (C++ function), 680
eprosima::fastdds::dds::DataReader::take_w_condition (C++ function), 678
eprosima::fastdds::dds::DataReader::type (C++ function), 683
eprosima::fastdds::dds::DataReader::wait_for_historical_data (C++ function), 681
eprosima::fastdds::dds::DataReader::wait_for_unread_message (C++ function), 681
eprosima::fastdds::dds::DATAREADER_QOS_DEFAULT (C++ member), 692
eprosima::fastdds::dds::DataReaderListener (C++ class), 686
eprosima::fastdds::dds::DataReaderListener::~DataReaderListener (C++ function), 686
eprosima::fastdds::dds::DataReaderListener::DataReaderListener (C++ function), 686
eprosima::fastdds::dds::DataReaderListener::on_data_available (C++ function), 686
eprosima::fastdds::dds::DataReaderListener::on_liveliness_changed (C++ function), 687
eprosima::fastdds::dds::DataReaderListener::on_requested_deadline_missed (C++ function), 687
eprosima::fastdds::dds::DataReaderListener::on_requested_incompatible_qos (C++ function), 687
eprosima::fastdds::dds::DataReaderListener::on_sample_lost (C++ function), 687
eprosima::fastdds::dds::DataReaderListener::on_sample_rejected (C++ function), 687
eprosima::fastdds::dds::DataReaderListener::on_subscription_matched (C++ function), 686
eprosima::fastdds::dds::DataReaderQos (C++ class), 687
eprosima::fastdds::dds::DataReaderQos::data_sharing (C++ function), 692
eprosima::fastdds::dds::DataReaderQos::DataReaderQos (C++ function), 688
eprosima::fastdds::dds::DataReaderQos::deadline (C++ function), 688
eprosima::fastdds::dds::DataReaderQos::destination_order (C++ function), 689
eprosima::fastdds::dds::DataReaderQos::durability (C++ function), 688
eprosima::fastdds::dds::DataReaderQos::durability_service (C++ function), 691
eprosima::fastdds::dds::DataReaderQos::endpoint (C++ function), 692
eprosima::fastdds::dds::DataReaderQos::expects_inline_qos (C++ function), 691
eprosima::fastdds::dds::DataReaderQos::history (C++ function), 689
eprosima::fastdds::dds::DataReaderQos::latency_budget (C++ function), 688
eprosima::fastdds::dds::DataReaderQos::lifespan (C++ function), 690
eprosima::fastdds::dds::DataReaderQos::liveliness (C++ function), 688
eprosima::fastdds::dds::DataReaderQos::ownership (C++ function), 690
eprosima::fastdds::dds::DataReaderQos::properties (C++ function), 691
eprosima::fastdds::dds::DataReaderQos::reader_data_lifecycle (C++ function), 690
eprosima::fastdds::dds::DataReaderQos::reader_resource_limits (C++ function), 692
eprosima::fastdds::dds::DataReaderQos::reliability (C++ function), 689
eprosima::fastdds::dds::DataReaderQos::reliable_reader_qos (C++ function), 691

eprosima::fastdds::dds::DataReaderQos::resource_limits (C++ function), 689
eprosima::fastdds::dds::DataReaderQos::time_based_filter (C++ function), 690
eprosima::fastdds::dds::DataReaderQos::type_consistency (C++ function), 691
eprosima::fastdds::dds::DataReaderQos::user_data (C++ function), 689, 690
eprosima::fastdds::dds::DataRepresentationId (C++ enum), 576
eprosima::fastdds::dds::DataRepresentationId::AUTO (C++ enumerator), 577
eprosima::fastdds::dds::DataRepresentationId::XCDR_DATA_REPRESENTATION (C++ enumerator), 576
eprosima::fastdds::dds::DataRepresentationId::XML_DATA_REPRESENTATION (C++ enumerator), 577
eprosima::fastdds::dds::DataRepresentationQosPolicy (C++ class), 577
eprosima::fastdds::dds::DataRepresentationQosPolicy::DataRepresentationQosPolicy (C++ function), 577
eprosima::fastdds::dds::DataRepresentationQosPolicy::clear (C++ function), 577
eprosima::fastdds::dds::DataRepresentationQosPolicy::operator= (C++ function), 577
eprosima::fastdds::dds::DataRepresentationQosPolicy::value (C++ member), 578
eprosima::fastdds::dds::DataRepresentationQosPolicy::xcdr_data_representation_preferred (C++ function), 577
eprosima::fastdds::dds::DataSharingKind (C++ enum), 580
eprosima::fastdds::dds::DataSharingKind::AUTO (C++ enumerator), 580
eprosima::fastdds::dds::DataSharingKind::OFF (C++ enumerator), 580
eprosima::fastdds::dds::DataSharingKind::ON (C++ enumerator), 580
eprosima::fastdds::dds::DataSharingQosPolicy (C++ class), 578
eprosima::fastdds::dds::DataSharingQosPolicy::DataSharingQosPolicy (C++ function), 578
eprosima::fastdds::dds::DataSharingQosPolicy::add_domain_id (C++ function), 579
eprosima::fastdds::dds::DataSharingQosPolicy::automatic (C++ function), 579
eprosima::fastdds::dds::DataSharingQosPolicy::clear (C++ function), 578
eprosima::fastdds::dds::DataSharingQosPolicy::data_sharing_listeners_thread (C++ function), 579, 580
eprosima::fastdds::dds::DataSharingQosPolicy::DataSharingQosPolicy (C++ function), 578
eprosima::fastdds::dds::DataSharingQosPolicy::domain_ids (C++ function), 578
eprosima::fastdds::dds::DataSharingQosPolicy::kind (C++ function), 578
eprosima::fastdds::dds::DataSharingQosPolicy::max_domains (C++ function), 579
eprosima::fastdds::dds::DataSharingQosPolicy::off (C++ function), 579
eprosima::fastdds::dds::DataSharingQosPolicy::on (C++ function), 579
eprosima::fastdds::dds::DataSharingQosPolicy::set_max_doma (C++ function), 578
eprosima::fastdds::dds::DataSharingQosPolicy::shm_director (C++ function), 578
eprosima::fastdds::dds::DataWriter (C++ class), 649
eprosima::fastdds::dds::DataWriter::assert_liveliness (C++ function), 654
eprosima::fastdds::dds::DataWriter::clear_history (C++ function), 655
eprosima::fastdds::dds::DataWriter::discard_loan (C++ function), 656
eprosima::fastdds::dds::DataWriter::dispose (C++ function), 654
eprosima::fastdds::dds::DataWriter::dispose_w_timestamp (C++ function), 654
eprosima::fastdds::dds::DataWriter::enable (C++ function), 650
eprosima::fastdds::dds::DataWriter::get_instance_handle (C++ function), 652
eprosima::fastdds::dds::DataWriter::get_key_value (C++ function), 652
eprosima::fastdds::dds::DataWriter::get_listener (C++ function), 653
eprosima::fastdds::dds::DataWriter::get_liveliness_lost_st (C++ function), 654
eprosima::fastdds::dds::DataWriter::get_matched_subscripti (C++ function), 655
eprosima::fastdds::dds::DataWriter::get_matched_subscripti (C++ function), 655
eprosima::fastdds::dds::DataWriter::get_offered_deadline_m (C++ function), 653
eprosima::fastdds::dds::DataWriter::get_offered_incompatib (C++ function), 653
eprosima::fastdds::dds::DataWriter::get_publication_matche (C++ function), 653
eprosima::fastdds::dds::DataWriter::get_publisher (C++ function), 654
eprosima::fastdds::dds::DataWriter::get_qos (C++ function), 653
eprosima::fastdds::dds::DataWriter::get_sending_locators (C++ function), 656
eprosima::fastdds::dds::DataWriter::get_topic (C++ function), 653
eprosima::fastdds::dds::DataWriter::get_type (C++ function), 652
eprosima::fastdds::dds::DataWriter::guid (C++ function), 652

eprosima::fastdds::dds::DataWriter::loan_sample (C++ function), 655
eprosima::fastdds::dds::DataWriter::LoanInitializationKind (C++ enum), 649
eprosima::fastdds::dds::DataWriter::LoanInitializationKind::CONSTRUCTED_LOAN_INITIALIZATION (C++ enumerator), 650
eprosima::fastdds::dds::DataWriter::LoanInitializationKind::NO_LOAN_INITIALIZATION (C++ enumerator), 649
eprosima::fastdds::dds::DataWriter::LoanInitializationKind::ZERO_LOAN_INITIALIZATION (C++ enumerator), 649
eprosima::fastdds::dds::DataWriter::lookup_instance (C++ function), 652
eprosima::fastdds::dds::DataWriter::register_instance (C++ function), 651
eprosima::fastdds::dds::DataWriter::register_instance_w_timestamp (C++ function), 651
eprosima::fastdds::dds::DataWriter::set_listener (C++ function), 653
eprosima::fastdds::dds::DataWriter::set_qos (C++ function), 653
eprosima::fastdds::dds::DataWriter::unregister_instance (C++ function), 651
eprosima::fastdds::dds::DataWriter::unregister_instance_w_timestamp (C++ function), 651
eprosima::fastdds::dds::DataWriter::wait_for_acknowledgments (C++ function), 652, 656
eprosima::fastdds::dds::DataWriter::write (C++ function), 650
eprosima::fastdds::dds::DataWriter::write_w_timestamp (C++ function), 650
eprosima::fastdds::dds::DATAWRITER_QOS_DEFAULT (C++ member), 663
eprosima::fastdds::dds::DataWriterListener (C++ class), 657
eprosima::fastdds::dds::DataWriterListener::~DataWriterListener (C++ function), 657
eprosima::fastdds::dds::DataWriterListener::DataWriterListener (C++ function), 657
eprosima::fastdds::dds::DataWriterListener::on_liveliness_lost (C++ function), 657
eprosima::fastdds::dds::DataWriterListener::on_offered_deadline_missed (C++ function), 657
eprosima::fastdds::dds::DataWriterListener::on_offered_incompatible_qos (C++ function), 657
eprosima::fastdds::dds::DataWriterListener::on_publication_matched (C++ function), 657
eprosima::fastdds::dds::DataWriterListener::on_unacknowledged_sample_removed (C++ function), 657
eprosima::fastdds::dds::DataWriterQos (C++ class), 658
eprosima::fastdds::dds::DataWriterQos::~DataWriterQos (C++ function), 658
eprosima::fastdds::dds::DataWriterQos::data_sharing (C++ function), 663
eprosima::fastdds::dds::DataWriterQos::DataWriterQos (C++ function), 658
eprosima::fastdds::dds::DataWriterQos::deadline (C++ function), 658, 659
eprosima::fastdds::dds::DataWriterQos::destination_order (C++ function), 659, 660
eprosima::fastdds::dds::DataWriterQos::durability (C++ function), 658
eprosima::fastdds::dds::DataWriterQos::durability_service (C++ function), 658
eprosima::fastdds::dds::DataWriterQos::endpoint (C++ function), 662
eprosima::fastdds::dds::DataWriterQos::history (C++ function), 660
eprosima::fastdds::dds::DataWriterQos::latency_budget (C++ function), 659
eprosima::fastdds::dds::DataWriterQos::lifespan (C++ function), 660
eprosima::fastdds::dds::DataWriterQos::liveliness (C++ function), 659
eprosima::fastdds::dds::DataWriterQos::ownership (C++ function), 661
eprosima::fastdds::dds::DataWriterQos::ownership_strength (C++ function), 661
eprosima::fastdds::dds::DataWriterQos::properties (C++ function), 662
eprosima::fastdds::dds::DataWriterQos::publish_mode (C++ function), 661, 662
eprosima::fastdds::dds::DataWriterQos::reliability (C++ function), 659
eprosima::fastdds::dds::DataWriterQos::reliable_writer_qos (C++ function), 662
eprosima::fastdds::dds::DataWriterQos::representation (C++ function), 662
eprosima::fastdds::dds::DataWriterQos::resource_limits (C++ function), 660
eprosima::fastdds::dds::DataWriterQos::throughput_controller (C++ function), 663
eprosima::fastdds::dds::DataWriterQos::transport_priority (C++ function), 660
eprosima::fastdds::dds::DataWriterQos::user_data (C++ function), 660, 661
eprosima::fastdds::dds::DataWriterQos::writer_data_lifecycle (C++ function), 661
eprosima::fastdds::dds::DataWriterQos::writer_resource_limits (C++ function), 662, 663
eprosima::fastdds::dds::DeadlineMissedStatus (C++ struct), 611
eprosima::fastdds::dds::DeadlineMissedStatus::~DeadlineMissedStatus (C++ function), 612
eprosima::fastdds::dds::DeadlineMissedStatus::DeadlineMissedStatus (C++ function), 612
eprosima::fastdds::dds::DeadlineMissedStatus::last_instance_handle (C++ member), 612

eprosima::fastdds::dds::DeadlineMissedStatus::total_count (C++ member), 612
eprosima::fastdds::dds::DeadlineMissedStatus::total_count_change (C++ member), 612
eprosima::fastdds::dds::DeadlineQosPolicy (C++ class), 580
eprosima::fastdds::dds::DeadlineQosPolicy::~DeadlineQosPolicy (C++ function), 580
eprosima::fastdds::dds::DeadlineQosPolicy::clear (C++ function), 580
eprosima::fastdds::dds::DeadlineQosPolicy::DeadlineQosPolicy (C++ function), 580
eprosima::fastdds::dds::DeadlineQosPolicy::period (C++ member), 581
eprosima::fastdds::dds::DEFAULT_DATA_REPRESENTATION (C++ member), 577
eprosima::fastdds::dds::DestinationOrderQosPolicy (C++ class), 581
eprosima::fastdds::dds::DestinationOrderQosPolicy::~DestinationOrderQosPolicy (C++ function), 581
eprosima::fastdds::dds::DestinationOrderQosPolicy::clear (C++ function), 581
eprosima::fastdds::dds::DestinationOrderQosPolicy::DestinationOrderQosPolicy (C++ function), 581
eprosima::fastdds::dds::DestinationOrderQosPolicy::kind (C++ member), 581
eprosima::fastdds::dds::DestinationOrderQosPolicyKind (C++ enum), 582
eprosima::fastdds::dds::DestinationOrderQosPolicyKind::BY_RECEPTION_TIMESTAMP_DESTINATION_ORDER_QOS (C++ enumerator), 582
eprosima::fastdds::dds::DestinationOrderQosPolicyKind::BY_SOURCE_TIMESTAMP_DESTINATION_ORDER_QOS (C++ enumerator), 582
eprosima::fastdds::dds::DisablePositiveACKsQosPolicy (C++ class), 582
eprosima::fastdds::dds::DisablePositiveACKsQosPolicy::~DisablePositiveACKsQosPolicy (C++ function), 582
eprosima::fastdds::dds::DisablePositiveACKsQosPolicy::clear (C++ function), 582
eprosima::fastdds::dds::DisablePositiveACKsQosPolicy::DisablePositiveACKsQosPolicy (C++ function), 582
eprosima::fastdds::dds::DisablePositiveACKsQosPolicy::duration (C++ member), 582
eprosima::fastdds::dds::DisablePositiveACKsQosPolicy::enabled (C++ member), 582
eprosima::fastdds::dds::DomainEntity (C++ class), 576
eprosima::fastdds::dds::DomainEntity::DomainEntity (C++ function), 576
eprosima::fastdds::dds::DomainParticipant (C++ class), 626
eprosima::fastdds::dds::DomainParticipant::~DomainParticipant (C++ function), 626
eprosima::fastdds::dds::DomainParticipant::assert_liveliness (C++ function), 632
eprosima::fastdds::dds::DomainParticipant::contains_entity (C++ function), 636
eprosima::fastdds::dds::DomainParticipant::create_contentfilteredtopic (C++ function), 629, 630
eprosima::fastdds::dds::DomainParticipant::create_multitopic (C++ function), 630
eprosima::fastdds::dds::DomainParticipant::create_publisher (C++ function), 627
eprosima::fastdds::dds::DomainParticipant::create_publisher (C++ function), 628
eprosima::fastdds::dds::DomainParticipant::create_subscriber (C++ function), 628
eprosima::fastdds::dds::DomainParticipant::create_subscriber (C++ function), 628
eprosima::fastdds::dds::DomainParticipant::create_topic (C++ function), 628
eprosima::fastdds::dds::DomainParticipant::create_topic_with_profile (C++ function), 629
eprosima::fastdds::dds::DomainParticipant::delete_contained_entities (C++ function), 632
eprosima::fastdds::dds::DomainParticipant::delete_contentfilteredtopic (C++ function), 630
eprosima::fastdds::dds::DomainParticipant::delete_multitopic (C++ function), 630
eprosima::fastdds::dds::DomainParticipant::delete_publisher (C++ function), 628
eprosima::fastdds::dds::DomainParticipant::delete_subscriber (C++ function), 628
eprosima::fastdds::dds::DomainParticipant::delete_topic (C++ function), 629
eprosima::fastdds::dds::DomainParticipant::find_topic (C++ function), 627
eprosima::fastdds::dds::DomainParticipant::find_type (C++ function), 631
eprosima::fastdds::dds::DomainParticipant::get_builtin_subscriber (C++ function), 637
eprosima::fastdds::dds::DomainParticipant::get_current_time (C++ function), 631
eprosima::fastdds::dds::DomainParticipant::get_default_publisher_qos (C++ function), 636
eprosima::fastdds::dds::DomainParticipant::get_default_subscriber_qos (C++ function), 633
eprosima::fastdds::dds::DomainParticipant::get_default_topic_qos (C++ function), 634
eprosima::fastdds::dds::DomainParticipant::get_discovered_participant_data (C++ function), 634, 635
eprosima::fastdds::dds::DomainParticipant::get_discovered_participants (C++ function), 635
eprosima::fastdds::dds::DomainParticipant::get_discovered_topic_data (C++ function), 635
eprosima::fastdds::dds::DomainParticipant::get_discovered_topics (C++ function), 636
eprosima::fastdds::dds::DomainParticipant::get_discovered_topics_size (C++ function), 635

eprosima::fastdds::dds::DomainParticipant::get_domain_id (C++ function), 632
eprosima::fastdds::dds::DomainParticipant::get_instance_handle (C++ function), 637
eprosima::fastdds::dds::DomainParticipant::get_listener (C++ function), 626
eprosima::fastdds::dds::DomainParticipant::get_participant_names (C++ function), 637
eprosima::fastdds::dds::DomainParticipant::get_publisher_profile_names (C++ function), 633
eprosima::fastdds::dds::DomainParticipant::get_qos (C++ function), 626
eprosima::fastdds::dds::DomainParticipant::get_resource_event (C++ function), 637
eprosima::fastdds::dds::DomainParticipant::get_subscriber_profile_names (C++ function), 634
eprosima::fastdds::dds::DomainParticipant::get_topic_qos_from_profile (C++ function), 635
eprosima::fastdds::dds::DomainParticipant::get_type_dependencies (C++ function), 637
eprosima::fastdds::dds::DomainParticipant::get_types (C++ function), 638
eprosima::fastdds::dds::DomainParticipant::guid (C++ function), 637
eprosima::fastdds::dds::DomainParticipant::has_active_entities (C++ function), 639
eprosima::fastdds::dds::DomainParticipant::ignore_participant (C++ function), 631
eprosima::fastdds::dds::DomainParticipant::ignore_publication (C++ function), 632
eprosima::fastdds::dds::DomainParticipant::ignore_subscription (C++ function), 632
eprosima::fastdds::dds::DomainParticipant::ignore_topic (C++ function), 631
eprosima::fastdds::dds::DomainParticipant::lookup_participant_factory (C++ function), 639
eprosima::fastdds::dds::DomainParticipant::lookup_participant_profile (C++ function), 631
eprosima::fastdds::dds::DomainParticipant::new_remote_endpoint_discovered (C++ function), 637
eprosima::fastdds::dds::DomainParticipant::register_content_filtered_topic_type (C++ function), 638
eprosima::fastdds::dds::DomainParticipant::register_remote_type (C++ function), 638
eprosima::fastdds::dds::DomainParticipant::register_type (C++ function), 636, 637
eprosima::fastdds::dds::DomainParticipant::set_default_publisher_qos (C++ function), 633
eprosima::fastdds::dds::DomainParticipant::set_default_subscriber_qos (C++ function), 633
eprosima::fastdds::dds::DomainParticipant::set_default_topic_qos (C++ function), 634
eprosima::fastdds::dds::DomainParticipant::set_listener (C++ function), 626, 627
eprosima::fastdds::dds::DomainParticipant::set_qos (C++ function), 626
eprosima::fastdds::dds::DomainParticipant::unregister_content_filtered_topic_type (C++ function), 639
eprosima::fastdds::dds::DomainParticipant::unregister_type (C++ function), 637
eprosima::fastdds::dds::DomainParticipantFactory (C++ class), 640
eprosima::fastdds::dds::DomainParticipantFactory::check_xml_static_discovery_profile (C++ function), 642
eprosima::fastdds::dds::DomainParticipantFactory::create_participant (C++ function), 640
eprosima::fastdds::dds::DomainParticipantFactory::create_participant_with_default_profile (C++ function), 640
eprosima::fastdds::dds::DomainParticipantFactory::delete_participant (C++ function), 641
eprosima::fastdds::dds::DomainParticipantFactory::get_default_domain_id (C++ function), 641
eprosima::fastdds::dds::DomainParticipantFactory::get_instance (C++ function), 643
eprosima::fastdds::dds::DomainParticipantFactory::get_participant (C++ function), 641
eprosima::fastdds::dds::DomainParticipantFactory::get_qos (C++ function), 642
eprosima::fastdds::dds::DomainParticipantFactory::get_shared_resource_event_thread (C++ function), 643
eprosima::fastdds::dds::DomainParticipantFactory::load_profiles (C++ function), 642
eprosima::fastdds::dds::DomainParticipantFactory::load_XML_profiles_file (C++ function), 642
eprosima::fastdds::dds::DomainParticipantFactory::load_XML_profiles_string (C++ function), 642
eprosima::fastdds::dds::DomainParticipantFactory::lookup_participant (C++ function), 640
eprosima::fastdds::dds::DomainParticipantFactory::lookup_participant_factory (C++ function), 641
eprosima::fastdds::dds::DomainParticipantFactory::set_default_domain_id (C++ function), 641
eprosima::fastdds::dds::DomainParticipantFactory::set_qos (C++ function), 642
eprosima::fastdds::dds::DomainParticipantFactoryQos (C++ class), 643
eprosima::fastdds::dds::DomainParticipantFactoryQos::~DomainParticipantFactoryQos (C++ function), 643
eprosima::fastdds::dds::DomainParticipantFactoryQos::DomainParticipantFactoryQos (C++ function), 643
eprosima::fastdds::dds::DomainParticipantFactoryQos::entity_factory (C++ function), 643
eprosima::fastdds::dds::DomainParticipantFactoryQos::file_watch_handlers (C++ function), 643, 644
eprosima::fastdds::dds::DomainParticipantFactoryQos::shm_watchdog_thread_config (C++ function), 643
eprosima::fastdds::dds::DomainParticipantListener (C++ class), 644

eprosima::fastdds::dds::DomainParticipantListener::on_publisher_discovery (*C++ function*), 644
eprosima::fastdds::dds::DomainParticipantListener::on_subscriber_discovery (*C++ function*), 644
eprosima::fastdds::dds::DomainParticipantListener::on_type_dependencies_reply (*C++ function*), 644
eprosima::fastdds::dds::DomainParticipantListener::on_type_discovery (*C++ function*), 645
eprosima::fastdds::dds::DomainParticipantListener::on_type_information_received (*C++ function*), 645
eprosima::fastdds::dds::DomainParticipantListener::~DomainParticipantListener (*C++ function*), 644
eprosima::fastdds::dds::DomainParticipantQos (*C++ class*), 646
eprosima::fastdds::dds::DomainParticipantQos::allocation (*C++ function*), 646
eprosima::fastdds::dds::DomainParticipantQos::DomainParticipantQos (*C++ function*), 647
eprosima::fastdds::dds::DomainParticipantQos::flow_controllers (*C++ function*), 648
eprosima::fastdds::dds::DomainParticipantQos::FlowControllerDescriptorList (*C++ type*), 646
eprosima::fastdds::dds::DomainParticipantQos::name (*C++ function*), 648
eprosima::fastdds::dds::DomainParticipantQos::properties (*C++ function*), 647
eprosima::fastdds::dds::DomainParticipantQos::security_log_threads (*C++ function*), 649
eprosima::fastdds::dds::DomainParticipantQos::setup_transports (*C++ function*), 648
eprosima::fastdds::dds::DomainParticipantQos::timed_events_thread (*C++ function*), 648, 649
eprosima::fastdds::dds::DomainParticipantQos::transport (*C++ function*), 647, 648
eprosima::fastdds::dds::DomainParticipantQos::user_data (*C++ function*), 646
eprosima::fastdds::dds::DomainParticipantQos::wire_protocol (*C++ function*), 647
eprosima::fastdds::dds::DurabilityQosPolicy (*C++ class*), 583
eprosima::fastdds::dds::DurabilityQosPolicy::~DurabilityQosPolicy (*C++ function*), 583
eprosima::fastdds::dds::DurabilityQosPolicy::clear (*C++ function*), 583
eprosima::fastdds::dds::DurabilityQosPolicy::durabilityKind (*C++ function*), 583
eprosima::fastdds::dds::DurabilityQosPolicy::DurabilityQosPolicy (*C++ function*), 583
eprosima::fastdds::dds::DurabilityQosPolicy::kind (*C++ member*), 583
eprosima::fastdds::dds::DurabilityQosPolicyKind (*C++ enum*), 583
eprosima::fastdds::dds::DurabilityQosPolicyKind::PERSISTENT (*C++ enumerator*), 584
eprosima::fastdds::dds::DurabilityQosPolicyKind::TRANSIENT (*C++ enumerator*), 583
eprosima::fastdds::dds::DurabilityQosPolicyKind::TRANSIENT_LOCAL (*C++ enumerator*), 583
eprosima::fastdds::dds::DurabilityQosPolicyKind::VOLATILE (*C++ enumerator*), 583
eprosima::fastdds::dds::DurabilityServiceQosPolicy (*C++ class*), 584
eprosima::fastdds::dds::DurabilityServiceQosPolicy::~DurabilityServiceQosPolicy (*C++ function*), 584
eprosima::fastdds::dds::DurabilityServiceQosPolicy::clear (*C++ function*), 584
eprosima::fastdds::dds::DurabilityServiceQosPolicy::DurabilityServiceQosPolicy (*C++ function*), 584
eprosima::fastdds::dds::DurabilityServiceQosPolicy::history_depth (*C++ member*), 584
eprosima::fastdds::dds::DurabilityServiceQosPolicy::history_kind (*C++ member*), 584
eprosima::fastdds::dds::DurabilityServiceQosPolicy::max_instances (*C++ member*), 585
eprosima::fastdds::dds::DurabilityServiceQosPolicy::max_samples (*C++ member*), 585
eprosima::fastdds::dds::DurabilityServiceQosPolicy::max_samples_per_instance (*C++ member*), 585
eprosima::fastdds::dds::DurabilityServiceQosPolicy::service_cleanup_delay (*C++ member*), 584
eprosima::fastdds::dds::Entity (*C++ class*), 575
eprosima::fastdds::dds::Entity::close (*C++ function*), 575
eprosima::fastdds::dds::Entity::enable (*C++ function*), 575
eprosima::fastdds::dds::Entity::Entity (*C++ function*), 575
eprosima::fastdds::dds::Entity::get_instance_handle (*C++ function*), 576
eprosima::fastdds::dds::Entity::get_status_changes (*C++ function*), 576
eprosima::fastdds::dds::Entity::get_status_mask (*C++ function*), 576
eprosima::fastdds::dds::Entity::get_statuscondition

(C++ function), 576
eprosima::fastdds::dds::Entity::is_enabled (C++ function), 576
eprosima::fastdds::dds::EntityFactoryQosPolicy (C++ class), 585
eprosima::fastdds::dds::EntityFactoryQosPolicy::EntityFactoryQosPolicy (C++ function), 585
eprosima::fastdds::dds::EntityFactoryQosPolicy::autoenable_created_entities (C++ member), 586
eprosima::fastdds::dds::EntityFactoryQosPolicy::~EntityFactoryQosPolicy (C++ function), 585
eprosima::fastdds::dds::FileConsumer (C++ class), 853
eprosima::fastdds::dds::FileConsumer::FileConsumer (C++ function), 853
eprosima::fastdds::dds::GenericDataQosPolicy (C++ class), 586
eprosima::fastdds::dds::GenericDataQosPolicy::clear (C++ function), 587
eprosima::fastdds::dds::GenericDataQosPolicy::data_vec (C++ function), 587
eprosima::fastdds::dds::GenericDataQosPolicy::getMaxSize (C++ function), 587
eprosima::fastdds::dds::GenericDataQosPolicy::GenericDataQosPolicy (C++ function), 586
eprosima::fastdds::dds::GenericDataQosPolicy::getValue (C++ function), 587
eprosima::fastdds::dds::GenericDataQosPolicy::operator= (C++ function), 586
eprosima::fastdds::dds::GenericDataQosPolicy::setMaxSize (C++ function), 587
eprosima::fastdds::dds::GenericDataQosPolicy::setValue (C++ function), 587
eprosima::fastdds::dds::GroupDataQosPolicy (C++ class), 587
eprosima::fastdds::dds::GuardCondition (C++ class), 620
eprosima::fastdds::dds::GuardCondition::get_trigger_value (C++ function), 620
eprosima::fastdds::dds::GuardCondition::set_trigger_value (C++ function), 620
eprosima::fastdds::dds::HistoryQosPolicy (C++ class), 588
eprosima::fastdds::dds::HistoryQosPolicy::~HistoryQosPolicy (C++ function), 588
eprosima::fastdds::dds::HistoryQosPolicy::clear (C++ function), 588
eprosima::fastdds::dds::HistoryQosPolicy::depth (C++ member), 588
eprosima::fastdds::dds::HistoryQosPolicy::HistoryQosPolicy (C++ function), 588
eprosima::fastdds::dds::HistoryQosPolicy::kind (C++ member), 588
eprosima::fastdds::dds::HistoryQosPolicyKind (C++ enum), 589
eprosima::fastdds::dds::HistoryQosPolicyKind::KEEP_ALL_HIS (C++ enumerator), 589
eprosima::fastdds::dds::HistoryQosPolicyKind::KEEP_LAST_HI (C++ enumerator), 589
eprosima::fastdds::dds::IContentFilter (C++ struct), 713
eprosima::fastdds::dds::IContentFilter::evaluate (C++ function), 713
eprosima::fastdds::dds::IContentFilter::FilterSampleInfo (C++ struct), 713
eprosima::fastdds::dds::IContentFilter::FilterSampleInfo::: (C++ member), 714
eprosima::fastdds::dds::IContentFilter::FilterSampleInfo::: (C++ member), 714
eprosima::fastdds::dds::IContentFilterFactory (C++ struct), 714
eprosima::fastdds::dds::IContentFilterFactory::create_cont (C++ function), 714
eprosima::fastdds::dds::IContentFilterFactory::delete_cont (C++ function), 714
eprosima::fastdds::dds::IncompatibleQosStatus (C++ struct), 612
eprosima::fastdds::dds::IncompatibleQosStatus::last_policy (C++ member), 612
eprosima::fastdds::dds::IncompatibleQosStatus::policies (C++ member), 612
eprosima::fastdds::dds::IncompatibleQosStatus::total_count (C++ member), 612
eprosima::fastdds::dds::IncompatibleQosStatus::total_count (C++ member), 612
eprosima::fastdds::dds::InconsistentTopicStatus (C++ type), 613
eprosima::fastdds::dds::InstanceStateKind (C++ enum), 692
eprosima::fastdds::dds::InstanceStateKind::ALIVE_INSTANCE_ (C++ enumerator), 693
eprosima::fastdds::dds::InstanceStateKind::NOT_ALIVE_DISPO (C++ enumerator), 693
eprosima::fastdds::dds::InstanceStateKind::NOT_ALIVE_NO_WR (C++ enumerator), 693
eprosima::fastdds::dds::LatencyBudgetQosPolicy (C++ class), 589
eprosima::fastdds::dds::LatencyBudgetQosPolicy::~LatencyBu (C++ function), 589
eprosima::fastdds::dds::LatencyBudgetQosPolicy::clear (C++ function), 589
eprosima::fastdds::dds::LatencyBudgetQosPolicy::duration (C++ member), 590
eprosima::fastdds::dds::LatencyBudgetQosPolicy::LatencyBud (C++ function), 589
eprosima::fastdds::dds::LifespanQosPolicy (C++ class), 590
eprosima::fastdds::dds::LifespanQosPolicy::~LifespanQosPol

(C++ function), 590
eprosima::fastdds::dds::LifespanQosPolicy::clear (C++ function), 590
eprosima::fastdds::dds::LifespanQosPolicy::duration (C++ member), 590
eprosima::fastdds::dds::LifespanQosPolicy::LifespanQosPolicy (C++ function), 590
eprosima::fastdds::dds::LivelinessChangedStatus (C++ struct), 613
eprosima::fastdds::dds::LivelinessChangedStatus::alive_count (C++ member), 613
eprosima::fastdds::dds::LivelinessChangedStatus::alive_count_change (C++ member), 613
eprosima::fastdds::dds::LivelinessChangedStatus::last_publication_handle (C++ member), 613
eprosima::fastdds::dds::LivelinessChangedStatus::not_alive_count (C++ member), 613
eprosima::fastdds::dds::LivelinessChangedStatus::not_alive_count_change (C++ member), 613
eprosima::fastdds::dds::LivelinessLostStatus (C++ type), 616
eprosima::fastdds::dds::LivelinessQosPolicy (C++ class), 590
eprosima::fastdds::dds::LivelinessQosPolicy::~LivelinessQosPolicy (C++ function), 591
eprosima::fastdds::dds::LivelinessQosPolicy::announcement_period (C++ member), 591
eprosima::fastdds::dds::LivelinessQosPolicy::clear (C++ function), 591
eprosima::fastdds::dds::LivelinessQosPolicy::kind (C++ member), 591
eprosima::fastdds::dds::LivelinessQosPolicy::lease_duration (C++ member), 591
eprosima::fastdds::dds::LivelinessQosPolicy::LivelinessQosPolicy (C++ function), 591
eprosima::fastdds::dds::LivelinessQosPolicyKind (C++ enum), 591
eprosima::fastdds::dds::LivelinessQosPolicyKind::AUTOMATIC_LIVELINESS_QOS (C++ enumerator), 591
eprosima::fastdds::dds::LivelinessQosPolicyKind::MANUAL_BY_PARTICIPANT_LIVELINESS_QOS (C++ enumerator), 591
eprosima::fastdds::dds::LivelinessQosPolicyKind::MANUAL_BY_TOPIC_LIVELINESS_QOS (C++ enumerator), 591
eprosima::fastdds::dds::LoanableArray (C++ struct), 622
eprosima::fastdds::dds::LoanableArray::buffer_for_loans (C++ function), 622
eprosima::fastdds::dds::LoanableCollection (C++ class), 622
eprosima::fastdds::dds::LoanableCollection::buffer (C++ function), 622
eprosima::fastdds::dds::LoanableCollection::has_ownership (C++ function), 622
eprosima::fastdds::dds::LoanableCollection::length (C++ function), 622
eprosima::fastdds::dds::LoanableCollection::loan (C++ function), 623
eprosima::fastdds::dds::LoanableCollection::maximum (C++ function), 622
eprosima::fastdds::dds::LoanableCollection::unloan (C++ function), 623
eprosima::fastdds::dds::LoanableSequence (C++ class), 624
eprosima::fastdds::dds::LoanableSequence::~LoanableSequence (C++ function), 625
eprosima::fastdds::dds::LoanableSequence::LoanableSequence (C++ function), 624, 625
eprosima::fastdds::dds::LoanableSequence::operator= (C++ function), 625
eprosima::fastdds::dds::Log (C++ class), 853
eprosima::fastdds::dds::Log::ClearConsumers (C++ function), 854
eprosima::fastdds::dds::Log::Context (C++ struct), 854
eprosima::fastdds::dds::Log::Entry (C++ struct), 854
eprosima::fastdds::dds::Log::Flush (C++ function), 854
eprosima::fastdds::dds::Log::GetVerbosity (C++ function), 854
eprosima::fastdds::dds::Log::KillThread (C++ function), 854
eprosima::fastdds::dds::Log::Kind (C++ enum), 853
eprosima::fastdds::dds::Log::Kind::Error (C++ enumerator), 853
eprosima::fastdds::dds::Log::Kind::Info (C++ enumerator), 853
eprosima::fastdds::dds::Log::Kind::Warning (C++ enumerator), 853
eprosima::fastdds::dds::Log::QueueLog (C++ function), 854
eprosima::fastdds::dds::Log::RegisterConsumer (C++ function), 854
eprosima::fastdds::dds::Log::ReportFilenames (C++ function), 854
eprosima::fastdds::dds::Log::ReportFunctions (C++ function), 854
eprosima::fastdds::dds::Log::Reset (C++ function), 854
eprosima::fastdds::dds::Log::SetCategoryFilter (C++ function), 854
eprosima::fastdds::dds::Log::SetErrorStringFilter (C++ function), 854
eprosima::fastdds::dds::Log::SetFilenameFilter (C++ function), 854
eprosima::fastdds::dds::Log::SetThreadConfig (C++ function), 854

eprosima::fastdds::dds::Log::SetVerbosity (C++ function), 854
eprosima::fastdds::dds::LogConsumer (C++ class), 855
eprosima::fastdds::dds::MatchedStatus (C++ struct), 613
eprosima::fastdds::dds::MatchedStatus::~MatchedStatus (C++ function), 614
eprosima::fastdds::dds::MatchedStatus::current_count (C++ member), 614
eprosima::fastdds::dds::MatchedStatus::current_count_change (C++ member), 614
eprosima::fastdds::dds::MatchedStatus::MatchedStatus (C++ function), 614
eprosima::fastdds::dds::MatchedStatus::total_count (C++ member), 614
eprosima::fastdds::dds::MatchedStatus::total_count_change (C++ member), 614
eprosima::fastdds::dds::OfferedDeadlineMissedStatus (C++ type), 614
eprosima::fastdds::dds::OfferedIncompatibleQosStatus (C++ type), 614
eprosima::fastdds::dds::OStreamConsumer (C++ class), 855
eprosima::fastdds::dds::OwnershipQosPolicy (C++ class), 592
eprosima::fastdds::dds::OwnershipQosPolicy::~OwnershipQosPolicy (C++ function), 592
eprosima::fastdds::dds::OwnershipQosPolicy::clear (C++ function), 592
eprosima::fastdds::dds::OwnershipQosPolicy::kind (C++ member), 592
eprosima::fastdds::dds::OwnershipQosPolicy::OwnershipQosPolicy (C++ function), 592
eprosima::fastdds::dds::OwnershipQosPolicyKind (C++ enum), 592
eprosima::fastdds::dds::OwnershipQosPolicyKind::EXCLUSIVE_OWNERSHIP_QOS (C++ enumerator), 592
eprosima::fastdds::dds::OwnershipQosPolicyKind::SHARED_OWNERSHIP_QOS (C++ enumerator), 592
eprosima::fastdds::dds::OwnershipStrengthQosPolicy (C++ class), 593
eprosima::fastdds::dds::OwnershipStrengthQosPolicy::~OwnershipStrengthQosPolicy (C++ function), 593
eprosima::fastdds::dds::OwnershipStrengthQosPolicy::clear (C++ function), 593
eprosima::fastdds::dds::OwnershipStrengthQosPolicy::OwnershipStrengthQosPolicy (C++ function), 593
eprosima::fastdds::dds::OwnershipStrengthQosPolicy::value (C++ member), 593
eprosima::fastdds::dds::PARTICIPANT_QOS_DEFAULT (C++ member), 649
eprosima::fastdds::dds::ParticipantResourceLimitsQos (C++ type), 593
eprosima::fastdds::dds::Partition_t (C++ class), 593
eprosima::fastdds::dds::Partition_t::name (C++ function), 594
eprosima::fastdds::dds::Partition_t::Partition_t (C++ function), 594
eprosima::fastdds::dds::Partition_t::size (C++ function), 594
eprosima::fastdds::dds::PartitionQosPolicy (C++ class), 594
eprosima::fastdds::dds::PartitionQosPolicy::~PartitionQosPolicy (C++ function), 594
eprosima::fastdds::dds::PartitionQosPolicy::begin (C++ function), 594
eprosima::fastdds::dds::PartitionQosPolicy::clear (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::const_iterator (C++ class), 595
eprosima::fastdds::dds::PartitionQosPolicy::const_iterator (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::empty (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::end (C++ function), 594
eprosima::fastdds::dds::PartitionQosPolicy::getNames (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::max_size (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::names (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::PartitionQosPolicy (C++ function), 594
eprosima::fastdds::dds::PartitionQosPolicy::push_back (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::set_max_size (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::setNames (C++ function), 595
eprosima::fastdds::dds::PartitionQosPolicy::size (C++ function), 595
eprosima::fastdds::dds::PresentationQosPolicy (C++ class), 596
eprosima::fastdds::dds::PresentationQosPolicy::~PresentationQosPolicy (C++ function), 596
eprosima::fastdds::dds::PresentationQosPolicy::access_scope (C++ member), 596
eprosima::fastdds::dds::PresentationQosPolicy::clear (C++ function), 596
eprosima::fastdds::dds::PresentationQosPolicy::coherent_access (C++ member), 596
eprosima::fastdds::dds::PresentationQosPolicy::ordered_access (C++ member), 596
eprosima::fastdds::dds::PresentationQosPolicy::PresentationQosPolicy (C++ function), 596

eprosima::fastdds::dds::PresentationQosPolicyAccessScopeKind (C++ enum), 597
eprosima::fastdds::dds::PresentationQosPolicyAccessScopeKind::GROUP_PRESENTATION_QOS (C++ enumerator), 597
eprosima::fastdds::dds::PresentationQosPolicyAccessScopeKind::INSTANCE_PRESENTATION_QOS (C++ enumerator), 597
eprosima::fastdds::dds::PresentationQosPolicyAccessScopeKind::TOPIC_PRESENTATION_QOS (C++ enumerator), 597
eprosima::fastdds::dds::PropertyPolicyQos (C++ type), 597
eprosima::fastdds::dds::PublicationMatchedStatus (C++ struct), 614
eprosima::fastdds::dds::PublicationMatchedStatus::last_subscription_handle (C++ member), 615
eprosima::fastdds::dds::Publisher (C++ class), 664
eprosima::fastdds::dds::Publisher::~Publisher (C++ function), 664
eprosima::fastdds::dds::Publisher::begin_coherent_changes (C++ function), 666
eprosima::fastdds::dds::Publisher::copy_from_topic_qos (C++ function), 667
eprosima::fastdds::dds::Publisher::create_datawriter (C++ function), 665
eprosima::fastdds::dds::Publisher::create_datawriter_with_profile (C++ function), 665
eprosima::fastdds::dds::Publisher::delete_contained_entities (C++ function), 666
eprosima::fastdds::dds::Publisher::delete_datawriter (C++ function), 665
eprosima::fastdds::dds::Publisher::enable (C++ function), 664
eprosima::fastdds::dds::Publisher::end_coherent_changes (C++ function), 666
eprosima::fastdds::dds::Publisher::get_datawriter_qos_from_profile (C++ function), 667
eprosima::fastdds::dds::Publisher::get_datawriters (C++ function), 668
eprosima::fastdds::dds::Publisher::get_default_datawriter_qos (C++ function), 667
eprosima::fastdds::dds::Publisher::get_instance_handle (C++ function), 668
eprosima::fastdds::dds::Publisher::get_listener (C++ function), 664
eprosima::fastdds::dds::Publisher::get_participant (C++ function), 666
eprosima::fastdds::dds::Publisher::get_qos (C++ function), 664
eprosima::fastdds::dds::Publisher::has_datawriters (C++ function), 668
eprosima::fastdds::dds::Publisher::lookup_datawriter (C++ function), 665
eprosima::fastdds::dds::Publisher::resume_publications (C++ function), 666
eprosima::fastdds::dds::Publisher::set_default_datawriter_qos (C++ function), 667
eprosima::fastdds::dds::Publisher::set_listener (C++ function), 664
eprosima::fastdds::dds::Publisher::set_qos (C++ function), 664
eprosima::fastdds::dds::Publisher::suspend_publications (C++ function), 666
eprosima::fastdds::dds::Publisher::wait_for_acknowledgments (C++ function), 666
eprosima::fastdds::dds::PUBLISHER_QOS_DEFAULT (C++ member), 670
eprosima::fastdds::dds::PublisherListener (C++ class), 668
eprosima::fastdds::dds::PublisherListener::~PublisherListener (C++ function), 668
eprosima::fastdds::dds::PublisherListener::PublisherListener (C++ function), 668
eprosima::fastdds::dds::PublisherQos (C++ class), 668
eprosima::fastdds::dds::PublisherQos::~PublisherQos (C++ function), 669
eprosima::fastdds::dds::PublisherQos::entity_factory (C++ function), 669
eprosima::fastdds::dds::PublisherQos::group_data (C++ function), 669
eprosima::fastdds::dds::PublisherQos::partition (C++ function), 669
eprosima::fastdds::dds::PublisherQos::presentation (C++ function), 669
eprosima::fastdds::dds::PublisherQos::PublisherQos (C++ function), 669
eprosima::fastdds::dds::PublishModeQosPolicy (C++ class), 597
eprosima::fastdds::dds::PublishModeQosPolicy::clear (C++ function), 597
eprosima::fastdds::dds::PublishModeQosPolicy::flow_control (C++ member), 597
eprosima::fastdds::dds::PublishModeQosPolicy::kind (C++ member), 597
eprosima::fastdds::dds::PublishModeQosPolicyKind (C++ enum), 598
eprosima::fastdds::dds::PublishModeQosPolicyKind::ASYNCHRONOUS_PUBLISH_MODE (C++ enumerator), 598
eprosima::fastdds::dds::PublishModeQosPolicyKind::SYNCHRONOUS_PUBLISH_MODE (C++ enumerator), 598
eprosima::fastdds::dds::QosPolicy (C++ class), 598
eprosima::fastdds::dds::QosPolicy::~QosPolicy (C++ function), 598
eprosima::fastdds::dds::QosPolicy::clear (C++ function), 598
eprosima::fastdds::dds::QosPolicy::hasChanged (C++ member), 599

eprosima::fastdds::dds::QosPolicy::QosPolicy (C++ function), 598
eprosima::fastdds::dds::QosPolicy::send_always (C++ function), 598
eprosima::fastdds::dds::QosPolicyCount (C++ struct), 615
eprosima::fastdds::dds::QosPolicyCount::count (C++ member), 615
eprosima::fastdds::dds::QosPolicyCount::policyId (C++ member), 615
eprosima::fastdds::dds::QosPolicyCount::QosPolicyCount (C++ function), 615
eprosima::fastdds::dds::QosPolicyCountSeq (C++ type), 615
eprosima::fastdds::dds::QosPolicyId_t (C++ enum), 599
eprosima::fastdds::dds::QosPolicyId_t::DATAREPRESENTATION_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::DEADLINE_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::DESTINATIONORDER_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::DISABLEPOSITIVEACKS_QOS_POLICY_ID (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::DURABILITY_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::DURABILITYSERVICE_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::ENTITYFACTORY_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::GROUPDATA_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::HISTORY_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::INVALID_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::LATENCYBUDGET_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::LIFESPAN_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::LIVELINESS_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::NEXT_QOS_POLICY_ID (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::OWNERSHIP_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::OWNERSHIPSTRENGTH_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::PARTICIPANTRESOURCELIMITS_QOS_POLICY_ID (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::PARTITION_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::PRESENTATION_QOS_POLICY_ID (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::PROPERTYPOLICY_QOS (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::PUBLISHMODE_QOS_POL (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::READERDATALIFECYCLE (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::READERRESOURCELIMIT (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::RELIABILITY_QOS_POL (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::RESOURCELIMITS_QOS_ (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::RTPSENDPOINT_QOS_PO (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::RTPSRELIABLEREADER_ (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::RTPSRELIABLEWRITER_ (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::TIMEBASEDFILTER_QOS (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::TOPICDATA_QOS_POLIC (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::TRANSPORTCONFIG_QOS (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::TRANSPORTPRIORITY_Q (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::TYPECONSISTENCY_QOS (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::TYPECONSISTENCYENFO (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::USERDATA_QOS_POLICY (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::WIREPROTOCOLCONFIG_ (C++ enumerator), 600
eprosima::fastdds::dds::QosPolicyId_t::WRITERDATALIFECYCLE (C++ enumerator), 599
eprosima::fastdds::dds::QosPolicyId_t::WRITERRESOURCELIMIT (C++ enumerator), 600
eprosima::fastdds::dds::ReadCondition (C++ class), 693
eprosima::fastdds::dds::ReadCondition::get_datareader (C++ function), 694
eprosima::fastdds::dds::ReadCondition::get_instance_state_ (C++ function), 694
eprosima::fastdds::dds::ReadCondition::get_sample_state_ma (C++ function), 694
eprosima::fastdds::dds::ReadCondition::get_trigger_value (C++ function), 694
eprosima::fastdds::dds::ReadCondition::get_view_state_mask (C++ function), 694
eprosima::fastdds::dds::ReaderDataLifecycleQosPolicy (C++ class), 600
eprosima::fastdds::dds::ReaderDataLifecycleQosPolicy::~Rea (C++ function), 600

eprosima::fastdds::dds::ReaderDataLifecycleQosPolicy::max_samples_per_instance (C++ member), 601
eprosima::fastdds::dds::ReaderDataLifecycleQosPolicy::max_samples_per_read (C++ member), 601
eprosima::fastdds::dds::ReaderDataLifecycleQosPolicy::ReaderDataLifecycleQosPolicy (C++ function), 600
eprosima::fastdds::dds::ReaderResourceLimitsQos (C++ class), 694
eprosima::fastdds::dds::ReaderResourceLimitsQos::ReaderResourceLimitsQos (C++ function), 694
eprosima::fastdds::dds::ReaderResourceLimitsQos::matched_publication_allocation (C++ member), 694
eprosima::fastdds::dds::ReaderResourceLimitsQos::max_samples_per_read (C++ member), 694
eprosima::fastdds::dds::ReaderResourceLimitsQos::outstanding_reads_allocation (C++ member), 694
eprosima::fastdds::dds::ReaderResourceLimitsQos::ReaderResourceLimitsQos (C++ function), 694
eprosima::fastdds::dds::ReaderResourceLimitsQos::sample_infos_allocation (C++ member), 694
eprosima::fastdds::dds::ReliabilityQosPolicy (C++ class), 601
eprosima::fastdds::dds::ReliabilityQosPolicy::ReliabilityQosPolicy (C++ function), 601
eprosima::fastdds::dds::ReliabilityQosPolicy::clear (C++ function), 601
eprosima::fastdds::dds::ReliabilityQosPolicy::kind (C++ member), 601
eprosima::fastdds::dds::ReliabilityQosPolicy::max_blocking_time (C++ member), 601
eprosima::fastdds::dds::ReliabilityQosPolicy::operator= (C++ function), 601
eprosima::fastdds::dds::ReliabilityQosPolicyKind (C++ enum), 602
eprosima::fastdds::dds::ReliabilityQosPolicyKind::BEST_EFFORT_RELIABILITY_QOS (C++ enumerator), 602
eprosima::fastdds::dds::ReliabilityQosPolicyKind::RELIABLE_RELIABILITY_QOS (C++ enumerator), 602
eprosima::fastdds::dds::RequestedDeadlineMissedStatus (C++ type), 615
eprosima::fastdds::dds::RequestedIncompatibleQosStatus (C++ type), 616
eprosima::fastdds::dds::ResourceLimitsQosPolicy (C++ class), 602
eprosima::fastdds::dds::ResourceLimitsQosPolicy::ResourceLimitsQosPolicy (C++ function), 602
eprosima::fastdds::dds::ResourceLimitsQosPolicy::allocated_samples (C++ member), 603
eprosima::fastdds::dds::ResourceLimitsQosPolicy::clear (C++ function), 602
eprosima::fastdds::dds::ResourceLimitsQosPolicy::extra_samples (C++ member), 603
eprosima::fastdds::dds::ResourceLimitsQosPolicy::max_instances (C++ member), 603
eprosima::fastdds::dds::ResourceLimitsQosPolicy::max_samples (C++ member), 603
eprosima::fastdds::dds::ResourceLimitsQosPolicy::max_samples_per_instance (C++ member), 603
eprosima::fastdds::dds::ResourceLimitsQosPolicy::operator= (C++ function), 603
eprosima::fastdds::dds::RTPSEndpointQos (C++ class), 603
eprosima::fastdds::dds::RTPSEndpointQos::entity_id (C++ member), 604
eprosima::fastdds::dds::RTPSEndpointQos::external_unicast_locators (C++ member), 604
eprosima::fastdds::dds::RTPSEndpointQos::history_memory_policy (C++ member), 604
eprosima::fastdds::dds::RTPSEndpointQos::ignore_non_matching_locators (C++ member), 604
eprosima::fastdds::dds::RTPSEndpointQos::multicast_locator_list (C++ member), 604
eprosima::fastdds::dds::RTPSEndpointQos::remote_locator_list (C++ member), 604
eprosima::fastdds::dds::RTPSEndpointQos::unicast_locator_list (C++ member), 604
eprosima::fastdds::dds::RTPSEndpointQos::user_defined_id (C++ member), 604
eprosima::fastdds::dds::RTPSReliableReaderQos (C++ class), 695
eprosima::fastdds::dds::RTPSReliableReaderQos::~RTPSReliableReaderQos (C++ function), 695
eprosima::fastdds::dds::RTPSReliableReaderQos::disable_positive_ACKs (C++ member), 695
eprosima::fastdds::dds::RTPSReliableReaderQos::RTPSReliableReaderQos (C++ function), 695
eprosima::fastdds::dds::RTPSReliableReaderQos::times (C++ member), 695
eprosima::fastdds::dds::RTPSReliableWriterQos (C++ class), 670
eprosima::fastdds::dds::RTPSReliableWriterQos::~RTPSReliableWriterQos (C++ function), 670
eprosima::fastdds::dds::RTPSReliableWriterQos::disable_heartbeat_piggyback (C++ member), 670
eprosima::fastdds::dds::RTPSReliableWriterQos::disable_positive_ACKs (C++ member), 670
eprosima::fastdds::dds::RTPSReliableWriterQos::RTPSReliableWriterQos (C++ function), 670
eprosima::fastdds::dds::RTPSReliableWriterQos::times (C++ member), 670
eprosima::fastdds::dds::SampleInfo (C++ struct), 695
eprosima::fastdds::dds::SampleInfo::absolute_generation_rank (C++ member), 696
eprosima::fastdds::dds::SampleInfo::disposed_generation_count (C++ member), 695
eprosima::fastdds::dds::SampleInfo::generation_rank (C++ member), 696

eprosima::fastdds::dds::SampleInfo::instance_handle (C++ member), 696
eprosima::fastdds::dds::SampleInfo::instance_state (C++ member), 695
eprosima::fastdds::dds::SampleInfo::no_writers_generation_count (C++ member), 696
eprosima::fastdds::dds::SampleInfo::publication_handle (C++ member), 696
eprosima::fastdds::dds::SampleInfo::reception_timestamp (C++ member), 696
eprosima::fastdds::dds::SampleInfo::related_sample_identity (C++ member), 696
eprosima::fastdds::dds::SampleInfo::sample_identity (C++ member), 696
eprosima::fastdds::dds::SampleInfo::sample_rank (C++ member), 696
eprosima::fastdds::dds::SampleInfo::sample_state (C++ member), 695
eprosima::fastdds::dds::SampleInfo::source_timestamp (C++ member), 696
eprosima::fastdds::dds::SampleInfo::valid_data (C++ member), 696
eprosima::fastdds::dds::SampleInfo::view_state (C++ member), 695
eprosima::fastdds::dds::SampleLostStatus (C++ type), 616
eprosima::fastdds::dds::SampleRejectedStatus (C++ struct), 616
eprosima::fastdds::dds::SampleRejectedStatus::last_instance_handle (C++ member), 616
eprosima::fastdds::dds::SampleRejectedStatus::last_reason (C++ member), 616
eprosima::fastdds::dds::SampleRejectedStatus::total_count (C++ member), 616
eprosima::fastdds::dds::SampleRejectedStatus::total_count_change (C++ member), 616
eprosima::fastdds::dds::SampleRejectedStatusKind (C++ enum), 617
eprosima::fastdds::dds::SampleRejectedStatusKind::NOT_REJECTED (C++ enumerator), 617
eprosima::fastdds::dds::SampleRejectedStatusKind::REJECTED_BY_INSTANCES_LIMIT (C++ enumerator), 617
eprosima::fastdds::dds::SampleRejectedStatusKind::REJECTED_BY_SAMPLES_LIMIT (C++ enumerator), 617
eprosima::fastdds::dds::SampleRejectedStatusKind::REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT (C++ enumerator), 617
eprosima::fastdds::dds::SampleStateKind (C++ enum), 696
eprosima::fastdds::dds::SampleStateKind::NOT_READ_SAMPLE_STATE (C++ enumerator), 697
eprosima::fastdds::dds::SampleStateKind::READ_SAMPLE_STATE (C++ enumerator), 697
eprosima::fastdds::dds::StackAllocatedSequence (C++ struct), 626
eprosima::fastdds::dds::StatusCondition (C++ class), 620
eprosima::fastdds::dds::StatusCondition::get_enabled_statuses (C++ function), 621
eprosima::fastdds::dds::StatusCondition::get_entity (C++ function), 621
eprosima::fastdds::dds::StatusCondition::get_trigger_value (C++ function), 620
eprosima::fastdds::dds::StatusCondition::set_enabled_statuses (C++ function), 620
eprosima::fastdds::dds::StatusMask (C++ class), 617
eprosima::fastdds::dds::StatusMask::all (C++ function), 618
eprosima::fastdds::dds::StatusMask::data_available (C++ function), 618
eprosima::fastdds::dds::StatusMask::data_on_readers (C++ function), 618
eprosima::fastdds::dds::StatusMask::inconsistent_topic (C++ function), 618
eprosima::fastdds::dds::StatusMask::is_active (C++ function), 618
eprosima::fastdds::dds::StatusMask::liveliness_changed (C++ function), 619
eprosima::fastdds::dds::StatusMask::liveliness_lost (C++ function), 618
eprosima::fastdds::dds::StatusMask::MaskType (C++ type), 617
eprosima::fastdds::dds::StatusMask::none (C++ function), 618
eprosima::fastdds::dds::StatusMask::offered_deadline_missed (C++ function), 618
eprosima::fastdds::dds::StatusMask::offered_incompatible_qos (C++ function), 618
eprosima::fastdds::dds::StatusMask::operator>> (C++ function), 617
eprosima::fastdds::dds::StatusMask::operator<< (C++ function), 617
eprosima::fastdds::dds::StatusMask::publication_matched (C++ function), 619
eprosima::fastdds::dds::StatusMask::requested_deadline_missed (C++ function), 618
eprosima::fastdds::dds::StatusMask::requested_incompatible_qos (C++ function), 618
eprosima::fastdds::dds::StatusMask::sample_lost (C++ function), 618
eprosima::fastdds::dds::StatusMask::sample_rejected (C++ function), 618
eprosima::fastdds::dds::StatusMask::StatusMask (C++ function), 617
eprosima::fastdds::dds::StatusMask::subscription_matched (C++ function), 619
eprosima::fastdds::dds::StdoutConsumer (C++ class), 856

eprosima::fastdds::dds::StdoutErrConsumer (C++ class), 856
eprosima::fastdds::dds::StdoutErrConsumer::stderr_threshold (C++ function), 856
eprosima::fastdds::dds::StdoutErrConsumer::STDERR_THRESHOLD_DEFAULT (C++ member), 856
eprosima::fastdds::dds::Subscriber (C++ class), 697
eprosima::fastdds::dds::Subscriber::~Subscriber (C++ function), 697
eprosima::fastdds::dds::Subscriber::begin_access (C++ function), 699
eprosima::fastdds::dds::Subscriber::copy_from_topic_qos (C++ function), 701
eprosima::fastdds::dds::Subscriber::create_datareader (C++ function), 698
eprosima::fastdds::dds::Subscriber::create_datareader_with_profile (C++ function), 698
eprosima::fastdds::dds::Subscriber::delete_contained_entities (C++ function), 700
eprosima::fastdds::dds::Subscriber::delete_datareader (C++ function), 698
eprosima::fastdds::dds::Subscriber::enable (C++ function), 697
eprosima::fastdds::dds::Subscriber::end_access (C++ function), 699
eprosima::fastdds::dds::Subscriber::get_datareader_qos_from_profile (C++ function), 701
eprosima::fastdds::dds::Subscriber::get_datareaders (C++ function), 699
eprosima::fastdds::dds::Subscriber::get_default_datareader_qos (C++ function), 700
eprosima::fastdds::dds::Subscriber::get_instance_handle (C++ function), 701
eprosima::fastdds::dds::Subscriber::get_listener (C++ function), 697
eprosima::fastdds::dds::Subscriber::get_participant (C++ function), 701
eprosima::fastdds::dds::Subscriber::get_qos (C++ function), 697
eprosima::fastdds::dds::Subscriber::has_datareaders (C++ function), 699
eprosima::fastdds::dds::Subscriber::lookup_datareader (C++ function), 699
eprosima::fastdds::dds::Subscriber::notify_datareaders (C++ function), 700
eprosima::fastdds::dds::Subscriber::set_default_datareader_qos (C++ function), 700
eprosima::fastdds::dds::Subscriber::set_listener (C++ function), 697, 698
eprosima::fastdds::dds::Subscriber::set_qos (C++ function), 697
eprosima::fastdds::dds::SUBSCRIBER_QOS_DEFAULT (C++ member), 703
eprosima::fastdds::dds::SubscriberListener (C++ class), 701
eprosima::fastdds::dds::SubscriberListener::~SubscriberListener (C++ function), 702
eprosima::fastdds::dds::SubscriberListener::on_data_on_readers (C++ function), 702
eprosima::fastdds::dds::SubscriberListener::SubscriberListener (C++ function), 702
eprosima::fastdds::dds::SubscriberQos (C++ class), 702
eprosima::fastdds::dds::SubscriberQos::~SubscriberQos (C++ function), 702
eprosima::fastdds::dds::SubscriberQos::entity_factory (C++ function), 703
eprosima::fastdds::dds::SubscriberQos::group_data (C++ function), 702, 703
eprosima::fastdds::dds::SubscriberQos::partition (C++ function), 702
eprosima::fastdds::dds::SubscriberQos::presentation (C++ function), 702
eprosima::fastdds::dds::SubscriberQos::SubscriberQos (C++ function), 702
eprosima::fastdds::dds::SubscriptionMatchedStatus (C++ struct), 619
eprosima::fastdds::dds::SubscriptionMatchedStatus::last_pu (C++ member), 619
eprosima::fastdds::dds::TimeBasedFilterQosPolicy (C++ class), 604
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::~TimeBas (C++ function), 605
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::clear (C++ function), 605
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::minimum_ (C++ member), 605
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::TimeBase (C++ function), 605
eprosima::fastdds::dds::Topic (C++ class), 711
eprosima::fastdds::dds::Topic::get_inconsistent_topic_stat (C++ function), 711
eprosima::fastdds::dds::Topic::get_listener (C++ function), 712
eprosima::fastdds::dds::Topic::get_participant (C++ function), 711
eprosima::fastdds::dds::Topic::get_qos (C++ function), 711
eprosima::fastdds::dds::Topic::set_listener (C++ function), 712
eprosima::fastdds::dds::Topic::set_qos (C++ function), 711
eprosima::fastdds::dds::TOPIC_QOS_DEFAULT (C++ member), 718
eprosima::fastdds::dds::TopicDataQosPolicy (C++ class), 605
eprosima::fastdds::dds::TopicDataType (C++

class), 704
eprosima::fastdds::dds::TopicDataType::~TopicDataType (*C++ function*), 705
eprosima::fastdds::dds::TopicDataType::auto_fill_type_information (*C++ function*), 706
eprosima::fastdds::dds::TopicDataType::auto_fill_type_object (*C++ function*), 706
eprosima::fastdds::dds::TopicDataType::constructSample (*C++ function*), 707
eprosima::fastdds::dds::TopicDataType::createData (*C++ function*), 706
eprosima::fastdds::dds::TopicDataType::deleteData (*C++ function*), 706
eprosima::fastdds::dds::TopicDataType::deserialize (*C++ function*), 705
eprosima::fastdds::dds::TopicDataType::getKey (*C++ function*), 706
eprosima::fastdds::dds::TopicDataType::getName (*C++ function*), 706
eprosima::fastdds::dds::TopicDataType::getSerializedSizeProvider (*C++ function*), 705
eprosima::fastdds::dds::TopicDataType::is_bounded (*C++ function*), 707
eprosima::fastdds::dds::TopicDataType::is_plain (*C++ function*), 707
eprosima::fastdds::dds::TopicDataType::m_isGetKeyDefined (*C++ member*), 707
eprosima::fastdds::dds::TopicDataType::m_typeSize (*C++ member*), 707
eprosima::fastdds::dds::TopicDataType::serialize (*C++ function*), 705
eprosima::fastdds::dds::TopicDataType::setName (*C++ function*), 706
eprosima::fastdds::dds::TopicDataType::TopicDataType (*C++ function*), 705
eprosima::fastdds::dds::TopicDataType::type_identifier (*C++ function*), 706, 707
eprosima::fastdds::dds::TopicDataType::type_information (*C++ function*), 707
eprosima::fastdds::dds::TopicDataType::type_object (*C++ function*), 707
eprosima::fastdds::dds::TopicDescription (*C++ class*), 710
eprosima::fastdds::dds::TopicDescription::get_name (*C++ function*), 711
eprosima::fastdds::dds::TopicDescription::get_participant (*C++ function*), 711
eprosima::fastdds::dds::TopicDescription::get_type_name (*C++ function*), 711
eprosima::fastdds::dds::TopicListener (*C++ class*), 715
eprosima::fastdds::dds::TopicListener::~TopicListener (*C++ function*), 715
eprosima::fastdds::dds::TopicListener::on_inconsistent_topic (*C++ function*), 715
eprosima::fastdds::dds::TopicListener::TopicListener (*C++ function*), 715
eprosima::fastdds::dds::TopicQos (*C++ class*), 715
eprosima::fastdds::dds::TopicQos::deadline (*C++ function*), 716
eprosima::fastdds::dds::TopicQos::destination_order (*C++ function*), 717
eprosima::fastdds::dds::TopicQos::durability (*C++ function*), 716
eprosima::fastdds::dds::TopicQos::durability_service (*C++ function*), 716
eprosima::fastdds::dds::TopicQos::history (*C++ function*), 717
eprosima::fastdds::dds::TopicQos::latency_budget (*C++ function*), 716
eprosima::fastdds::dds::TopicQos::lifespan (*C++ function*), 718
eprosima::fastdds::dds::TopicQos::liveliness (*C++ function*), 716, 717
eprosima::fastdds::dds::TopicQos::ownership (*C++ function*), 718
eprosima::fastdds::dds::TopicQos::reliability (*C++ function*), 717
eprosima::fastdds::dds::TopicQos::representation (*C++ function*), 718
eprosima::fastdds::dds::TopicQos::resource_limits (*C++ function*), 717, 718
eprosima::fastdds::dds::TopicQos::topic_data (*C++ function*), 715
eprosima::fastdds::dds::TopicQos::TopicQos (*C++ function*), 715
eprosima::fastdds::dds::TopicQos::transport_priority (*C++ function*), 718
eprosima::fastdds::dds::TransportConfigQos (*C++ class*), 605
eprosima::fastdds::dds::TransportConfigQos::~TransportConfigQos (*C++ function*), 605
eprosima::fastdds::dds::TransportConfigQos::builtin_transports (*C++ member*), 606
eprosima::fastdds::dds::TransportConfigQos::clear (*C++ function*), 605
eprosima::fastdds::dds::TransportConfigQos::listen_socket_buffer_size (*C++ member*), 606
eprosima::fastdds::dds::TransportConfigQos::max_msg_size_no_frag (*C++ member*), 606
eprosima::fastdds::dds::TransportConfigQos::netmask_filter (*C++ member*), 606
eprosima::fastdds::dds::TransportConfigQos::send_socket_buffer_size (*C++ member*), 606
eprosima::fastdds::dds::TransportConfigQos::TransportConfigQos (*C++ function*), 605
eprosima::fastdds::dds::TransportConfigQos::use_builtin_transports

(C++ member), 606
eprosima::fastdds::dds::TransportConfigQos::user_transports (C++ member), 606
eprosima::fastdds::dds::TransportPriorityQosPolicy (C++ class), 606
eprosima::fastdds::dds::TransportPriorityQosPolicy::TransportPriorityQosPolicy (C++ function), 607
eprosima::fastdds::dds::TransportPriorityQosPolicy::clear (C++ function), 607
eprosima::fastdds::dds::TransportPriorityQosPolicy::operator= (C++ function), 607
eprosima::fastdds::dds::TransportPriorityQosPolicy::value (C++ member), 607
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy (C++ class), 607
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::TypeConsistencyEnforcementQosPolicy (C++ function), 607
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::clear (C++ function), 607
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::ignore_type_values (C++ member), 608
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::ignore_type_names (C++ member), 608
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::ignore_sequence_bounds (C++ member), 607
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::ignore_string_bounds (C++ member), 608
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::ignore_member_names (C++ member), 607
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::prevent_type_widening (C++ member), 608
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::operator= (C++ function), 607
eprosima::fastdds::dds::TypeConsistencyKind (C++ enum), 608
eprosima::fastdds::dds::TypeConsistencyKind::ALLOW_TYPE_COERCION (C++ enumerator), 608
eprosima::fastdds::dds::TypeConsistencyKind::DISALLOW_TYPE_COERCION (C++ enumerator), 608
eprosima::fastdds::dds::TypeConsistencyQos (C++ class), 703
eprosima::fastdds::dds::TypeConsistencyQos::~TypeConsistencyQos (C++ function), 703
eprosima::fastdds::dds::TypeConsistencyQos::clear (C++ function), 703
eprosima::fastdds::dds::TypeConsistencyQos::representation (C++ member), 704
eprosima::fastdds::dds::TypeConsistencyQos::type_consistency (C++ member), 704
eprosima::fastdds::dds::TypeConsistencyQos::TypeConsistencyQos (C++ function), 703
eprosima::fastdds::dds::TypeIdV1 (C++ class), 719
eprosima::fastdds::dds::TypeIdV1::~TypeIdV1 (C++ function), 719
eprosima::fastdds::dds::TypeIdV1::clear (C++ function), 719
eprosima::fastdds::dds::TypeIdV1::get (C++ function), 719
eprosima::fastdds::dds::TypeIdV1::m_type_identifier (C++ member), 719
eprosima::fastdds::dds::TypeIdV1::TypeIdV1 (C++ function), 719
eprosima::fastdds::dds::TypeObjectV1 (C++ class), 720
eprosima::fastdds::dds::TypeObjectV1::~TypeObjectV1 (C++ function), 721
eprosima::fastdds::dds::TypeObjectV1::clear (C++ function), 721
eprosima::fastdds::dds::TypeObjectV1::m_type_object (C++ member), 721
eprosima::fastdds::dds::TypeObjectV1::TypeObjectV1 (C++ function), 720, 721
eprosima::fastdds::dds::TypeSupport (C++ class), 708
eprosima::fastdds::dds::TypeSupport::create_data (C++ function), 710
eprosima::fastdds::dds::TypeSupport::delete_data (C++ function), 710
eprosima::fastdds::dds::TypeSupport::deserialize (C++ function), 709
eprosima::fastdds::dds::TypeSupport::empty (C++ function), 710
eprosima::fastdds::dds::TypeSupport::get_serialized_size_p (C++ function), 709
eprosima::fastdds::dds::TypeSupport::get_type_name (C++ function), 709
eprosima::fastdds::dds::TypeSupport::is_bounded (C++ function), 710
eprosima::fastdds::dds::TypeSupport::is_plain (C++ function), 710
eprosima::fastdds::dds::TypeSupport::operator= (C++ function), 708
eprosima::fastdds::dds::TypeSupport::register_type (C++ function), 708
eprosima::fastdds::dds::TypeSupport::serialize (C++ function), 709
eprosima::fastdds::dds::TypeSupport::TypeSupport (C++ function), 708
eprosima::fastdds::dds::UserDataQosPolicy (C++ class), 609
eprosima::fastdds::dds::ViewStateKind (C++ enum), 704
eprosima::fastdds::dds::ViewStateKind::NEW_VIEW_STATE

(C++ enumerator), 704
eprosima::fastdds::dds::ViewStateKind::NOT_NEW_VIEW_STATE (C++ enumerator), 704
eprosima::fastdds::dds::WaitSet (C++ class), 621
eprosima::fastdds::dds::WaitSet::attach_condition (C++ function), 621
eprosima::fastdds::dds::WaitSet::detach_condition (C++ function), 621
eprosima::fastdds::dds::WaitSet::get_conditions (C++ function), 621
eprosima::fastdds::dds::WaitSet::wait (C++ function), 621
eprosima::fastdds::dds::WireProtocolConfigQos (C++ class), 609
eprosima::fastdds::dds::WireProtocolConfigQos::eProsima_participant_id (C++ function), 609
eprosima::fastdds::dds::WireProtocolConfigQos::eprosima (C++ member), 609
eprosima::fastdds::dds::WireProtocolConfigQos::eprosima (C++ function), 609
eprosima::fastdds::dds::WireProtocolConfigQos::default_unicast_locator_list (C++ member), 610
eprosima::fastdds::dds::WireProtocolConfigQos::default_unicast_locator_list (C++ member), 610
eprosima::fastdds::dds::WireProtocolConfigQos::default_unicast_locator_list (C++ member), 610
eprosima::fastdds::dds::WireProtocolConfigQos::multicast_locator_list (C++ member), 610
eprosima::fastdds::dds::WireProtocolConfigQos::participant_id (C++ member), 609
eprosima::fastdds::dds::WireProtocolConfigQos::port (C++ member), 609
eprosima::fastdds::dds::WireProtocolConfigQos::prefix (C++ member), 609
eprosima::fastdds::dds::WireProtocolConfigQos::throughput_controller (C++ member), 609
eprosima::fastdds::dds::WireProtocolConfigQos::WireProtocolConfigQos (C++ function), 609
eprosima::fastdds::dds::WriterDataLifecycleQosPolicy (C++ class), 610
eprosima::fastdds::dds::WriterDataLifecycleQosPolicy::WriterDataLifecycleQosPolicy (C++ function), 610
eprosima::fastdds::dds::WriterDataLifecycleQosPolicy::autodispose_unregistered_instances (C++ member), 610
eprosima::fastdds::dds::WriterDataLifecycleQosPolicy::WriterDataLifecycleQosPolicy (C++ function), 610
eprosima::fastdds::dds::WriterResourceLimitsQos (C++ class), 611
eprosima::fastdds::dds::WriterResourceLimitsQos::WriterResourceLimitsQos (C++ function), 611
eprosima::fastdds::dds::WriterResourceLimitsQos::matched_subscriber_allocation (C++ member), 611
eprosima::fastdds::dds::WriterResourceLimitsQos::reader_filters_allocation (C++ member), 611
eprosima::fastdds::dds::WriterResourceLimitsQos::WriterResourceLimitsQos (C++ function), 611
eprosima::fastdds::dds::xtypes::TypeInformation (C++ class), 719
eprosima::fastdds::dds::xtypes::TypeInformation::~TypeInformation (C++ function), 720
eprosima::fastdds::dds::xtypes::TypeInformation::assigned (C++ function), 720
eprosima::fastdds::dds::xtypes::TypeInformation::clear (C++ function), 720
eprosima::fastdds::dds::xtypes::TypeInformation::type_information (C++ member), 720
eprosima::fastdds::dds::xtypes::TypeInformation::TypeInformation (C++ function), 720
eprosima::fastdds::rtps::c_VendorId_eProsima (C++ member), 781
eprosima::fastdds::rtps::c_VendorId_Unknown (C++ member), 781
eprosima::fastdds::rtps::ChainingTransport (C++ class), 835
eprosima::fastdds::rtps::ChainingTransport::~ChainingTransport (C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::AddDefaultOutputLocator (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::ChainingTransport (C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::CloseInputChannel (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::configureInitialPeerLocator (C++ function), 837
eprosima::fastdds::rtps::ChainingTransport::DoInputLocatorsMatch (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::fillMetatrafficMulticastLocator (C++ function), 837
eprosima::fastdds::rtps::ChainingTransport::fillMetatrafficUnicastLocator (C++ function), 837
eprosima::fastdds::rtps::ChainingTransport::fillUnicastLocator (C++ function), 837
eprosima::fastdds::rtps::ChainingTransport::getDefaultMetatrafficMulticastLocators (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::getDefaultMetatrafficUnicastLocators (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::getDefaultUnicastLocators (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::init (C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::is_local_locator (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::is_localhost_allowed (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::is_locator_allowed (C++ function), 838
eprosima::fastdds::rtps::ChainingTransport::IsInputChannelOpen

(C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::IsInputSupported (C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::max_message_size (C++ function), 837
eprosima::fastdds::rtps::ChainingTransport::max_send_buffer_size (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::NormalizeLocator (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::OpenInputChannel (C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::OpenOutputChannel (C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::remove (C++ function), 838
eprosima::fastdds::rtps::ChainingTransport::RemoteToMainLocal (C++ function), 835
eprosima::fastdds::rtps::ChainingTransport::send_resource_list (C++ function), 836
eprosima::fastdds::rtps::ChainingTransport::send (C++ function), 837
eprosima::fastdds::rtps::ChainingTransport::transform_remote_locator (C++ function), 837, 838
eprosima::fastdds::rtps::ChainingTransport::update_network_interfaces (C++ function), 838
eprosima::fastdds::rtps::ChainingTransportDescriptor (C++ struct), 834
eprosima::fastdds::rtps::ChainingTransportDescriptor::low_level_descriptor (C++ member), 834
eprosima::fastdds::rtps::ChainingTransportDescriptor::max_message_size (C++ function), 834
eprosima::fastdds::rtps::ChainingTransportDescriptor::send_buffer_size (C++ function), 834
eprosima::fastdds::rtps::ContentFilterProperty (C++ class), 737
eprosima::fastdds::rtps::ContentFilterProperty::AdditionalConfiguration (C++ struct), 738
eprosima::fastdds::rtps::ContentFilterProperty::AdditionalConfiguration::expression_cha (C++ member), 738
eprosima::fastdds::rtps::ContentFilterProperty::AdditionalConfiguration::expression_par (C++ member), 738
eprosima::fastdds::rtps::ContentFilterProperty::content_filtered_topic_name (C++ member), 737
eprosima::fastdds::rtps::ContentFilterProperty::ContentFilterProperty (C++ function), 737
eprosima::fastdds::rtps::ContentFilterProperty::expression_parameters (C++ member), 737
eprosima::fastdds::rtps::ContentFilterProperty::filter_class_name (C++ member), 737
eprosima::fastdds::rtps::ContentFilterProperty::filter_expression (C++ member), 737
eprosima::fastdds::rtps::ContentFilterProperty::related_topic_name (C++ member), 737
eprosima::fastdds::rtps::DEFAULT_XCDR_VERSION (C++ member), 743
eprosima::fastdds::rtps::ExternalLocators (C++ type), 726
eprosima::fastdds::rtps::FASTDDS_FLOW_CONTROLLER_DEFAULT (C++ member), 786
eprosima::fastdds::rtps::FASTDDS_STATISTICS_FLOW_CONTROLLE (C++ member), 786
eprosima::fastdds::rtps::FlowControllerDescriptor (C++ struct), 786
eprosima::fastdds::rtps::FlowControllerDescriptor::max_byt (C++ member), 786
eprosima::fastdds::rtps::FlowControllerDescriptor::name (C++ member), 786
eprosima::fastdds::rtps::FlowControllerDescriptor::period_ (C++ member), 786
eprosima::fastdds::rtps::FlowControllerDescriptor::schedul (C++ member), 786
eprosima::fastdds::rtps::FlowControllerDescriptor::sender_ (C++ member), 786
eprosima::fastdds::rtps::FlowControllerSchedulerPolicy (C++ enum), 786
eprosima::fastdds::rtps::FlowControllerSchedulerPolicy::FI (C++ enumerator), 786
eprosima::fastdds::rtps::FlowControllerSchedulerPolicy::HI (C++ enumerator), 787
eprosima::fastdds::rtps::FlowControllerSchedulerPolicy::PR (C++ enumerator), 787
eprosima::fastdds::rtps::FlowControllerSchedulerPolicy::RO (C++ enumerator), 787
eprosima::fastdds::rtps::LocatorList (C++ class), 756
eprosima::fastdds::rtps::LocatorList::~LocatorList (C++ function), 756
eprosima::fastdds::rtps::LocatorList::assign (C++ function), 756
eprosima::fastdds::rtps::LocatorList::begin (C++ function), 756
eprosima::fastdds::rtps::LocatorList::capacity (C++ function), 756
eprosima::fastdds::rtps::LocatorList::clear (C++ function), 757
eprosima::fastdds::rtps::LocatorList::end (C++ function), 756
eprosima::fastdds::rtps::LocatorList::erase (C++ function), 757
eprosima::fastdds::rtps::LocatorList::isValid (C++ function), 757
eprosima::fastdds::rtps::LocatorList::LocatorList (C++ function), 756
eprosima::fastdds::rtps::LocatorList::operator= (C++ function), 756
eprosima::fastdds::rtps::LocatorList::operator== (C++ function), 756
eprosima::fastdds::rtps::LocatorList::push_back

(C++ function), 757
eprosima::fastdds::rtps::LocatorList::reserve (C++ function), 756
eprosima::fastdds::rtps::LocatorList::resize (C++ function), 757
eprosima::fastdds::rtps::LocatorList::size (C++ function), 756
eprosima::fastdds::rtps::LocatorList::swap (C++ function), 757
eprosima::fastdds::rtps::LocatorListConstIterator (C++ type), 757
eprosima::fastdds::rtps::LocatorListIterator (C++ type), 757
eprosima::fastdds::rtps::Locators (C++ class), 758
eprosima::fastdds::rtps::Locators::Locators (C++ function), 758
eprosima::fastdds::rtps::Locators::operator!= (C++ function), 759
eprosima::fastdds::rtps::Locators::operator* (C++ function), 759
eprosima::fastdds::rtps::Locators::operator++ (C++ function), 758
eprosima::fastdds::rtps::Locators::operator== (C++ function), 758
eprosima::fastdds::rtps::LocatorsIterator (C++ struct), 758
eprosima::fastdds::rtps::LocatorsIterator::operator!= (C++ function), 758
eprosima::fastdds::rtps::LocatorsIterator::operator* (C++ function), 758
eprosima::fastdds::rtps::LocatorsIterator::operator++ (C++ function), 758
eprosima::fastdds::rtps::LocatorsIterator::operator== (C++ function), 758
eprosima::fastdds::rtps::LocatorWithMask (C++ class), 764
eprosima::fastdds::rtps::LocatorWithMask::mask (C++ function), 765
eprosima::fastdds::rtps::LocatorWithMask::matches (C++ function), 765
eprosima::fastdds::rtps::LocatorWithMask::operator= (C++ function), 765
eprosima::fastdds::rtps::operator>> (C++ function), 760
eprosima::fastdds::rtps::operator<< (C++ function), 760
eprosima::fastdds::rtps::RemoteServerAttributes (C++ class), 731
eprosima::fastdds::rtps::RemoteServerAttributes::guidPrefix (C++ member), 731
eprosima::fastdds::rtps::RemoteServerAttributes::locatorList (C++ member), 731
eprosima::fastdds::rtps::RemoteServerList_t (C++ type), 731
eprosima::fastdds::rtps::s_defaultTTL (C++ member), 834
eprosima::fastdds::rtps::s_IPv4AddressAny (C++ member), 832
eprosima::fastdds::rtps::s_IPv6AddressAny (C++ member), 832
eprosima::fastdds::rtps::s_maximumInitialPeersRange (C++ member), 832
eprosima::fastdds::rtps::s_maximumMessageSize (C++ member), 832
eprosima::fastdds::rtps::s_minimumSocketBuffer (C++ member), 832
eprosima::fastdds::rtps::SendResourceList (C++ type), 832
eprosima::fastdds::rtps::SharedMemTransportDescriptor (C++ struct), 849
eprosima::fastdds::rtps::SharedMemTransportDescriptor::~Sh (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::cre (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::dum (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::hea (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::max (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::min (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::ope (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::ope (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::por (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::rtp (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::seg (C++ function), 850
eprosima::fastdds::rtps::SharedMemTransportDescriptor::Sha (C++ function), 850
eprosima::fastdds::rtps::SocketTransportDescriptor (C++ struct), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::~Socke (C++ function), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::interf (C++ member), 834
eprosima::fastdds::rtps::SocketTransportDescriptor::interf (C++ member), 834
eprosima::fastdds::rtps::SocketTransportDescriptor::interf (C++ member), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::min_se

(C++ function), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::max_message_size
(C++ member), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::operator=
(C++ function), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::receiveBufferSize
(C++ function), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::receiveBufferSize
(C++ member), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::sendBufferSize
(C++ member), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::SocketTransportDescriptor
(C++ function), 833
eprosima::fastdds::rtps::SocketTransportDescriptor::TTL
(C++ member), 834
eprosima::fastdds::rtps::TCPTransportDescriptor
(C++ struct), 842
eprosima::fastdds::rtps::TCPTransportDescriptor::~TCPTransportDescriptor
(C++ function), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::accept_thread
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::add_listener_port
(C++ function), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::apply_security
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::calculate_crc
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::check_crc
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::enable_tcp_nodelay
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::keep_alive_frequency_ms
(C++ member), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::keep_alive_thread
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::keep_alive_timeout_ms
(C++ member), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::listening_ports
(C++ member), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::logical_port_increment
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::logical_port_range
(C++ member), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::max_logical_port
(C++ member), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::max_msg_size_no_delay
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::operator=
(C++ function), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::operator==
(C++ function), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::tcp_negotiation_timeout
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::TCPTransportDescriptor
(C++ function), 843
eprosima::fastdds::rtps::TCPTransportDescriptor::tls_config
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig
(C++ struct), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::~TLSConfig
(C++ function), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::add_verify_file
(C++ function), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::cert_chain_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::dh_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::handshake_role
(C++ function), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::operator=
(C++ function), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::password
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::private_key_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::rsa_private_key_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::set_default_verify_path
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSConfig
(C++ function), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::tmp_dh_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::trust_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::verify_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::verify_mode
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode
(C++ enum), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::ALLOW_UNTRUSTED
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::VERIFY_NONE
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::VERIFY_PEER
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::version
(C++ member), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSVersion
(C++ enum), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSVersion::TLSv1_2
(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSVersion::TLSv1_3
(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSVersion::TLSv1_default
(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSVersion::TLSv1_default_secure
(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::use_tls
(C++ member), 844
eprosima::fastdds::rtps::TCPTransportDescriptor::tls_config
(C++ member), 844

(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSOptions::TLSV1_3
(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSOptions::TLSV1_3_RFC
(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSOptions::TLSV1_RFC
(C++ enumerator), 845
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode
(C++ enum), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::VERIFY_CLIENT_CERT
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::VERIFY_CLIENT_CERT_STRICT
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::VERIFY_FAIL_IF_NO_PEER_CERT_CERT
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::VERIFY_NONE
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::VerifyMode::VERIFY_PEER
(C++ enumerator), 846
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::add_dh_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::add_verify_path
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::add_verify_file
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::add_verify_mode
(C++ member), 847
eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::add_verify_paths
(C++ member), 847
eprosima::fastdds::rtps::TCPv4TransportDescriptor
(C++ struct), 848
eprosima::fastdds::rtps::TCPv4TransportDescriptor::TCPv4TransportDescriptor
(C++ function), 848
eprosima::fastdds::rtps::TCPv4TransportDescriptor::create_fastdds_port
(C++ function), 848
eprosima::fastdds::rtps::TCPv4TransportDescriptor::WAN_address
(C++ function), 848
eprosima::fastdds::rtps::TCPv4TransportDescriptor::operator=
(C++ function), 848
eprosima::fastdds::rtps::TCPv4TransportDescriptor::set_WAN_address
(C++ function), 848
eprosima::fastdds::rtps::TCPv4TransportDescriptor::wan_addr
(C++ member), 848
eprosima::fastdds::rtps::TCPv6TransportDescriptor
(C++ struct), 849
eprosima::fastdds::rtps::TCPv6TransportDescriptor::TCPv6TransportDescriptor
(C++ function), 849
eprosima::fastdds::rtps::TCPv6TransportDescriptor::create_fastdds_port
(C++ function), 849
eprosima::fastdds::rtps::TCPv6TransportDescriptor::operator=
(C++ function), 849
eprosima::fastdds::rtps::TCPv6TransportDescriptor::set_WAN_address
(C++ function), 849
eprosima::fastdds::rtps::TCPv6TransportDescriptor::wan_addr
(C++ member), 849
eprosima::fastdds::rtps::TLSConfig::TLSOptions::TLSV1_3
(C++ function), 849
eprosima::fastdds::rtps::TLSConfig::TLSOptions::TLSV1_3_RFC
(C++ function), 849
eprosima::fastdds::rtps::TLSConfig::TLSOptions::TLSV1_RFC
(C++ function), 849
eprosima::fastdds::rtps::TLSConfig::TLSOptions::testUDPv4TransportDescriptor
(C++ struct), 841
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::~test_UDPv4TransportDescriptor
(C++ function), 841
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::create_fastdds_port
(C++ function), 841
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::default_send_port
(C++ type), 841
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::dh_file
(C++ member), 842
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::interfaceWhiteList
(C++ member), 842
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::max_message_size
(C++ member), 841
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::max_send_buffer_size
(C++ member), 842
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::operator=
(C++ function), 841
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::send_buffer_size
(C++ member), 842
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::set_send_buffer_size
(C++ function), 841
eprosima::fastdds::rtps::TLSConfig::VerifyMode::test_UDPv4TransportDescriptor::test_UDPv4TransportDescriptor
(C++ function), 841
eprosima::fastdds::rtps::TransportDescriptorInterface

(C++ *struct*), 827
eprosima::fastdds::rtps::TransportDescriptorInterface::~TransportDescriptorInterface
(C++ *function*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::is_local_locator_allowed
(C++ *function*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::max_message_size
(C++ *function*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::max_message_size_
(C++ *member*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::max_send_buffer_size
(C++ *function*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::operator=
(C++ *function*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::operator==
(C++ *function*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::TransportDescriptorInterface
(C++ *function*), 828
eprosima::fastdds::rtps::TransportDescriptorInterface::update_send_buffer_size
(C++ *function*), 828
eprosima::fastdds::rtps::TransportInterface
(C++ *class*), 829
eprosima::fastdds::rtps::TransportInterface::~TransportInterface
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::AddDefaultOutputLocator
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::CloseInputChannel
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::CloseOutputChannel
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::configureInitialPeerLocator
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::DoInputLocatorsMatch
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::fillMetatrafficMulticastLocator
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::fillMetatrafficUnicastLocator
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::getConfiguration
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::getDefaultMetatrafficMulticastLocators
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::getDefaultMetatrafficUnicastLocators
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::getDefaultUnicastLocators
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::init
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::is_local_locator
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::is_localhost_allowed
(C++ *function*), 832
eprosima::fastdds::rtps::TransportInterface::is_locator_allowed
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::IsInputChannelOpen
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::IsLocatorSupported
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::kind
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::max_recv_buffer_size
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::netmask_filter
(C++ *function*), 832
eprosima::fastdds::rtps::TransportInterface::NormalizeLocator
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::OpenInputChannel
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::OpenOutputChannel
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::OpenOutputChannels
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::operator=
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::RemoteToMainLocal
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::select_locators
(C++ *function*), 830
eprosima::fastdds::rtps::TransportInterface::shutdown
(C++ *function*), 831
eprosima::fastdds::rtps::TransportInterface::transform_remote_locator
(C++ *function*), 829, 831
eprosima::fastdds::rtps::TransportInterface::TransportInterface
(C++ *function*), 829
eprosima::fastdds::rtps::TransportInterface::update_network_interfaces
(C++ *function*), 831
eprosima::fastdds::rtps::TransportReceiverInterface
(C++ *class*), 832
eprosima::fastdds::rtps::TransportReceiverInterface::~TransportReceiverInterface
(C++ *function*), 832
eprosima::fastdds::rtps::TransportReceiverInterface::OnDataReceived
(C++ *function*), 832
eprosima::fastdds::rtps::UDPTransportDescriptor
(C++ *struct*), 839
eprosima::fastdds::rtps::UDPTransportDescriptor::~UDPTransportDescriptor
(C++ *function*), 839
eprosima::fastdds::rtps::UDPTransportDescriptor::m_output_ifname
(C++ *member*), 839
eprosima::fastdds::rtps::UDPTransportDescriptor::non_blocking_send
(C++ *member*), 839
eprosima::fastdds::rtps::UDPTransportDescriptor::operator=

(C++ function), 839
eprosima::fastdds::rtps::UDPTransportDescriptor::operator= (C++ function), 839
eprosima::fastdds::rtps::UDPTransportDescriptor::UDPTransportDescriptor (C++ function), 839
eprosima::fastdds::rtps::UDPv4TransportDescriptor (C++ struct), 840
eprosima::fastdds::rtps::UDPv4TransportDescriptor::UDPv4TransportDescriptor (C++ function), 840
eprosima::fastdds::rtps::UDPv4TransportDescriptor::max_message_size_ (C++ function), 840
eprosima::fastdds::rtps::UDPv4TransportDescriptor::operator= (C++ function), 840
eprosima::fastdds::rtps::UDPv4TransportDescriptor::send_buffer_size_ (C++ function), 840
eprosima::fastdds::rtps::UDPv6TransportDescriptor (C++ struct), 840
eprosima::fastdds::rtps::UDPv6TransportDescriptor::UDPv6TransportDescriptor (C++ function), 840
eprosima::fastdds::rtps::UDPv6TransportDescriptor::max_message_size_ (C++ function), 840
eprosima::fastdds::rtps::UDPv6TransportDescriptor::operator= (C++ function), 840
eprosima::fastdds::rtps::UDPv6TransportDescriptor::send_buffer_size_ (C++ function), 840
eprosima::fastdds::rtps::VendorId_t (C++ type), 783
eprosima::fastdds::statistics::ACKNACK_COUNT_TOPIC (C++ member), 860
eprosima::fastdds::statistics::DATA_COUNT_TOPIC (C++ member), 861
eprosima::fastdds::statistics::dds::DataReaderQos (C++ class), 860
eprosima::fastdds::statistics::dds::DataReaderQos::DataReaderQos (C++ function), 860
eprosima::fastdds::statistics::dds::DataWriterQos (C++ class), 859
eprosima::fastdds::statistics::dds::DataWriterQos::DataWriterQos (C++ function), 859
eprosima::fastdds::statistics::dds::DomainParticipant (C++ class), 856
eprosima::fastdds::statistics::dds::DomainParticipant::disable_monitor_service (C++ function), 857
eprosima::fastdds::statistics::dds::DomainParticipant::disable_statistics_datawriter (C++ function), 857
eprosima::fastdds::statistics::dds::DomainParticipant::enable_monitor_service (C++ function), 857
eprosima::fastdds::statistics::dds::DomainParticipant::enable_statistics_datawriter (C++ function), 856
eprosima::fastdds::statistics::dds::DomainParticipant::enable_statistics_datawriter_with_profile (C++ function), 857
eprosima::fastdds::statistics::dds::DomainParticipant::fill_discovery_data_from_cdr_message (C++ function), 857–859
eprosima::fastdds::statistics::dds::DomainParticipant::narrow (C++ function), 859
eprosima::fastdds::statistics::dds::STATISTICS_DATAREADER_QOS_PROFILE (C++ member), 860
eprosima::fastdds::statistics::dds::STATISTICS_DATAWRITER_QOS_PROFILE (C++ member), 859
eprosima::fastdds::statistics::DISCOVERY_TOPIC (C++ member), 861
eprosima::fastdds::statistics::EDP_PACKETS_TOPIC (C++ member), 861
eprosima::fastdds::statistics::GAP_COUNT_TOPIC (C++ member), 861
eprosima::fastdds::statistics::HEARTBEAT_COUNT_TOPIC (C++ member), 860
eprosima::fastdds::statistics::HISTORY_LATENCY_TOPIC (C++ member), 860
eprosima::fastdds::statistics::NACKFRAG_COUNT_TOPIC (C++ member), 861
eprosima::fastdds::statistics::NETWORK_LATENCY_TOPIC (C++ member), 860
eprosima::fastdds::statistics::PDP_PACKETS_TOPIC (C++ member), 861
eprosima::fastdds::statistics::PHYSICAL_DATA_TOPIC (C++ member), 861
eprosima::fastdds::statistics::PUBLICATION_THROUGHPUT_TOPIC (C++ member), 860
eprosima::fastdds::statistics::RESENT_DATAS_TOPIC (C++ member), 860
eprosima::fastdds::statistics::RTPS_LOST_TOPIC (C++ member), 860
eprosima::fastdds::statistics::RTPS_SENT_TOPIC (C++ member), 860
eprosima::fastdds::statistics::SAMPLE_DATAS_TOPIC (C++ member), 861
eprosima::fastdds::statistics::SUBSCRIPTION_THROUGHPUT_TOPIC (C++ member), 860
eprosima::fastrtps::c_TimeInfinite (C++ member), 774
eprosima::fastrtps::c_TimeInvalid (C++ member), 774
eprosima::fastrtps::c_TimeZero (C++ member), 774
eprosima::fastrtps::Duration_t (C++ type), 775
eprosima::fastrtps::operator!= (C++ function), 778
eprosima::fastrtps::operator+ (C++ function), 778
eprosima::fastrtps::operator== (C++ function), 777
eprosima::fastrtps::operator- (C++ function), 778
eprosima::fastrtps::operator> (C++ function), 778
eprosima::fastrtps::operator>= (C++ function), 778

eprosima::fastrtps::operator< (*C++ function*), 777
eprosima::fastrtps::operator<= (*C++ function*), 777
eprosima::fastrtps::operator<< (*C++ function*), 778
eprosima::fastrtps::rtps::AuthenticatedPeerCredentialToken (*C++ type*), 780
eprosima::fastrtps::rtps::BinaryProperty (*C++ class*), 738
eprosima::fastrtps::rtps::BinaryPropertyHelper (*C++ class*), 738
eprosima::fastrtps::rtps::BinaryPropertySeq (*C++ type*), 738
eprosima::fastrtps::rtps::BuiltinAttributes (*C++ class*), 721
eprosima::fastrtps::rtps::BuiltinAttributes::avoid_builtin_multicast (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinAttributes::discovery_config (*C++ member*), 721
eprosima::fastrtps::rtps::BuiltinAttributes::initialPeersList (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinAttributes::metatrafficExternalUnicastLocatorList (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinAttributes::metatrafficMulticastLocatorList (*C++ member*), 721
eprosima::fastrtps::rtps::BuiltinAttributes::metatrafficUnicastLocatorList (*C++ member*), 721
eprosima::fastrtps::rtps::BuiltinAttributes::metatrafficUnicastPort (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinAttributes::networkConfiguration (*C++ member*), 721
eprosima::fastrtps::rtps::BuiltinAttributes::readerHistoryMemoryPolicy (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinAttributes::readerPayloadSize (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinAttributes::typelookup_config (*C++ member*), 721
eprosima::fastrtps::rtps::BuiltinAttributes::use_WriterLivelinessProtocol (*C++ member*), 721
eprosima::fastrtps::rtps::BuiltinAttributes::writerHistoryMemoryPolicy (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinAttributes::writerPayloadSize (*C++ member*), 722
eprosima::fastrtps::rtps::BuiltinEndpointSet_t (*C++ type*), 781
eprosima::fastrtps::rtps::c_default_RTPSParticipant_qos (*C++ member*), 722
eprosima::fastrtps::rtps::c_EntityId_ReaderLiveliness (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_ReaderLivelinessSecure (*C++ member*), 744
eprosima::fastrtps::rtps::c_EntityId_RTPSParticipant (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_SEDPPubReader (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_SEDPPubWriter (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_SEDPSubReader (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_SEDPSubWriter (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_SPDPReader (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_SPDPWriter (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_TypeLookup_reply_read (*C++ member*), 744
eprosima::fastrtps::rtps::c_EntityId_TypeLookup_reply_writ (*C++ member*), 744
eprosima::fastrtps::rtps::c_EntityId_TypeLookup_request_re (*C++ member*), 744
eprosima::fastrtps::rtps::c_EntityId_TypeLookup_request_wr (*C++ member*), 744
eprosima::fastrtps::rtps::c_EntityId_Unknown (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_WriterLiveliness (*C++ member*), 743
eprosima::fastrtps::rtps::c_EntityId_WriterLivelinessSecur (*C++ member*), 744
eprosima::fastrtps::rtps::c_Guid_Unknown (*C++ member*), 747
eprosima::fastrtps::rtps::c_GuidPrefix_Unknown (*C++ member*), 749
eprosima::fastrtps::rtps::c_InstanceHandle_Unknown (*C++ member*), 751
eprosima::fastrtps::rtps::c_ProtocolVersion (*C++ member*), 781
eprosima::fastrtps::rtps::c_ProtocolVersion_2_0 (*C++ member*), 781
eprosima::fastrtps::rtps::c_ProtocolVersion_2_1 (*C++ member*), 781
eprosima::fastrtps::rtps::c_ProtocolVersion_2_2 (*C++ member*), 781
eprosima::fastrtps::rtps::c_ProtocolVersion_2_3 (*C++ member*), 781
eprosima::fastrtps::rtps::c_SequenceNumber_Unknown (*C++ member*), 769
eprosima::fastrtps::rtps::CacheChange_t (*C++ struct*), 738
eprosima::fastrtps::rtps::CacheChange_t::CacheChange_t (*C++ function*), 739
eprosima::fastrtps::rtps::CacheChange_t::contains_first_fr (*C++ function*), 739
eprosima::fastrtps::rtps::CacheChange_t::copy (*C++ function*), 739
eprosima::fastrtps::rtps::CacheChange_t::copy_not_memcpy (*C++ function*), 739

eprosima::fastrtps::rtps::CacheChange_t::get_missing_fragments (C++ function), 739
eprosima::fastrtps::rtps::CacheChange_t::getFragmentCount (C++ function), 739
eprosima::fastrtps::rtps::CacheChange_t::getFragmentSize (C++ function), 739
eprosima::fastrtps::rtps::CacheChange_t::inline_qos (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::instanceHandle (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::is_fully_assembled (C++ function), 739
eprosima::fastrtps::rtps::CacheChange_t::isRead (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::kind (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::sequenceNumber (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::serializedPayload (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::setFragmentSize (C++ function), 739
eprosima::fastrtps::rtps::CacheChange_t::sourceTimestamp (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::vendorId (C++ member), 740
eprosima::fastrtps::rtps::CacheChange_t::writerGUID (C++ member), 740
eprosima::fastrtps::rtps::CDRMessage_t (C++ struct), 742
eprosima::fastrtps::rtps::CDRMessage_t::buffer (C++ member), 742
eprosima::fastrtps::rtps::CDRMessage_t::CDRMessage_t (C++ function), 742
eprosima::fastrtps::rtps::CDRMessage_t::length (C++ member), 742
eprosima::fastrtps::rtps::CDRMessage_t::max_size (C++ member), 742
eprosima::fastrtps::rtps::CDRMessage_t::msg_endian (C++ member), 742
eprosima::fastrtps::rtps::CDRMessage_t::pos (C++ member), 742
eprosima::fastrtps::rtps::CDRMessage_t::reserved_size (C++ member), 742
eprosima::fastrtps::rtps::ChangeForReader_t (C++ class), 740
eprosima::fastrtps::rtps::ChangeForReader_t::getChange (C++ function), 741
eprosima::fastrtps::rtps::ChangeForReaderCmp (C++ struct), 741
eprosima::fastrtps::rtps::ChangeForReaderStatus_t (C++ enum), 741
eprosima::fastrtps::rtps::ChangeForReaderStatus_t::ACKNOWLEDGED (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeForReaderStatus_t::REQUESTED (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeForReaderStatus_t::UNACKNOWLEDGED (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeForReaderStatus_t::UNDERWAY (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeForReaderStatus_t::UNSENT (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeKind_t (C++ enum), 741
eprosima::fastrtps::rtps::ChangeKind_t::ALIVE (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeKind_t::NOT_ALIVE_DISPOSED (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeKind_t::NOT_ALIVE_DISPOSED_UNREGISTERED (C++ enumerator), 741
eprosima::fastrtps::rtps::ChangeKind_t::NOT_ALIVE_UNREGISTERED (C++ enumerator), 741
eprosima::fastrtps::rtps::Count_t (C++ type), 781
eprosima::fastrtps::rtps::DataHolder (C++ class), 780
eprosima::fastrtps::rtps::DataHolderHelper (C++ class), 780
eprosima::fastrtps::rtps::DataHolderSeq (C++ type), 780
eprosima::fastrtps::rtps::DiscoveryProtocol (C++ enum), 722
eprosima::fastrtps::rtps::DiscoveryProtocol::BACKUP (C++ enumerator), 723
eprosima::fastrtps::rtps::DiscoveryProtocol::CLIENT (C++ enumerator), 723
eprosima::fastrtps::rtps::DiscoveryProtocol::EXTERNAL (C++ enumerator), 723
eprosima::fastrtps::rtps::DiscoveryProtocol::NONE (C++ enumerator), 722
eprosima::fastrtps::rtps::DiscoveryProtocol::SERVER (C++ enumerator), 723
eprosima::fastrtps::rtps::DiscoveryProtocol::SIMPLE (C++ enumerator), 722
eprosima::fastrtps::rtps::DiscoveryProtocol::SUPER_CLIENT (C++ enumerator), 723
eprosima::fastrtps::rtps::DiscoverySettings (C++ class), 723
eprosima::fastrtps::rtps::DiscoverySettings::discoveryProtocol (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::discoveryServersList (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::getStaticEndpointXMLFilename (C++ function), 723
eprosima::fastrtps::rtps::DiscoverySettings::ignoreParticipantFlags (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::initial_announcements (C++ member), 724

eprosima::fastrtps::rtps::DiscoverySettings::leaseDuration (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::leaseDuration_announcementperiod (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::m_discoveryServers (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::m_PDPFactory (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::m_simpleEDP (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::setStaticEndpointXMLFilename (C++ function), 723
eprosima::fastrtps::rtps::DiscoverySettings::static_endpoint_xml_config (C++ function), 723, 724
eprosima::fastrtps::rtps::DiscoverySettings::use_SIMPLE_EndpointDiscoveryProtocol (C++ member), 724
eprosima::fastrtps::rtps::DiscoverySettings::use_STATIC_EndpointDiscoveryProtocol (C++ member), 724
eprosima::fastrtps::rtps::DurabilityKind_t (C++ type), 782
eprosima::fastrtps::rtps::Endianness_t (C++ enum), 782
eprosima::fastrtps::rtps::Endianness_t::BIGEND (C++ enumerator), 782
eprosima::fastrtps::rtps::Endianness_t::LITTLEEND (C++ enumerator), 782
eprosima::fastrtps::rtps::Endpoint (C++ class), 785
eprosima::fastrtps::rtps::Endpoint::getAttributes (C++ function), 785
eprosima::fastrtps::rtps::Endpoint::getGuid (C++ function), 785
eprosima::fastrtps::rtps::Endpoint::getMutex (C++ function), 785
eprosima::fastrtps::rtps::EndpointAttributes (C++ class), 725
eprosima::fastrtps::rtps::EndpointAttributes::data_sharing_configuration (C++ function), 725
eprosima::fastrtps::rtps::EndpointAttributes::durabilityKind (C++ member), 725
eprosima::fastrtps::rtps::EndpointAttributes::endpointKind (C++ member), 725
eprosima::fastrtps::rtps::EndpointAttributes::external_unicast_locators (C++ member), 725
eprosima::fastrtps::rtps::EndpointAttributes::getEntityID (C++ function), 725
eprosima::fastrtps::rtps::EndpointAttributes::getUserDefinedID (C++ function), 725
eprosima::fastrtps::rtps::EndpointAttributes::ignore_non_matching_locators (C++ member), 725
eprosima::fastrtps::rtps::EndpointAttributes::multicastLocatorList (C++ member), 726
eprosima::fastrtps::rtps::EndpointAttributes::remoteLocatorList (C++ member), 726
eprosima::fastrtps::rtps::EndpointAttributes::setEntityID (C++ function), 725
eprosima::fastrtps::rtps::EndpointAttributes::setUserDefinedID (C++ function), 725
eprosima::fastrtps::rtps::EndpointAttributes::topicKind (C++ member), 725
eprosima::fastrtps::rtps::EndpointAttributes::unicastLocatorList (C++ member), 726
eprosima::fastrtps::rtps::EndpointKind_t (C++ type), 782
eprosima::fastrtps::rtps::EntityId_t (C++ struct), 745
eprosima::fastrtps::rtps::EntityId_t::cmp (C++ function), 746
eprosima::fastrtps::rtps::EntityId_t::EntityId_t (C++ function), 745
eprosima::fastrtps::rtps::EntityId_t::operator= (C++ function), 745
eprosima::fastrtps::rtps::EntityId_t::operator< (C++ function), 746
eprosima::fastrtps::rtps::EntityId_t::to_uint32 (C++ function), 746
eprosima::fastrtps::rtps::Exception (C++ class), 785
eprosima::fastrtps::rtps::Exception::~Exception (C++ function), 785
eprosima::fastrtps::rtps::Exception::minor (C++ function), 785
eprosima::fastrtps::rtps::Exception::raise (C++ function), 785
eprosima::fastrtps::rtps::Exception::what (C++ function), 785
eprosima::fastrtps::rtps::FragmentNumber_t (C++ type), 747
eprosima::fastrtps::rtps::FragmentNumberSet_t (C++ type), 747
eprosima::fastrtps::rtps::GUID_t (C++ struct), 747
eprosima::fastrtps::rtps::GUID_t::entityId (C++ member), 748
eprosima::fastrtps::rtps::GUID_t::GUID_t (C++ function), 747, 748
eprosima::fastrtps::rtps::GUID_t::guidPrefix (C++ member), 748

eprosima::fastrtps::rtps::GUID_t::is_builtin (C++ function), 748
eprosima::fastrtps::rtps::GUID_t::is_from_this_host (C++ function), 748
eprosima::fastrtps::rtps::GUID_t::is_from_this_process_ (C++ function), 748
eprosima::fastrtps::rtps::GUID_t::is_on_same_host_as (C++ function), 748
eprosima::fastrtps::rtps::GUID_t::is_on_same_process_as (C++ function), 748
eprosima::fastrtps::rtps::GuidPrefix_t (C++ struct), 750
eprosima::fastrtps::rtps::GuidPrefix_t::cmp (C++ function), 751
eprosima::fastrtps::rtps::GuidPrefix_t::GuidPrefix_t (C++ function), 750
eprosima::fastrtps::rtps::GuidPrefix_t::is_from_this_host (C++ function), 750
eprosima::fastrtps::rtps::GuidPrefix_t::is_from_this_process_ (C++ function), 750
eprosima::fastrtps::rtps::GuidPrefix_t::is_on_same_host_as (C++ function), 750
eprosima::fastrtps::rtps::GuidPrefix_t::is_on_same_process_as (C++ function), 750
eprosima::fastrtps::rtps::GuidPrefix_t::operator!= (C++ function), 750
eprosima::fastrtps::rtps::GuidPrefix_t::operator< (C++ function), 750
eprosima::fastrtps::rtps::GuidPrefix_t::operator== (C++ function), 751
eprosima::fastrtps::rtps::History (C++ class), 787
eprosima::fastrtps::rtps::History::changesBegin (C++ function), 789
eprosima::fastrtps::rtps::History::changesEnd (C++ function), 789
eprosima::fastrtps::rtps::History::find_change (C++ function), 789
eprosima::fastrtps::rtps::History::find_change_nts (C++ function), 788
eprosima::fastrtps::rtps::History::get_earliest_change (C++ function), 790
eprosima::fastrtps::rtps::History::get_max_change (C++ function), 789
eprosima::fastrtps::rtps::History::get_min_change (C++ function), 789
eprosima::fastrtps::rtps::History::getHistorySize (C++ function), 788
eprosima::fastrtps::rtps::History::getMutex (C++ function), 790
eprosima::fastrtps::rtps::History::getTypeMaxSerialized (C++ function), 789
eprosima::fastrtps::rtps::History::isFull (C++ function), 788
eprosima::fastrtps::rtps::History::m_att (C++ member), 790
eprosima::fastrtps::rtps::History::matches_change (C++ function), 789
eprosima::fastrtps::rtps::History::release_Cache (C++ function), 787
eprosima::fastrtps::rtps::History::remove_all_changes (C++ function), 788
eprosima::fastrtps::rtps::History::remove_change (C++ function), 788, 789
eprosima::fastrtps::rtps::History::remove_change_nts (C++ function), 788
eprosima::fastrtps::rtps::History::reserve_Cache (C++ function), 787
eprosima::fastrtps::rtps::HistoryAttributes (C++ class), 726
eprosima::fastrtps::rtps::HistoryAttributes::extraReserved (C++ member), 727
eprosima::fastrtps::rtps::HistoryAttributes::HistoryAttrib (C++ function), 726, 727
eprosima::fastrtps::rtps::HistoryAttributes::initialReserv (C++ member), 727
eprosima::fastrtps::rtps::HistoryAttributes::maximumReserv (C++ member), 727
eprosima::fastrtps::rtps::HistoryAttributes::memoryPolicy (C++ member), 727
eprosima::fastrtps::rtps::HistoryAttributes::payloadMaxSiz (C++ member), 727
eprosima::fastrtps::rtps::IChangePool (C++ class), 790
eprosima::fastrtps::rtps::IChangePool::release_cache (C++ function), 790
eprosima::fastrtps::rtps::IChangePool::reserve_cache (C++ function), 790
eprosima::fastrtps::rtps::IdentityStatusToken (C++ type), 780
eprosima::fastrtps::rtps::IdentityToken (C++ type), 780
eprosima::fastrtps::rtps::iHandle2GUID (C++ function), 753
eprosima::fastrtps::rtps::InitialAnnouncementConfig (C++ struct), 727
eprosima::fastrtps::rtps::InitialAnnouncementConfig::count (C++ member), 728
eprosima::fastrtps::rtps::InitialAnnouncementConfig::perio (C++ member), 728
eprosima::fastrtps::rtps::InstanceHandle_t (C++ struct), 751
eprosima::fastrtps::rtps::InstanceHandle_t::isDefined (C++ function), 752
eprosima::fastrtps::rtps::InstanceHandle_t::operator= (C++ function), 752
eprosima::fastrtps::rtps::InstanceHandle_t::value (C++ member), 752

eprosima::fastrtps::rtps::IPayloadPool (*C++ class*), 791
eprosima::fastrtps::rtps::IPayloadPool::get_payload (*C++ function*), 791
eprosima::fastrtps::rtps::IPayloadPool::release_payload (*C++ function*), 792
eprosima::fastrtps::rtps::IsAddressDefined (*C++ function*), 754
eprosima::fastrtps::rtps::IsLocatorValid (*C++ function*), 754
eprosima::fastrtps::rtps::LivelinessData (*C++ struct*), 820
eprosima::fastrtps::rtps::LivelinessData::count (*C++ member*), 821
eprosima::fastrtps::rtps::LivelinessData::guid (*C++ member*), 821
eprosima::fastrtps::rtps::LivelinessData::kind (*C++ member*), 821
eprosima::fastrtps::rtps::LivelinessData::lease_duration (*C++ member*), 821
eprosima::fastrtps::rtps::LivelinessData::LivelinessData (*C++ function*), 820
eprosima::fastrtps::rtps::LivelinessData::operator!= (*C++ function*), 821
eprosima::fastrtps::rtps::LivelinessData::operator== (*C++ function*), 821
eprosima::fastrtps::rtps::LivelinessData::status (*C++ member*), 821
eprosima::fastrtps::rtps::LivelinessData::time (*C++ member*), 821
eprosima::fastrtps::rtps::Locator_t (*C++ class*), 754
eprosima::fastrtps::rtps::Locator_t::address (*C++ member*), 755
eprosima::fastrtps::rtps::Locator_t::get_address (*C++ function*), 755
eprosima::fastrtps::rtps::Locator_t::kind (*C++ member*), 755
eprosima::fastrtps::rtps::Locator_t::Locator_t (*C++ function*), 755
eprosima::fastrtps::rtps::Locator_t::operator= (*C++ function*), 755
eprosima::fastrtps::rtps::Locator_t::port (*C++ member*), 755
eprosima::fastrtps::rtps::Locator_t::set_address (*C++ function*), 755
eprosima::fastrtps::rtps::Locator_t::set_Invalid_Address (*C++ function*), 755
eprosima::fastrtps::rtps::LocatorSelector (*C++ class*), 762
eprosima::fastrtps::rtps::LocatorSelector::add_entry (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::clear (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::enable (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::for_each (*C++ function*), 764
eprosima::fastrtps::rtps::LocatorSelector::is_selected (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::iterator (*C++ class*), 764
eprosima::fastrtps::rtps::LocatorSelector::iterator::opera (*C++ function*), 764
eprosima::fastrtps::rtps::LocatorSelector::iterator::opera (*C++ function*), 764
eprosima::fastrtps::rtps::LocatorSelector::iterator::opera (*C++ function*), 764
eprosima::fastrtps::rtps::LocatorSelector::iterator::opera (*C++ function*), 764
eprosima::fastrtps::rtps::LocatorSelector::IteratorIndex (*C++ struct*), 764
eprosima::fastrtps::rtps::LocatorSelector::LocatorSelector (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::remove_entry (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::reset (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::select (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::selected_size (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::selection_start (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::state_has_chang (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelector::transport_start (*C++ function*), 763
eprosima::fastrtps::rtps::LocatorSelectorEntry (*C++ struct*), 761
eprosima::fastrtps::rtps::LocatorSelectorEntry::enable (*C++ function*), 761
eprosima::fastrtps::rtps::LocatorSelectorEntry::enabled (*C++ member*), 761
eprosima::fastrtps::rtps::LocatorSelectorEntry::EntryState (*C++ struct*), 762
eprosima::fastrtps::rtps::LocatorSelectorEntry::EntryState (*C++ function*), 762
eprosima::fastrtps::rtps::LocatorSelectorEntry::EntryState (*C++ member*), 762
eprosima::fastrtps::rtps::LocatorSelectorEntry::EntryState (*C++ member*), 762
eprosima::fastrtps::rtps::LocatorSelectorEntry::LocatorSel (*C++ function*), 761
eprosima::fastrtps::rtps::LocatorSelectorEntry::multicast (*C++ member*), 761
eprosima::fastrtps::rtps::LocatorSelectorEntry::remote_gui (*C++ member*), 761

eprosima::fastrtps::rtps::LocatorSelectorEntry::reset (C++ function), 761
eprosima::fastrtps::rtps::LocatorSelectorEntry::state (C++ member), 761
eprosima::fastrtps::rtps::LocatorSelectorEntry::transport_should_process (C++ member), 762
eprosima::fastrtps::rtps::LocatorSelectorEntry::unicast (C++ member), 761
eprosima::fastrtps::rtps::MatchingInfo (C++ class), 765
eprosima::fastrtps::rtps::MatchingInfo::MatchingInfo (C++ function), 765
eprosima::fastrtps::rtps::MatchingInfo::remoteEndpointGuid (C++ member), 765
eprosima::fastrtps::rtps::MatchingInfo::status (C++ member), 765
eprosima::fastrtps::rtps::MatchingStatus (C++ enum), 766
eprosima::fastrtps::rtps::MatchingStatus::MATCHED_MATCHING (C++ enumerator), 766
eprosima::fastrtps::rtps::MatchingStatus::REMOVED_MATCHING (C++ enumerator), 766
eprosima::fastrtps::rtps::MemoryManagementPolicy (C++ enum), 815
eprosima::fastrtps::rtps::MemoryManagementPolicy::DYNAMIC_RESERVE_MEMORY_MODE (C++ enumerator), 815
eprosima::fastrtps::rtps::MemoryManagementPolicy::DYNAMIC_REUSABLE_MEMORY_MODE (C++ enumerator), 815
eprosima::fastrtps::rtps::MemoryManagementPolicy::PREALLOCATED_MEMORY_MODE (C++ enumerator), 815
eprosima::fastrtps::rtps::MemoryManagementPolicy::PREALLOCATED_WITH_REALLOC_MEMORY_MODE (C++ enumerator), 815
eprosima::fastrtps::rtps::octet (C++ type), 782
eprosima::fastrtps::rtps::operator!= (C++ function), 746, 749, 752, 759, 769, 776
eprosima::fastrtps::rtps::operator+ (C++ function), 770, 777
eprosima::fastrtps::rtps::operator== (C++ function), 746, 749, 752, 759, 761, 769, 776, 797
eprosima::fastrtps::rtps::operator- (C++ function), 770, 771, 777
eprosima::fastrtps::rtps::operator> (C++ function), 770, 776
eprosima::fastrtps::rtps::operator>= (C++ function), 770, 776
eprosima::fastrtps::rtps::operator>> (C++ function), 747, 749, 751, 753, 760
eprosima::fastrtps::rtps::operator< (C++ function), 749, 752, 759, 770, 776
eprosima::fastrtps::rtps::operator<= (C++ function), 770, 776
eprosima::fastrtps::rtps::operator<< (C++ function), 747, 749, 751, 752, 759, 767, 771, 776, 782
eprosima::fastrtps::rtps::participant_stateless_message_re
(C++ member), 744
eprosima::fastrtps::rtps::participant_stateless_message_wr
(C++ member), 744
eprosima::fastrtps::rtps::participant_volatile_message_sec
(C++ member), 744
eprosima::fastrtps::rtps::participant_volatile_message_sec
(C++ member), 744
eprosima::fastrtps::rtps::ParticipantAuthenticationInfo (C++ struct), 797
eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::g
(C++ member), 797
eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::s
(C++ member), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo (C++ struct), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOV
(C++ enum), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOV
(C++ enumerator), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOV
(C++ enumerator), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOV
(C++ enumerator), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOV
(C++ enumerator), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOV
(C++ enumerator), 797
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::info
(C++ member), 798
eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::status
(C++ member), 798
eprosima::fastrtps::rtps::ParticipantFilteringFlags (C++ enum), 728
eprosima::fastrtps::rtps::ParticipantFilteringFlags::FILTE
(C++ enumerator), 728
eprosima::fastrtps::rtps::ParticipantFilteringFlags::FILTE
(C++ enumerator), 728
eprosima::fastrtps::rtps::ParticipantFilteringFlags::FILTE
(C++ enumerator), 728
eprosima::fastrtps::rtps::ParticipantFilteringFlags::NO_FI
(C++ enumerator), 728
eprosima::fastrtps::rtps::ParticipantProxyData (C++ class), 798
eprosima::fastrtps::rtps::ParticipantProxyData::clear (C++ function), 798
eprosima::fastrtps::rtps::ParticipantProxyData::copy (C++ function), 798
eprosima::fastrtps::rtps::ParticipantProxyData::default_lo
(C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::get_backup
(C++ function), 799
eprosima::fastrtps::rtps::ParticipantProxyData::get_persis

(C++ function), 799
eprosima::fastrtps::rtps::ParticipantProxyData::get_sample_identity (C++ function), 799
eprosima::fastrtps::rtps::ParticipantProxyData::get_serialized_size (C++ function), 798
eprosima::fastrtps::rtps::ParticipantProxyData::m_available_builtin_endpoints (C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::m_expects_inline_qos (C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::m_guid (C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::m_leaseDuration (C++ member), 800
eprosima::fastrtps::rtps::ParticipantProxyData::m_networkConfiguration (C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::m_participantName (C++ member), 800
eprosima::fastrtps::rtps::ParticipantProxyData::m_protocolVersion (C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::m_VendorId (C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::metatraffic_locators (C++ member), 799
eprosima::fastrtps::rtps::ParticipantProxyData::readFromCDRMessage (C++ function), 798
eprosima::fastrtps::rtps::ParticipantProxyData::set_announced_locators (C++ function), 799
eprosima::fastrtps::rtps::ParticipantProxyData::set_persistence_guid (C++ function), 799
eprosima::fastrtps::rtps::ParticipantProxyData::set_sample_identity (C++ function), 799
eprosima::fastrtps::rtps::ParticipantProxyData::updateData (C++ function), 798
eprosima::fastrtps::rtps::ParticipantProxyData::writeToCDRMessage (C++ function), 798
eprosima::fastrtps::rtps::PermissionsCredentialToken (C++ type), 780
eprosima::fastrtps::rtps::PermissionsToken (C++ type), 781
eprosima::fastrtps::rtps::PortParameters (C++ class), 766
eprosima::fastrtps::rtps::PortParameters::domainIDGain (C++ member), 766
eprosima::fastrtps::rtps::PortParameters::getMulticastPort (C++ function), 766
eprosima::fastrtps::rtps::PortParameters::getUnicastPort (C++ function), 766
eprosima::fastrtps::rtps::PortParameters::offsetd0 (C++ member), 766
eprosima::fastrtps::rtps::PortParameters::offsetd1 (C++ member), 766
eprosima::fastrtps::rtps::PortParameters::offsetd2 (C++ member), 767
eprosima::fastrtps::rtps::PortParameters::offsetd3 (C++ member), 767
eprosima::fastrtps::rtps::PortParameters::participantIDGain (C++ member), 766
eprosima::fastrtps::rtps::PortParameters::portBase (C++ member), 766
eprosima::fastrtps::rtps::PropertiesSerializer (C++ class), 767
eprosima::fastrtps::rtps::PropertyHelper (C++ class), 767
eprosima::fastrtps::rtps::PropertyPolicy (C++ class), 728
eprosima::fastrtps::rtps::PropertyPolicy::binary_properties (C++ function), 728
eprosima::fastrtps::rtps::PropertyPolicy::properties (C++ function), 728
eprosima::fastrtps::rtps::PropertyPolicyHelper (C++ class), 728
eprosima::fastrtps::rtps::PropertyPolicyHelper::find_property (C++ function), 729
eprosima::fastrtps::rtps::PropertyPolicyHelper::get_property (C++ function), 729
eprosima::fastrtps::rtps::PropertyPolicyHelper::get_property_value (C++ function), 729
eprosima::fastrtps::rtps::PropertyPolicyHelper::length (C++ function), 729
eprosima::fastrtps::rtps::PropertySeq (C++ type), 767
eprosima::fastrtps::rtps::ProtocolVersion_t (C++ struct), 782
eprosima::fastrtps::rtps::ReaderAttributes (C++ class), 729
eprosima::fastrtps::rtps::ReaderAttributes::data_sharing_listener (C++ member), 730
eprosima::fastrtps::rtps::ReaderAttributes::disable_positive_ACKs (C++ member), 729
eprosima::fastrtps::rtps::ReaderAttributes::endpoint (C++ member), 729
eprosima::fastrtps::rtps::ReaderAttributes::expectsInlineQos (C++ member), 729
eprosima::fastrtps::rtps::ReaderAttributes::liveliness_kind (C++ member), 729
eprosima::fastrtps::rtps::ReaderAttributes::liveliness_lease_duration (C++ member), 729
eprosima::fastrtps::rtps::ReaderAttributes::matched_writers_allocation (C++ member), 730
eprosima::fastrtps::rtps::ReaderAttributes::times (C++ member), 729
eprosima::fastrtps::rtps::ReaderDiscoveryInfo (C++ struct), 801
eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERY_STATUS (C++ enum), 801
eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERY_STATUS::CHANGED_QOS_READER (C++ enumerator), 801
eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERY_STATUS

(C++ enumerator), 801
eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERY_STATUS::CHANGED_QOS_READER (C++ enumerator), 801
eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERY_STATUS::REMOVED_READER (C++ enumerator), 801
eprosima::fastrtps::rtps::ReaderDiscoveryInfo::info (C++ member), 801
eprosima::fastrtps::rtps::ReaderDiscoveryInfo::status (C++ member), 801
eprosima::fastrtps::rtps::ReaderHistory (C++ class), 792
eprosima::fastrtps::rtps::ReaderHistory::add_change (C++ function), 794
eprosima::fastrtps::rtps::ReaderHistory::can_change_be_added_nts (C++ function), 793
eprosima::fastrtps::rtps::ReaderHistory::completed_change (C++ function), 793, 794
eprosima::fastrtps::rtps::ReaderHistory::matches_change (C++ function), 794
eprosima::fastrtps::rtps::ReaderHistory::ReaderHistory (C++ function), 793
eprosima::fastrtps::rtps::ReaderHistory::received_change (C++ function), 793
eprosima::fastrtps::rtps::ReaderHistory::remove_change (C++ function), 795
eprosima::fastrtps::rtps::ReaderHistory::remove_change_sub (C++ function), 794
eprosima::fastrtps::rtps::ReaderHistory::remove_changes_with_guid (C++ function), 794
eprosima::fastrtps::rtps::ReaderHistory::remove_fastest_changes_upto (C++ function), 795
eprosima::fastrtps::rtps::ReaderHistory::writer_unmatched (C++ function), 795
eprosima::fastrtps::rtps::ReaderHistory::~ReaderHistory (C++ function), 795
eprosima::fastrtps::rtps::ReaderListener (C++ class), 809
eprosima::fastrtps::rtps::ReaderListener::on_data_available (C++ function), 811
eprosima::fastrtps::rtps::ReaderListener::on_incompatible_type (C++ function), 811
eprosima::fastrtps::rtps::ReaderListener::on_liveliness_changed (C++ function), 810
eprosima::fastrtps::rtps::ReaderListener::on_requested_incompatible_qos (C++ function), 810
eprosima::fastrtps::rtps::ReaderListener::on_sample_lost (C++ function), 810
eprosima::fastrtps::rtps::ReaderListener::on_sample_rejected (C++ function), 810
eprosima::fastrtps::rtps::ReaderListener::on_writer_discovery (C++ function), 810
eprosima::fastrtps::rtps::ReaderListener::onNewCacheChangeAdded (C++ function), 810
eprosima::fastrtps::rtps::ReaderListener::onReaderMatched (C++ function), 809
eprosima::fastrtps::rtps::ReaderProxyData (C++ class), 801
eprosima::fastrtps::rtps::ReaderProxyData::clear (C++ function), 802
eprosima::fastrtps::rtps::ReaderProxyData::copy (C++ function), 802
eprosima::fastrtps::rtps::ReaderProxyData::get_sample_iden (C++ function), 801
eprosima::fastrtps::rtps::ReaderProxyData::get_serialized_ (C++ function), 801
eprosima::fastrtps::rtps::ReaderProxyData::is_update_allow (C++ function), 802
eprosima::fastrtps::rtps::ReaderProxyData::m_qos (C++ member), 802
eprosima::fastrtps::rtps::ReaderProxyData::plugin_security (C++ member), 802
eprosima::fastrtps::rtps::ReaderProxyData::readFromCDRMess (C++ function), 802
eprosima::fastrtps::rtps::ReaderProxyData::security_attrib (C++ member), 802
eprosima::fastrtps::rtps::ReaderProxyData::set_sample_iden (C++ function), 801
eprosima::fastrtps::rtps::ReaderProxyData::update (C++ function), 802
eprosima::fastrtps::rtps::ReaderProxyData::writeToCDRMessa (C++ function), 802
eprosima::fastrtps::rtps::ReaderTimes (C++ class), 730
eprosima::fastrtps::rtps::ReaderTimes::heartbeatResponseDe (C++ member), 730
eprosima::fastrtps::rtps::ReaderTimes::initialAcknackDelay (C++ member), 730
eprosima::fastrtps::rtps::ReliabilityKind_t (C++ type), 783
eprosima::fastrtps::rtps::RemoteLocatorList (C++ struct), 767
eprosima::fastrtps::rtps::RemoteLocatorList::add_multicast (C++ function), 768
eprosima::fastrtps::rtps::RemoteLocatorList::add_unicast_l (C++ function), 768
eprosima::fastrtps::rtps::RemoteLocatorList::multicast (C++ member), 768
eprosima::fastrtps::rtps::RemoteLocatorList::operator= (C++ function), 768
eprosima::fastrtps::rtps::RemoteLocatorList::RemoteLocator (C++ function), 768
eprosima::fastrtps::rtps::RemoteLocatorList::unicast (C++ member), 768
eprosima::fastrtps::rtps::RemoteLocatorsAllocationAttribut (C++ struct), 730
eprosima::fastrtps::rtps::RemoteLocatorsAllocationAttribut (C++ member), 730
eprosima::fastrtps::rtps::RemoteLocatorsAllocationAttribut

(C++ *member*), 730
eprosima::fastrtps::rtps::RTPSDomain (C++ *class*), 816
eprosima::fastrtps::rtps::RTPSDomain::createParticipant (C++ *function*), 816
eprosima::fastrtps::rtps::RTPSDomain::createRTPSReader (C++ *function*), 819
eprosima::fastrtps::rtps::RTPSDomain::createRTPSWriter (C++ *function*), 817, 818
eprosima::fastrtps::rtps::RTPSDomain::removeRTPSParticipant (C++ *function*), 820
eprosima::fastrtps::rtps::RTPSDomain::removeRTPSReader (C++ *function*), 820
eprosima::fastrtps::rtps::RTPSDomain::removeRTPSWriter (C++ *function*), 819
eprosima::fastrtps::rtps::RTPSDomain::set_filename (C++ *function*), 816
eprosima::fastrtps::rtps::RTPSDomain::stopAll (C++ *function*), 816
eprosima::fastrtps::rtps::RTPSParticipant (C++ *class*), 804
eprosima::fastrtps::rtps::RTPSParticipant::announceRTPSParticipantState (C++ *function*), 804
eprosima::fastrtps::rtps::RTPSParticipant::enable (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::getDomainId (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::getEntityID (C++ *function*), 807
eprosima::fastrtps::rtps::RTPSParticipant::getGuid (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::getMaxDataSize (C++ *function*), 804
eprosima::fastrtps::rtps::RTPSParticipant::getMaxMessageSize (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::getParticipantNames (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::getRTPSParticipantAttributes (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::getUserDefinedID (C++ *function*), 805
eprosima::fastrtps::rtps::RTPSParticipant::ignore_participant (C++ *function*), 807
eprosima::fastrtps::rtps::RTPSParticipant::ignore_reader (C++ *function*), 807
eprosima::fastrtps::rtps::RTPSParticipant::ignore_writer (C++ *function*), 807
eprosima::fastrtps::rtps::RTPSParticipant::is_security_enabled_for_reader (C++ *function*), 807
eprosima::fastrtps::rtps::RTPSParticipant::is_security_enabled_for_writer (C++ *function*), 807
eprosima::fastrtps::rtps::RTPSParticipant::newRemoteReaderDiscovered (C++ *function*), 805
eprosima::fastrtps::rtps::RTPSParticipant::newRemoteWriter (C++ *function*), 804
eprosima::fastrtps::rtps::RTPSParticipant::registerReader (C++ *function*), 805
eprosima::fastrtps::rtps::RTPSParticipant::registerWriter (C++ *function*), 805
eprosima::fastrtps::rtps::RTPSParticipant::resetRTPSParticipant (C++ *function*), 804
eprosima::fastrtps::rtps::RTPSParticipant::set_check_type_function (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::set_listener (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::stopRTPSParticipant (C++ *function*), 804
eprosima::fastrtps::rtps::RTPSParticipant::typelookup_manager_and_protocol (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::update_attributes (C++ *function*), 805
eprosima::fastrtps::rtps::RTPSParticipant::updateReader (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipant::updateWriter (C++ *function*), 805
eprosima::fastrtps::rtps::RTPSParticipant::wlp (C++ *function*), 806
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes (C++ *struct*), 731
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::data_limits (C++ *member*), 732
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::data_max_serialized (C++ *member*), 732
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::guid_prefix (C++ *member*), 731
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::max_unicast_locators (C++ *member*), 731
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::max_user_data (C++ *member*), 731
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::participant_id (C++ *member*), 731
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::readers (C++ *member*), 732
eprosima::fastrtps::rtps::RTPSParticipantAllocationAttributes::writers (C++ *member*), 731
eprosima::fastrtps::rtps::RTPSParticipantAttributes (C++ *class*), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::allocation (C++ *member*), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::builtin (C++ *member*), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::builtin_controllers_sender_thread_attributes (C++ *member*), 733

(C++ member), 734
eprosima::fastrtps::rtps::RTPSParticipantAttributes::defaultExternalUnicastLocators (C++ member), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::defaultMulticastLocatorList (C++ member), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::defaultUnicastLocatorList (C++ member), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::discovery_server_thread (C++ member), 734
eprosima::fastrtps::rtps::RTPSParticipantAttributes::flow_controllers (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::getName (C++ function), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::ignored_entities (C++ member), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::listenSocketBufferSize (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::max_msg_size_no_frag (C++ member), 734
eprosima::fastrtps::rtps::RTPSParticipantAttributes::netmaskFilter (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::participantID (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::port (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::prefix (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::properties (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::use_client_ack_nack (C++ member), 734
eprosima::fastrtps::rtps::RTPSParticipantAttributes::sendSocketBufferSize (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::setName (C++ function), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::setup_transports (C++ function), 732
eprosima::fastrtps::rtps::RTPSParticipantAttributes::throughput_controller (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::timeSeparation (C++ member), 734
eprosima::fastrtps::rtps::RTPSParticipantAttributes::useBuiltinTransports (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::userData (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantAttributes::userTransports (C++ member), 733
eprosima::fastrtps::rtps::RTPSParticipantListener (C++ class), 807
eprosima::fastrtps::rtps::RTPSParticipantListener::onParticipantChangedQos (C++ function), 809
eprosima::fastrtps::rtps::RTPSParticipantListener::onParticipantDiscovery (C++ function), 808
eprosima::fastrtps::rtps::RTPSParticipantListener::onRTPSParticipantImportantInfoChanged (C++ function), 809
eprosima::fastrtps::rtps::RTPSParticipantListener::onRTPSParticipantTypeIntrospectionQueueFull (C++ function), 807, 808
eprosima::fastrtps::rtps::RTPSParticipantListener::onReaderDiscovery (C++ function), 808
eprosima::fastrtps::rtps::RTPSParticipantListener::onWriterDiscovery (C++ function), 808
eprosima::fastrtps::rtps::RTPSReader (C++ class), 811
eprosima::fastrtps::rtps::RTPSReader::assert_writer_liveliness (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::begin_sample_access_nts (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::change_read_by_user (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::change_removed_by_history (C++ function), 813
eprosima::fastrtps::rtps::RTPSReader::end_sample_access_nts (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::expectsInlineQos (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::get_content_filter (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::getHistory (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::getListener (C++ function), 813
eprosima::fastrtps::rtps::RTPSReader::is_sample_valid (C++ function), 815
eprosima::fastrtps::rtps::RTPSReader::isInCleanState (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::liveliness_changed_status (C++ member), 815
eprosima::fastrtps::rtps::RTPSReader::matched_writer_add (C++ function), 811
eprosima::fastrtps::rtps::RTPSReader::matched_writer_is_matched (C++ function), 812
eprosima::fastrtps::rtps::RTPSReader::matched_writer_remove (C++ function), 811
eprosima::fastrtps::rtps::RTPSReader::nextUnreadCache (C++ function), 813
eprosima::fastrtps::rtps::RTPSReader::nextUntakenCache (C++ function), 813
eprosima::fastrtps::rtps::RTPSReader::processDataFragMsg (C++ function), 812
eprosima::fastrtps::rtps::RTPSReader::processDataMsg (C++ function), 812
eprosima::fastrtps::rtps::RTPSReader::processGapMsg (C++ function), 812
eprosima::fastrtps::rtps::RTPSReader::processHeartbeatMsg (C++ function), 812
eprosima::fastrtps::rtps::RTPSReader::releaseCache (C++ function), 813
eprosima::fastrtps::rtps::RTPSReader::reserveCache

(C++ function), 813
eprosima::fastrtps::rtps::RTPSReader::set_content_filter (C++ function), 814
eprosima::fastrtps::rtps::RTPSReader::setListener (C++ function), 813
eprosima::fastrtps::rtps::RTPSWriter (C++ class), 821
eprosima::fastrtps::rtps::RTPSWriter::calculateMaxDataSize (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::deliver_sample_nts (C++ function), 825
eprosima::fastrtps::rtps::RTPSWriter::get_disable_positive_acks (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::get_liveliness_announcement_period (C++ function), 825
eprosima::fastrtps::rtps::RTPSWriter::get_liveliness_kind (C++ function), 825
eprosima::fastrtps::rtps::RTPSWriter::get_liveliness_lease_duration (C++ function), 825
eprosima::fastrtps::rtps::RTPSWriter::get_separate_sending (C++ function), 824
eprosima::fastrtps::rtps::RTPSWriter::get_seq_num_max (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::get_seq_num_min (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::getListener (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::getMaxDataSize (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::getRTPSParticipant (C++ function), 824
eprosima::fastrtps::rtps::RTPSWriter::getTypeMaxSerialized (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::has_been_fully_delivered (C++ function), 822
eprosima::fastrtps::rtps::RTPSWriter::is_acked_by_all (C++ function), 822
eprosima::fastrtps::rtps::RTPSWriter::is_data_sharing_compatible (C++ function), 825
eprosima::fastrtps::rtps::RTPSWriter::isAsync (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::liveliness_lost_status (C++ member), 826
eprosima::fastrtps::rtps::RTPSWriter::matched_reader_add (C++ function), 822
eprosima::fastrtps::rtps::RTPSWriter::matched_reader_is_matched (C++ function), 822
eprosima::fastrtps::rtps::RTPSWriter::matched_reader_remove (C++ function), 822
eprosima::fastrtps::rtps::RTPSWriter::new_change (C++ function), 821
eprosima::fastrtps::rtps::RTPSWriter::process_acknack (C++ function), 824
eprosima::fastrtps::rtps::RTPSWriter::process_nack_frag (C++ function), 824
eprosima::fastrtps::rtps::RTPSWriter::reader_data_filter (C++ function), 822
eprosima::fastrtps::rtps::RTPSWriter::release_change (C++ function), 822
eprosima::fastrtps::rtps::RTPSWriter::remove_older_changes (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::send_nts (C++ function), 825
eprosima::fastrtps::rtps::RTPSWriter::set_separate_sending (C++ function), 824
eprosima::fastrtps::rtps::RTPSWriter::try_remove_change (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::updateAttributes (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriter::wait_for_acknowledge (C++ function), 824
eprosima::fastrtps::rtps::RTPSWriter::wait_for_all_acked (C++ function), 823
eprosima::fastrtps::rtps::RTPSWriterPublishMode (C++ enum), 734
eprosima::fastrtps::rtps::RTPSWriterPublishMode::ASYNCHRONOUS_WRITER (C++ enumerator), 734
eprosima::fastrtps::rtps::RTPSWriterPublishMode::SYNCHRONOUS_WRITER (C++ enumerator), 734
eprosima::fastrtps::rtps::SampleIdentity (C++ class), 769
eprosima::fastrtps::rtps::SampleIdentity::operator= (C++ function), 769
eprosima::fastrtps::rtps::SampleIdentity::operator< (C++ function), 769
eprosima::fastrtps::rtps::SampleIdentity::SampleIdentity (C++ function), 769
eprosima::fastrtps::rtps::sedp_builtin_publications_secure_reader (C++ member), 744
eprosima::fastrtps::rtps::sedp_builtin_publications_secure_writer (C++ member), 744
eprosima::fastrtps::rtps::sedp_builtin_subscriptions_secure_reader (C++ member), 744
eprosima::fastrtps::rtps::sedp_builtin_subscriptions_secure_writer (C++ member), 744
eprosima::fastrtps::rtps::SendBuffersAllocationAttributes (C++ struct), 734
eprosima::fastrtps::rtps::SendBuffersAllocationAttributes::initial_buffers_count (C++ member), 734
eprosima::fastrtps::rtps::SendBuffersAllocationAttributes::max_buffers_count (C++ member), 734
eprosima::fastrtps::rtps::SequenceNumber_t (C++ struct), 771
eprosima::fastrtps::rtps::SequenceNumber_t::operator++ (C++ function), 772
eprosima::fastrtps::rtps::SequenceNumber_t::operator+= (C++ function), 772
eprosima::fastrtps::rtps::SequenceNumber_t::SequenceNumber

(C++ function), 772
eprosima::fastrtps::rtps::SequenceNumber_t::to4long (C++ function), 772
eprosima::fastrtps::rtps::SequenceNumberDiff (C++ struct), 772
eprosima::fastrtps::rtps::SequenceNumberHash (C++ struct), 772
eprosima::fastrtps::rtps::SequenceNumberSet_t (C++ type), 772
eprosima::fastrtps::rtps::SerializedPayload_t (C++ struct), 773
eprosima::fastrtps::rtps::SerializedPayload_t::copy (C++ function), 773
eprosima::fastrtps::rtps::SerializedPayload_t::data (C++ member), 774
eprosima::fastrtps::rtps::SerializedPayload_t::empty (C++ function), 773
eprosima::fastrtps::rtps::SerializedPayload_t::encapsulation (C++ member), 774
eprosima::fastrtps::rtps::SerializedPayload_t::length (C++ member), 774
eprosima::fastrtps::rtps::SerializedPayload_t::max_size (C++ member), 774
eprosima::fastrtps::rtps::SerializedPayload_t::pos (C++ member), 774
eprosima::fastrtps::rtps::SerializedPayload_t::representation_header_size (C++ member), 774
eprosima::fastrtps::rtps::SerializedPayload_t::reserve (C++ function), 773
eprosima::fastrtps::rtps::SerializedPayload_t::SerializedPayload_t (C++ function), 773
eprosima::fastrtps::rtps::SimpleEDPAttributes (C++ class), 735
eprosima::fastrtps::rtps::SimpleEDPAttributes::use_PublicationReaderANDSubscriptionWriter (C++ member), 735
eprosima::fastrtps::rtps::SimpleEDPAttributes::use_PublicationWriterANDSubscriptionReader (C++ member), 735
eprosima::fastrtps::rtps::sort_seqNum (C++ function), 773
eprosima::fastrtps::rtps::SubmessageFlag (C++ type), 783
eprosima::fastrtps::rtps::Time_t (C++ class), 778
eprosima::fastrtps::rtps::Time_t::fraction (C++ function), 779
eprosima::fastrtps::rtps::Time_t::from_ns (C++ function), 779
eprosima::fastrtps::rtps::Time_t::nanosec (C++ function), 779
eprosima::fastrtps::rtps::Time_t::now (C++ function), 780
eprosima::fastrtps::rtps::Time_t::seconds (C++ function), 779
eprosima::fastrtps::rtps::Time_t::Time_t (C++ function), 779
eprosima::fastrtps::rtps::Time_t::to_ns (C++ function), 779
eprosima::fastrtps::rtps::Token (C++ type), 781
eprosima::fastrtps::rtps::TopicKind_t (C++ type), 783
eprosima::fastrtps::rtps::TypeLookupSettings (C++ class), 735
eprosima::fastrtps::rtps::TypeLookupSettings::use_client (C++ member), 735
eprosima::fastrtps::rtps::TypeLookupSettings::use_server (C++ member), 735
eprosima::fastrtps::rtps::VariableLengthDataLimits (C++ struct), 735
eprosima::fastrtps::rtps::VariableLengthDataLimits::max_da
eprosima::fastrtps::rtps::VariableLengthDataLimits::max_pa
eprosima::fastrtps::rtps::VariableLengthDataLimits::max_pr
eprosima::fastrtps::rtps::VariableLengthDataLimits::max_us
eprosima::fastrtps::rtps::WriteParams (C++ class), 783
eprosima::fastrtps::rtps::WriteParams::related_sample_iden
eprosima::fastrtps::rtps::WriteParams::sample_identity
eprosima::fastrtps::rtps::WriteParams::source_timestamp
eprosima::fastrtps::rtps::WriteParams::write_params_defaul (C++ function), 784
eprosima::fastrtps::rtps::WriterAttributes (C++ class), 736
eprosima::fastrtps::rtps::WriterAttributes::disable_heartb
eprosima::fastrtps::rtps::WriterAttributes::disable_positi (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::endpoint (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::flow_controlle (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::keep_duration (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::liveliness_ann (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::liveliness_kin (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::liveliness_lea (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::matched_reader (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::mode (C++ member), 736

eprosima::fastrtps::rtps::WriterAttributes::throughputController (C++ member), 736
eprosima::fastrtps::rtps::WriterAttributes::times (C++ member), 736
eprosima::fastrtps::rtps::WriterDiscoveryInfo (C++ struct), 803
eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERY_STATUS (C++ enum), 803
eprosima::fastrtps::rtps::WriterDiscoveryInfo::CHANGED_QOS_WRITER (C++ enumerator), 803
eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERED_WRITER (C++ enumerator), 803
eprosima::fastrtps::rtps::WriterDiscoveryInfo::IGNORED_WRITER (C++ enumerator), 803
eprosima::fastrtps::rtps::WriterDiscoveryInfo::REMOVED_WRITER (C++ enumerator), 803
eprosima::fastrtps::rtps::WriterDiscoveryInfo::info (C++ member), 803
eprosima::fastrtps::rtps::WriterDiscoveryInfo::status (C++ member), 803
eprosima::fastrtps::rtps::WriterHistory (C++ class), 795
eprosima::fastrtps::rtps::WriterHistory::add_change (C++ function), 796
eprosima::fastrtps::rtps::WriterHistory::matched_change (C++ function), 796
eprosima::fastrtps::rtps::WriterHistory::remove_change (C++ function), 797
eprosima::fastrtps::rtps::WriterHistory::remove_change_g (C++ function), 796
eprosima::fastrtps::rtps::WriterHistory::remove_min_change (C++ function), 796
eprosima::fastrtps::rtps::WriterHistory::WriterHistory (C++ function), 796
eprosima::fastrtps::rtps::WriterListener (C++ class), 826
eprosima::fastrtps::rtps::WriterListener::on_incompatible_type (C++ function), 827
eprosima::fastrtps::rtps::WriterListener::on_liveliness_lost (C++ function), 827
eprosima::fastrtps::rtps::WriterListener::on_offered_incompatible_qos (C++ function), 826
eprosima::fastrtps::rtps::WriterListener::on_reader_discovery (C++ function), 827
eprosima::fastrtps::rtps::WriterListener::onWriterChangeReceivedByAll (C++ function), 827
eprosima::fastrtps::rtps::WriterListener::onWriterMatched (C++ function), 826
eprosima::fastrtps::rtps::WriterProxyData (C++ class), 803
eprosima::fastrtps::rtps::WriterProxyData::clear (C++ function), 803
eprosima::fastrtps::rtps::WriterProxyData::copy (C++ function), 804
eprosima::fastrtps::rtps::WriterProxyData::get_sample_iden (C++ function), 803
eprosima::fastrtps::rtps::WriterProxyData::get_serialized_ (C++ function), 804
eprosima::fastrtps::rtps::WriterProxyData::is_update_allow (C++ function), 803
eprosima::fastrtps::rtps::WriterProxyData::m_qos (C++ member), 804
eprosima::fastrtps::rtps::WriterProxyData::plugin_security (C++ member), 804
eprosima::fastrtps::rtps::WriterProxyData::readFromCDRMess (C++ function), 804
eprosima::fastrtps::rtps::WriterProxyData::security_attrib (C++ member), 804
eprosima::fastrtps::rtps::WriterProxyData::set_sample_iden (C++ function), 803
eprosima::fastrtps::rtps::WriterProxyData::update (C++ function), 803
eprosima::fastrtps::rtps::WriterProxyData::writeToCDRMessa (C++ function), 804
eprosima::fastrtps::rtps::WriterTimes (C++ struct), 737
eprosima::fastrtps::rtps::WriterTimes::heartbeatPeriod (C++ member), 737
eprosima::fastrtps::rtps::WriterTimes::initialHeartbeatDel (C++ member), 737
eprosima::fastrtps::rtps::WriterTimes::nackResponseDelay (C++ member), 737
eprosima::fastrtps::rtps::WriterTimes::nackSupressionDurat (C++ member), 737
eprosima::fastrtps::Time_t (C++ struct), 775
eprosima::fastrtps::Time_t::now (C++ function), 775
eprosima::fastrtps::Time_t::Time_t (C++ function), 775
eprosima::fastrtps::Time_t::to_ns (C++ function), 775
EPROSIMA_LOG_ERROR (C macro), 855
EPROSIMA_LOG_INFO (C macro), 855
EPROSIMA_LOG_WARNING (C macro), 855
EXCLUSIVE_COMPATIBLE_QOS (class in fastdds), 873
expects_inline_qos() (fastdds.DataReaderQos method), 966
external_unicast_locators (fastdds.RTPSEndpointQos property), 881
extra_samples (fastdds.ResourceLimitsQosPolicy property), 880

F

FASTDDS_SEQUENCE (C macro), 625
FASTDDS_SQLFILTER_NAME (C macro), 713
FASTDDS_STATUS_COUNT (C macro), 619
file_watch_threads() (fastdds.DomainParticipantFactoryQos method),

912
find_topic() (*fastdds.DomainParticipant method*), 897
find_type() (*fastdds.DomainParticipant method*), 897
flow_controller_name (*fastdds.PublishModeQosPolicy property*), 877
flow_controllers() (*fastdds.DomainParticipantQos method*), 917

G

generation_rank (*fastdds.SampleInfo property*), 974
GenericDataQosPolicy (*class in fastdds*), 870
get() (*fastdds.TypeIdV1 method*), 996
get() (*fastdds.TypeObjectV1 method*), 997
get_builtin_subscriber() (*fastdds.DomainParticipant method*), 898
get_current_time() (*fastdds.DomainParticipant method*), 898
get_datareader_qos_from_profile() (*fastdds.Subscriber method*), 977
get_datareaders() (*fastdds.Subscriber method*), 977
get_datawriter_qos_from_profile() (*fastdds.Publisher method*), 940
get_datawriters() (*fastdds.Publisher method*), 940
get_default_datareader_qos() (*fastdds.Subscriber method*), 978
get_default_datawriter_qos() (*fastdds.Publisher method*), 940
get_default_participant_qos() (*fastdds.DomainParticipantFactory method*), 909
get_default_publisher_qos() (*fastdds.DomainParticipant method*), 898
get_default_subscriber_qos() (*fastdds.DomainParticipant method*), 898
get_default_topic_qos() (*fastdds.DomainParticipant method*), 899
get_discovered_participant_data() (*fastdds.DomainParticipant method*), 899
get_discovered_participants() (*fastdds.DomainParticipant method*), 900
get_discovered_topic_data() (*fastdds.DomainParticipant method*), 900
get_discovered_topics() (*fastdds.DomainParticipant method*), 900
get_domain_id() (*fastdds.DomainParticipant method*), 900
get_first_untaken_info() (*fastdds.DataReader method*), 947
get_impl() (*fastdds.Topic method*), 984
get_impl() (*fastdds.TopicDescription method*), 989
get_instance() (*fastdds.DomainParticipantFactory static method*), 910
get_instance_handle() (*fastdds.DataReader method*), 948
get_instance_handle() (*fastdds.DataWriter method*), 921
get_instance_handle() (*fastdds.DomainParticipant method*), 900
get_instance_handle() (*fastdds.Entity method*), 862
get_instance_handle() (*fastdds.Publisher method*), 941
get_instance_handle() (*fastdds.Subscriber method*), 978
get_key() (*fastdds.TypeSupport method*), 998
get_key_value() (*fastdds.DataReader method*), 948
get_key_value() (*fastdds.DataWriter method*), 921
get_listener() (*fastdds.DataReader method*), 948
get_listener() (*fastdds.DataWriter method*), 921
get_listener() (*fastdds.DomainParticipant method*), 900
get_listener() (*fastdds.Publisher method*), 941
get_listener() (*fastdds.Subscriber method*), 978
get_listener() (*fastdds.Topic method*), 984
get_listening_locators() (*fastdds.DataReader method*), 948
get_liveliness_changed_status() (*fastdds.DataReader method*), 948
get_liveliness_lost_status() (*fastdds.DataWriter method*), 922
get_matched_publication_data() (*fastdds.DataReader method*), 948
get_matched_publications() (*fastdds.DataReader method*), 948
get_matched_subscription_data() (*fastdds.DataWriter method*), 922
get_matched_subscriptions() (*fastdds.DataWriter method*), 922
get_name() (*fastdds.TopicDescription method*), 989
get_offered_deadline_missed_status() (*fastdds.DataWriter method*), 922
get_offered_incompatible_qos_status() (*fastdds.DataWriter method*), 922
get_participant() (*fastdds.Publisher method*), 941
get_participant() (*fastdds.Subscriber method*), 979
get_participant() (*fastdds.Topic method*), 984
get_participant() (*fastdds.TopicDescription method*), 989
get_participant_names() (*fastdds.DomainParticipant method*), 900
get_participant_qos_from_profile() (*fastdds.DomainParticipantFactory method*), 910
get_publication_matched_status() (*fastdds.DataWriter method*), 922
get_publisher() (*fastdds.DataWriter method*), 922
get_publisher_qos_from_profile() (*fastdds.DomainParticipant method*), 900
get_qos() (*fastdds.DataReader method*), 949
get_qos() (*fastdds.DataWriter method*), 922

get_qos() (*fastdds.DomainParticipant method*), 901
get_qos() (*fastdds.DomainParticipantFactory method*), 910
get_qos() (*fastdds.Publisher method*), 941
get_qos() (*fastdds.Subscriber method*), 979
get_qos() (*fastdds.Topic method*), 984
get_readerqos() (*fastdds.DataReaderQos method*), 966
get_requested_deadline_missed_status() (*fastdds.DataReader method*), 949
get_requested_incompatible_qos_status() (*fastdds.DataReader method*), 949
get_resource_event() (*fastdds.DomainParticipant method*), 901
get_sample_lost_status() (*fastdds.DataReader method*), 949
get_sample_rejected_status() (*fastdds.DataReader method*), 949
get_sending_locators() (*fastdds.DataWriter method*), 923
get_serialized_size_provider() (*fastdds.TypeSupport method*), 998
get_shared_instance() (*fastdds.DomainParticipantFactory static method*), 910
get_status_changes() (*fastdds.Entity method*), 862
get_status_mask() (*fastdds.Entity method*), 862
get_statuscondition() (*fastdds.Entity method*), 862
get_subscriber() (*fastdds.DataReader method*), 949
get_subscriber_qos_from_profile() (*fastdds.DomainParticipant method*), 901
get_subscription_matched_status() (*fastdds.DataReader method*), 949
get_topic() (*fastdds.DataWriter method*), 923
get_topic_qos_from_profile() (*fastdds.DomainParticipant method*), 901
get_topicdescription() (*fastdds.DataReader method*), 950
get_type() (*fastdds.DataWriter method*), 923
get_type_dependencies() (*fastdds.DomainParticipant method*), 901
get_type_name() (*fastdds.TopicDescription method*), 989
get_type_name() (*fastdds.TypeSupport method*), 999
get_types() (*fastdds.DomainParticipant method*), 902
get_unread_count() (*fastdds.DataReader method*), 950
get_writerqos() (*fastdds.DataWriterQos method*), 931
getKey() (*fastdds.TopicDataType method*), 986
getName() (*fastdds.TopicDataType method*), 986
getNames() (*fastdds.PartitionQosPolicy method*), 875
getSerializedSizeProvider() (*fastdds.TopicDataType method*), 986
getValue() (*fastdds.GenericDataQosPolicy method*), 870
group_data() (*fastdds.PublisherQos method*), 944
group_data() (*fastdds.SubscriberQos method*), 981
GROUP_PRESENTATION_QOS (*class in fastdds*), 876
GROUPDATA_QOS_POLICY_ID (*class in fastdds*), 878
GroupDataQosPolicy (*class in fastdds*), 871
guid() (*fastdds.DataReader method*), 950
guid() (*fastdds.DataWriter method*), 923
guid() (*fastdds.DomainParticipant method*), 902

H

has_datareaders() (*fastdds.Subscriber method*), 979
has_datawriters() (*fastdds.Publisher method*), 942
has_ownership() (*fastdds.LoanableCollection method*), 891
hasChanged (*fastdds.QosPolicy property*), 877
history() (*fastdds.DataReaderQos method*), 966
history() (*fastdds.DataWriterQos method*), 931
history() (*fastdds.TopicQos method*), 992
history_depth (*fastdds.DurabilityServiceQosPolicy property*), 869
history_kind (*fastdds.DurabilityServiceQosPolicy property*), 869
history_memory_policy (*fastdds.RTPSEndpointQos property*), 881
HISTORY_QOS_POLICY_ID (*class in fastdds*), 878
HistoryQosPolicy (*class in fastdds*), 871

I

ignore_non_matching_locators (*fastdds.RTPSEndpointQos property*), 881
ignore_non_matching_locators (*fastdds.WireProtocolConfigQos property*), 884
ignore_participant() (*fastdds.DomainParticipant method*), 902
ignore_publication() (*fastdds.DomainParticipant method*), 902
ignore_subscription() (*fastdds.DomainParticipant method*), 902
ignore_topic() (*fastdds.DomainParticipant method*), 902
IncompatibleQosStatus (*class in fastdds*), 886
inconsistent_topic() (*fastdds.StatusMask static method*), 889
instance_handle (*fastdds.SampleInfo property*), 974
INSTANCE_PRESENTATION_QOS (*class in fastdds*), 876
instance_state (*fastdds.SampleInfo property*), 975
INVALID_QOS_POLICY_ID (*class in fastdds*), 878
is_active() (*fastdds.StatusMask method*), 889
is_bounded() (*fastdds.TopicDataType method*), 986
is_bounded() (*fastdds.TypeSupport method*), 999
is_enabled() (*fastdds.Entity method*), 862
is_plain() (*fastdds.TopicDataType method*), 986

is_plain() (*fastdds.TypeSupport method*), 999
is_sample_valid() (*fastdds.DataReader method*), 950

K

KEEP_ALL_HISTORY_QOS (*class in fastdds*), 871
KEEP_LAST_HISTORY_QOS (*class in fastdds*), 871
kind (*fastdds.DestinationOrderQosPolicy property*), 867
kind (*fastdds.DurabilityQosPolicy property*), 868
kind (*fastdds.HistoryQosPolicy property*), 871
kind (*fastdds.LivelinessQosPolicy property*), 873
kind (*fastdds.OwnershipQosPolicy property*), 873
kind (*fastdds.PublishModeQosPolicy property*), 877
kind (*fastdds.ReliabilityQosPolicy property*), 879
kind() (*fastdds.DataSharingQosPolicy method*), 865

L

last_instance_handle (*fastdds.DeadlineMissedStatus property*), 886
last_instance_handle (*fastdds.SampleRejectedStatus property*), 888
last_policy_id (*fastdds.IncompatibleQosStatus property*), 886
last_publication_handle (*fastdds.LivelinessChangedStatus property*), 887
last_publication_handle (*fastdds.SubscriptionMatchedStatus property*), 891
last_reason (*fastdds.SampleRejectedStatus property*), 888
last_subscription_handle (*fastdds.PublicationMatchedStatus property*), 887
latency_budget() (*fastdds.DataReaderQos method*), 967
latency_budget() (*fastdds.DataWriterQos method*), 932
latency_budget() (*fastdds.TopicQos method*), 992
LATENCYBUDGET_QOS_POLICY_ID (*class in fastdds*), 878
LatencyBudgetQosPolicy (*class in fastdds*), 872
lease_duration (*fastdds.LivelinessQosPolicy property*), 873
length() (*fastdds.LoanableCollection method*), 891
lifespan() (*fastdds.DataReaderQos method*), 967
lifespan() (*fastdds.DataWriterQos method*), 932
lifespan() (*fastdds.TopicQos method*), 993
LIFESPAN_QOS_POLICY_ID (*class in fastdds*), 878
LifespanQosPolicy (*class in fastdds*), 872
listen_socket_buffer_size (*fastdds.TransportConfigQos property*), 882
liveliness() (*fastdds.DataReaderQos method*), 968
liveliness() (*fastdds.DataWriterQos method*), 933
liveliness() (*fastdds.TopicQos method*), 993
liveliness_changed() (*fastdds.StatusMask static method*), 889
liveliness_lost() (*fastdds.StatusMask static method*), 889
LIVELINESS_QOS_POLICY_ID (*class in fastdds*), 878
LivelinessChangedStatus (*class in fastdds*), 886
LivelinessQosPolicy (*class in fastdds*), 872
load_profiles() (*fastdds.DomainParticipantFactory method*), 911
load_XML_profiles_file() (*fastdds.DomainParticipantFactory method*), 910
load_XML_profiles_string() (*fastdds.DomainParticipantFactory method*), 910
loan() (*fastdds.LoanableCollection method*), 892
loan_sample() (*fastdds.DataWriter method*), 923
LoanableCollection (*class in fastdds*), 891
LOCATOR_ADDRESS_INVALID (*C macro*), 753
LOCATOR_INVALID (*C macro*), 753
LOCATOR_KIND_INVALID (*C macro*), 753
LOCATOR_KIND_RESERVED (*C macro*), 753
LOCATOR_KIND_SHM (*C macro*), 754
LOCATOR_KIND_TCPv4 (*C macro*), 754
LOCATOR_KIND_TCPv6 (*C macro*), 754
LOCATOR_KIND_UDPv4 (*C macro*), 753
LOCATOR_KIND_UDPv6 (*C macro*), 754
LOCATOR_PORT_INVALID (*C macro*), 753
lookup_content_filter_factory() (*fastdds.DomainParticipant method*), 903
lookup_datareader() (*fastdds.Subscriber method*), 979
lookup_datawriter() (*fastdds.Publisher method*), 942
lookup_instance() (*fastdds.DataReader method*), 951
lookup_instance() (*fastdds.DataWriter method*), 924
lookup_participant() (*fastdds.DomainParticipantFactory method*), 911
lookup_participants() (*fastdds.DomainParticipantFactory method*), 911
lookup_topicdescription() (*fastdds.DomainParticipant method*), 903

M

m_force_type_validation (*fastdds.TypeConsistencyEnforcementQosPolicy property*), 883
m_ignore_member_names (*fastdds.TypeConsistencyEnforcementQosPolicy property*), 883
m_ignore_sequence_bounds (*fastdds.TypeConsistencyEnforcementQosPolicy property*), 883

m_ignore_string_bounds (*fastdds.TypeConsistencyEnforcementQosPolicy property*), 883
m_isGetKeyDefined (*fastdds.TopicDataType property*), 986
m_kind (*fastdds.TypeConsistencyEnforcementQosPolicy property*), 884
m_prevent_type_widening (*fastdds.TypeConsistencyEnforcementQosPolicy property*), 884
m_type_identifier (*fastdds.TypeIdV1 property*), 996
m_type_object (*fastdds.TypeObjectV1 property*), 997
m_typeSize (*fastdds.TopicDataType property*), 986
m_value (*fastdds.DataRepresentationQosPolicy property*), 863
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS (*class in fastdds*), 873
MANUAL_BY_TOPIC_LIVELINESS_QOS (*class in fastdds*), 873
matched_publisher_allocation (*fastdds.ReaderResourceLimitsQos property*), 974
matched_subscriber_allocation (*fastdds.WriterResourceLimitsQos property*), 885
MatchedStatus (*class in fastdds*), 887
max_blocking_time (*fastdds.ReliabilityQosPolicy property*), 879
max_domains() (*fastdds.DataSharingQosPolicy method*), 865
max_instances (*fastdds.DurabilityServiceQosPolicy property*), 869
max_instances (*fastdds.ResourceLimitsQosPolicy property*), 880
max_msg_size_no_frag (*fastdds.TransportConfigQos property*), 882
max_samples (*fastdds.DurabilityServiceQosPolicy property*), 869
max_samples (*fastdds.ResourceLimitsQosPolicy property*), 880
max_samples_per_instance (*fastdds.DurabilityServiceQosPolicy property*), 869
max_samples_per_instance (*fastdds.ResourceLimitsQosPolicy property*), 880
max_samples_per_read (*fastdds.ReaderResourceLimitsQos property*), 974
max_size() (*fastdds.PartitionQosPolicy method*), 875
maximum() (*fastdds.LoanableCollection method*), 892
minimum_separation (*fastdds.TimeBasedFilterQosPolicy property*), 881
multicast_locator_list (*fastdds.RTPSEndpointQos property*), 881

N

name() (*fastdds.DomainParticipantQos method*), 917
name() (*fastdds.Partition_t method*), 874
names() (*fastdds.PartitionQosPolicy method*), 875
netmask_filter (*fastdds.TransportConfigQos property*), 882
new_remote_endpoint_discovered() (*fastdds.DomainParticipant method*), 903
NEW_VIEW_STATE (*class in fastdds*), 983
NEXT_QOS_POLICY_ID (*class in fastdds*), 879
no_writers_generation_count (*fastdds.SampleInfo property*), 975
none() (*fastdds.StatusMask static method*), 889
not_alive_count (*fastdds.LivelinessChangedStatus property*), 887
not_alive_count_change (*fastdds.LivelinessChangedStatus property*), 887
NOT_ALIVE_DISPOSED_INSTANCE_STATE (*class in fastdds*), 973
NOT_ALIVE_NO_WRITERS_INSTANCE_STATE (*class in fastdds*), 973
NOT_NEW_VIEW_STATE (*class in fastdds*), 983
NOT_READ_SAMPLE_STATE (*class in fastdds*), 975
NOT_REJECTED (*class in fastdds*), 889
notify_datareaders() (*fastdds.Subscriber method*), 979

O

OFF (*class in fastdds*), 866
off() (*fastdds.DataSharingQosPolicy method*), 865
offered_deadline_missed() (*fastdds.StatusMask static method*), 890
offered_incompatible_qos() (*fastdds.StatusMask static method*), 890
ON (*class in fastdds*), 866
on() (*fastdds.DataSharingQosPolicy method*), 865
on_data_available() (*fastdds.DataReaderListener method*), 962
on_data_on_readers() (*fastdds.SubscriberListener method*), 981
on_inconsistent_topic() (*fastdds.TopicListener method*), 989
on_liveliness_changed() (*fastdds.DataReaderListener method*), 962
on_liveliness_lost() (*fastdds.DataWriterListener method*), 927
on_offered_deadline_missed() (*fastdds.DataWriterListener method*), 927
on_offered_incompatible_qos() (*fastdds.DataWriterListener method*), 927

on_participant_discovery() (*fastdds.DomainParticipantListener method*), 913
on_publication_matched() (*fastdds.DataWriterListener method*), 928
on_publisher_discovery() (*fastdds.DomainParticipantListener method*), 913
on_requested_deadline_missed() (*fastdds.DataReaderListener method*), 962
on_requested_incompatible_qos() (*fastdds.DataReaderListener method*), 962
on_sample_lost() (*fastdds.DataReaderListener method*), 962
on_sample_rejected() (*fastdds.DataReaderListener method*), 962
on_subscriber_discovery() (*fastdds.DomainParticipantListener method*), 914
on_subscription_matched() (*fastdds.DataReaderListener method*), 962
on_type_dependencies_reply() (*fastdds.DomainParticipantListener method*), 914
on_type_discovery() (*fastdds.DomainParticipantListener method*), 914
on_type_information_received() (*fastdds.DomainParticipantListener method*), 914
on_unacknowledged_sample_removed() (*fastdds.DataWriterListener method*), 928
ordered_access (*fastdds.PresentationQosPolicy property*), 876
outstanding_reads_allocation (*fastdds.ReaderResourceLimitsQos property*), 974
ownership() (*fastdds.DataReaderQos method*), 968
ownership() (*fastdds.DataWriterQos method*), 933
ownership() (*fastdds.TopicQos method*), 994
OWNERSHIP_QOS_POLICY_ID (*class in fastdds*), 878
ownership_strength() (*fastdds.DataWriterQos method*), 934
OwnershipQosPolicy (*class in fastdds*), 873
OWNERSHIPSTRENGTH_QOS_POLICY_ID (*class in fastdds*), 878
OwnershipStrengthQosPolicy (*class in fastdds*), 874

P

participant_id (*fastdds.WireProtocolConfigQos property*), 885
PARTICIPANTRESOURCELIMITS_QOS_POLICY_ID (*class in fastdds*), 878
partition() (*fastdds.PublisherQos method*), 945
partition() (*fastdds.SubscriberQos method*), 982
PARTITION_QOS_POLICY_ID (*class in fastdds*), 878
Partition_t (*class in fastdds*), 874
PartitionQosPolicy (*class in fastdds*), 874
period (*fastdds.DeadlineQosPolicy property*), 866
PERSISTENT_DURABILITY_QOS (*class in fastdds*), 868
PL_CDR_BE (*C macro*), 773
PL_CDR_LE (*C macro*), 773
policies (*fastdds.IncompatibleQosStatus property*), 886
policy_id (*fastdds.QosPolicyCount property*), 888
port (*fastdds.WireProtocolConfigQos property*), 885
prefix (*fastdds.WireProtocolConfigQos property*), 885
presentation() (*fastdds.PublisherQos method*), 945
presentation() (*fastdds.SubscriberQos method*), 982
PRESENTATION_QOS_POLICY_ID (*class in fastdds*), 878
PresentationQosPolicy (*class in fastdds*), 876
properties() (*fastdds.DataReaderQos method*), 969
properties() (*fastdds.DataWriterQos method*), 934
properties() (*fastdds.DomainParticipantQos method*), 917
PROPERTYPOLICY_QOS_POLICY_ID (*class in fastdds*), 878
publication_handle (*fastdds.SampleInfo property*), 975
publication_matched() (*fastdds.StatusMask static method*), 890
PublicationMatchedStatus (*class in fastdds*), 887
publish_mode() (*fastdds.DataWriterQos method*), 935
Publisher (*class in fastdds*), 939
PUBLISHER_QOS_DEFAULT (*class in fastdds*), 946
PublisherListener (*class in fastdds*), 944
PublisherQos (*class in fastdds*), 944
PUBLISHMODE_QOS_POLICY_ID (*class in fastdds*), 878
PublishModeQosPolicy (*class in fastdds*), 877
push_back() (*fastdds.PartitionQosPolicy method*), 875

Q

QosPolicy (*class in fastdds*), 877
QosPolicyCount (*class in fastdds*), 888

R

read() (*fastdds.DataReader method*), 951
read_instance() (*fastdds.DataReader method*), 954
read_next_instance() (*fastdds.DataReader method*), 954
read_next_instance_w_condition() (*fastdds.DataReader method*), 956
read_next_sample() (*fastdds.DataReader method*), 956
READ_SAMPLE_STATE (*class in fastdds*), 975
read_w_condition() (*fastdds.DataReader method*), 957
reader_data_lifecycle() (*fastdds.DataReaderQos method*), 969

reader_filters_allocation (*fastdds.WriterResourceLimitsQos property*), 885
reader_resource_limits() (*fastdds.DataReaderQos method*), 970
READERDATALIFECYCLE_QOS_POLICY_ID (*class in fastdds*), 878
ReaderDataLifecycleQosPolicy (*class in fastdds*), 879
READERRESOURCELIMITS_QOS_POLICY_ID (*class in fastdds*), 878
ReaderResourceLimitsQos (*class in fastdds*), 974
reception_timestamp (*fastdds.SampleInfo property*), 975
register_content_filter_factory() (*fastdds.DomainParticipant method*), 903
register_instance() (*fastdds.DataWriter method*), 924
register_instance_w_timestamp() (*fastdds.DataWriter method*), 924
register_remote_type() (*fastdds.DomainParticipant method*), 904
register_type() (*fastdds.DomainParticipant method*), 905
register_type() (*fastdds.TypeSupport method*), 999
REJECTED_BY_INSTANCES_LIMIT (*class in fastdds*), 889
REJECTED_BY_SAMPLES_LIMIT (*class in fastdds*), 889
REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT (*class in fastdds*), 889
related_sample_identity (*fastdds.SampleInfo property*), 975
reliability() (*fastdds.DataReaderQos method*), 970
reliability() (*fastdds.DataWriterQos method*), 935
reliability() (*fastdds.TopicQos method*), 994
RELIABILITY_QOS_POLICY_ID (*class in fastdds*), 878
ReliabilityQosPolicy (*class in fastdds*), 879
reliable_reader_qos() (*fastdds.DataReaderQos method*), 971
RELIABLE_RELIABILITY_QOS (*class in fastdds*), 880
reliable_writer_qos() (*fastdds.DataWriterQos method*), 936
remote_locator_list (*fastdds.RTPSEndpointQos property*), 881
representation (*fastdds.TypeConsistencyQos property*), 983
requested_deadline_missed() (*fastdds.StatusMask static method*), 890
requested_incompatible_qos() (*fastdds.StatusMask static method*), 890
resize() (*fastdds.GenericDataQosPolicy method*), 870
resource_limits() (*fastdds.DataReaderQos method*), 971
resource_limits() (*fastdds.DataWriterQos method*), 936
resource_limits() (*fastdds.TopicQos method*), 995
RESOURCELIMITS_QOS_POLICY_ID (*class in fastdds*), 878
ResourceLimitsQosPolicy (*class in fastdds*), 880
resume_publications() (*fastdds.Publisher method*), 942
return_loan() (*fastdds.DataReader method*), 957
RTPSENDPOINT_QOS_POLICY_ID (*class in fastdds*), 878
RTPSEndpointQos (*class in fastdds*), 881
RTPSMESSAGE_COMMON_DATA_PAYLOAD_SIZE (*C macro*), 743
RTPSMESSAGE_COMMON_RTPS_PAYLOAD_SIZE (*C macro*), 743
RTPSMESSAGE_DATA_EXTRA_INLINEQOS_SIZE (*C macro*), 743
RTPSMESSAGE_DATA_MIN_LENGTH (*C macro*), 743
RTPSMESSAGE_DEFAULT_SIZE (*C macro*), 743
RTPSMESSAGE_HEADER_SIZE (*C macro*), 743
RTPSMESSAGE_INFOTS_SIZE (*C macro*), 743
RTPSMESSAGE_OCTETSTOINLINEQOS_DATAFRAGSUBMSG (*C macro*), 743
RTPSMESSAGE_OCTETSTOINLINEQOS_DATASUBMSG (*C macro*), 743
RTPSMESSAGE_SUBMESSAGEHEADER_SIZE (*C macro*), 743
RTPSRELIABLEREADER_QOS_POLICY_ID (*class in fastdds*), 878
RTPSReliableReaderQos (*class in fastdds*), 974
RTPSRELIABLEWRITER_QOS_POLICY_ID (*class in fastdds*), 878
RTPSReliableWriterQos (*class in fastdds*), 946

S

sample_identity (*fastdds.SampleInfo property*), 975
sample_infos_allocation (*fastdds.ReaderResourceLimitsQos property*), 974
sample_lost() (*fastdds.StatusMask static method*), 890
sample_rank (*fastdds.SampleInfo property*), 975
sample_rejected() (*fastdds.StatusMask static method*), 890
sample_state (*fastdds.SampleInfo property*), 975
SampleInfo (*class in fastdds*), 974
SampleRejectedStatus (*class in fastdds*), 888
send_always() (*fastdds.QosPolicy method*), 877
send_socket_buffer_size (*fastdds.TransportConfigQos property*), 882
serialize() (*fastdds.TopicDataType method*), 987
serialize() (*fastdds.TypeSupport method*), 999
service_cleanup_delay (*fastdds.DurabilityServiceQosPolicy property*), 869
set() (*fastdds.TypeSupport method*), 1000

set_default_datareader_qos() (*fastdds.Subscriber method*), 979
set_default_datawriter_qos() (*fastdds.Publisher method*), 942
set_default_participant_qos() (*fastdds.DomainParticipantFactory method*), 911
set_default_publisher_qos() (*fastdds.DomainParticipant method*), 905
set_default_subscriber_qos() (*fastdds.DomainParticipant method*), 905
set_default_topic_qos() (*fastdds.DomainParticipant method*), 906
set_listener() (*fastdds.DataReader method*), 958
set_listener() (*fastdds.DataWriter method*), 924
set_listener() (*fastdds.DomainParticipant method*), 906
set_listener() (*fastdds.Publisher method*), 942
set_listener() (*fastdds.Subscriber method*), 980
set_listener() (*fastdds.Topic method*), 984
set_max_domains() (*fastdds.DataSharingQosPolicy method*), 866
set_max_size() (*fastdds.GenericDataQosPolicy method*), 870
set_max_size() (*fastdds.PartitionQosPolicy method*), 875
set_qos() (*fastdds.DataReader method*), 958
set_qos() (*fastdds.DataWriter method*), 924
set_qos() (*fastdds.DomainParticipant method*), 906
set_qos() (*fastdds.DomainParticipantFactory method*), 911
set_qos() (*fastdds.Publisher method*), 943
set_qos() (*fastdds.Subscriber method*), 980
set_qos() (*fastdds.Topic method*), 984
setName() (*fastdds.TopicDataType method*), 987
setNames() (*fastdds.PartitionQosPolicy method*), 875
setup_transports() (*fastdds.DomainParticipantQos method*), 918
setValue() (*fastdds.GenericDataQosPolicy method*), 870
SHARED_OWNERSHIP_QOS (*class in fastdds*), 873
shm_directory() (*fastdds.DataSharingQosPolicy method*), 866
shm_watchdog_thread() (*fastdds.DomainParticipantFactoryQos method*), 913
size() (*fastdds.Partition_t method*), 874
size() (*fastdds.PartitionQosPolicy method*), 876
source_timestamp (*fastdds.SampleInfo property*), 975
StatusMask (*class in fastdds*), 889
Subscriber (*class in fastdds*), 975
SUBSCRIBER_QOS_DEFAULT (*class in fastdds*), 983
SubscriberListener (*class in fastdds*), 981
SubscriberQos (*class in fastdds*), 981
subscription_matched() (*fastdds.StatusMask static method*), 890
SubscriptionMatchedStatus (*class in fastdds*), 891
suspend_publications() (*fastdds.Publisher method*), 943
SYNCHRONOUS_PUBLISH_MODE (*class in fastdds*), 877

T

take() (*fastdds.DataReader method*), 958
take_instance() (*fastdds.DataReader method*), 959
take_next_instance() (*fastdds.DataReader method*), 959
take_next_instance_w_condition() (*fastdds.DataReader method*), 960
take_next_sample() (*fastdds.DataReader method*), 960
take_w_condition() (*fastdds.DataReader method*), 961
thisown (*fastdds.BaseStatus property*), 885
thisown (*fastdds.DataReader property*), 961
thisown (*fastdds.DataReaderListener property*), 963
thisown (*fastdds.DataReaderQos property*), 972
thisown (*fastdds.DataRepresentationQosPolicy property*), 863
thisown (*fastdds.DataSharingQosPolicy property*), 866
thisown (*fastdds.DataWriter property*), 925
thisown (*fastdds.DataWriterListener property*), 928
thisown (*fastdds.DataWriterQos property*), 937
thisown (*fastdds.DeadlineMissedStatus property*), 886
thisown (*fastdds.DeadlineQosPolicy property*), 866
thisown (*fastdds.DestinationOrderQosPolicy property*), 867
thisown (*fastdds.DisablePositiveACKsQosPolicy property*), 867
thisown (*fastdds.DomainEntity property*), 863
thisown (*fastdds.DomainParticipant property*), 907
thisown (*fastdds.DomainParticipantFactory property*), 912
thisown (*fastdds.DomainParticipantFactoryQos property*), 913
thisown (*fastdds.DomainParticipantListener property*), 914
thisown (*fastdds.DomainParticipantQos property*), 918
thisown (*fastdds.DurabilityQosPolicy property*), 868
thisown (*fastdds.DurabilityServiceQosPolicy property*), 869
thisown (*fastdds.Entity property*), 863
thisown (*fastdds.EntityFactoryQosPolicy property*), 869
thisown (*fastdds.GenericDataQosPolicy property*), 871
thisown (*fastdds.GroupDataQosPolicy property*), 871
thisown (*fastdds.HistoryQosPolicy property*), 871
thisown (*fastdds.IncompatibleQosStatus property*), 886
thisown (*fastdds.LatencyBudgetQosPolicy property*), 872

thisown (*fastdds.LifespanQosPolicy property*), 872
thisown (*fastdds.LivelinessChangedStatus property*), 887
thisown (*fastdds.LivelinessQosPolicy property*), 873
thisown (*fastdds.LoanableCollection property*), 892
thisown (*fastdds.MatchedStatus property*), 887
thisown (*fastdds.OwnershipQosPolicy property*), 873
thisown (*fastdds.OwnershipStrengthQosPolicy property*), 874
thisown (*fastdds.Partition_t property*), 874
thisown (*fastdds.PartitionQosPolicy property*), 876
thisown (*fastdds.PresentationQosPolicy property*), 876
thisown (*fastdds.PublicationMatchedStatus property*), 887
thisown (*fastdds.Publisher property*), 943
thisown (*fastdds.PublisherListener property*), 944
thisown (*fastdds.PublisherQos property*), 946
thisown (*fastdds.PublishModeQosPolicy property*), 877
thisown (*fastdds.QosPolicy property*), 877
thisown (*fastdds.QosPolicyCount property*), 888
thisown (*fastdds.ReaderDataLifecycleQosPolicy property*), 879
thisown (*fastdds.ReaderResourceLimitsQos property*), 974
thisown (*fastdds.ReliabilityQosPolicy property*), 879
thisown (*fastdds.ResourceLimitsQosPolicy property*), 880
thisown (*fastdds.RTPSEndpointQos property*), 881
thisown (*fastdds.RTPSReliableReaderQos property*), 974
thisown (*fastdds.RTPSReliableWriterQos property*), 946
thisown (*fastdds.SampleInfo property*), 975
thisown (*fastdds.SampleRejectedStatus property*), 888
thisown (*fastdds.StatusMask property*), 890
thisown (*fastdds.Subscriber property*), 980
thisown (*fastdds.SubscriberListener property*), 981
thisown (*fastdds.SubscriberQos property*), 983
thisown (*fastdds.SubscriptionMatchedStatus property*), 891
thisown (*fastdds.TimeBasedFilterQosPolicy property*), 881
thisown (*fastdds.Topic property*), 984
thisown (*fastdds.TopicDataQosPolicy property*), 882
thisown (*fastdds.TopicDataType property*), 987
thisown (*fastdds.TopicDescription property*), 989
thisown (*fastdds.TopicListener property*), 989
thisown (*fastdds.TopicQos property*), 995
thisown (*fastdds.TransportConfigQos property*), 882
thisown (*fastdds.TransportPriorityQosPolicy property*), 883
thisown (*fastdds.TypeConsistencyEnforcementQosPolicy property*), 884
thisown (*fastdds.TypeConsistencyQos property*), 983
thisown (*fastdds.TypeIdV1 property*), 996
thisown (*fastdds.TypeInformation property*), 997
thisown (*fastdds.TypeObjectV1 property*), 997
thisown (*fastdds.TypeSupport property*), 1000
thisown (*fastdds.UserDataQosPolicy property*), 884
thisown (*fastdds.WireProtocolConfigQos property*), 885
thisown (*fastdds.WriterDataLifecycleQosPolicy property*), 885
thisown (*fastdds.WriterResourceLimitsQos property*), 885
time_based_filter() (*fastdds.DataReaderQos method*), 972
TIME_T_INFINITE_NANOSECONDS (*C macro*), 775
TIME_T_INFINITE_SECONDS (*C macro*), 775
TIMEBASEDFILTER_QOS_POLICY_ID (*class in fastdds*), 878
TimeBasedFilterQosPolicy (*class in fastdds*), 881
timed_events_thread() (*fastdds.DomainParticipantQos method*), 918
times (*fastdds.RTPSReliableReaderQos property*), 974
times (*fastdds.RTPSReliableWriterQos property*), 946
Topic (*class in fastdds*), 984
topic_data() (*fastdds.TopicQos method*), 995
TOPIC_PRESENTATION_QOS (*class in fastdds*), 876
TOPIC_QOS_DEFAULT (*class in fastdds*), 996
TOPICDATA_QOS_POLICY_ID (*class in fastdds*), 878
TopicDataQosPolicy (*class in fastdds*), 882
TopicDataType (*class in fastdds*), 985
TopicDescription (*class in fastdds*), 989
TopicListener (*class in fastdds*), 989
TopicQos (*class in fastdds*), 990
total_count (*fastdds.BaseStatus property*), 885
total_count (*fastdds.DeadlineMissedStatus property*), 886
total_count (*fastdds.IncompatibleQosStatus property*), 886
total_count (*fastdds.MatchedStatus property*), 887
total_count (*fastdds.SampleRejectedStatus property*), 888
total_count_change (*fastdds.BaseStatus property*), 885
total_count_change (*fastdds.DeadlineMissedStatus property*), 886
total_count_change (*fastdds.IncompatibleQosStatus property*), 886
total_count_change (*fastdds.MatchedStatus property*), 887
total_count_change (*fastdds.SampleRejectedStatus property*), 888
TRANSIENT_DURABILITY_QOS (*class in fastdds*), 868
TRANSIENT_LOCAL_DURABILITY_QOS (*class in fastdds*), 868
transport() (*fastdds.DomainParticipantQos method*), 918
transport_priority() (*fastdds.DataWriterQos*

method), 937
transport_priority() (*fastdds.TopicQos method*), 996
TRANSPORTCONFIG_QOS_POLICY_ID (*class in fastdds*), 878
TransportConfigQos (*class in fastdds*), 882
TRANSPORTPRIORITY_QOS_POLICY_ID (*class in fastdds*), 878
TransportPriorityQosPolicy (*class in fastdds*), 883
type() (*fastdds.DataReader method*), 961
type_consistency (*fastdds.TypeConsistencyQos property*), 983
type_consistency() (*fastdds.DataReaderQos method*), 972
type_identifier() (*fastdds.TopicDataType method*), 987
type_information (*fastdds.TypeInformation property*), 997
type_information() (*fastdds.TopicDataType method*), 987
type_object() (*fastdds.TopicDataType method*), 988
TYPECONSISTENCY_QOS_POLICY_ID (*class in fastdds*), 878
TYPECONSISTENCYENFORCEMENT_QOS_POLICY_ID (*class in fastdds*), 878
TypeConsistencyEnforcementQosPolicy (*class in fastdds*), 883
TypeConsistencyQos (*class in fastdds*), 983
TypeIdV1 (*class in fastdds*), 996
TypeInformation (*class in fastdds*), 997
TYPELOOKUP_DATA_MAX_SIZE (*C macro*), 800
TypeObjectV1 (*class in fastdds*), 997
TypeSupport (*class in fastdds*), 998

U

unicast_locator_list (*fastdds.RTPSEndpointQos property*), 881
unloan() (*fastdds.LoanableCollection method*), 892
unregister_content_filter_factory() (*fastdds.DomainParticipant method*), 907
unregister_instance() (*fastdds.DataWriter method*), 925
unregister_instance_w_timestamp() (*fastdds.DataWriter method*), 925
unregister_type() (*fastdds.DomainParticipant method*), 907
use_builtin_transports (*fastdds.TransportConfigQos property*), 882
user_data() (*fastdds.DataReaderQos method*), 973
user_data() (*fastdds.DataWriterQos method*), 937
user_data() (*fastdds.DomainParticipantQos method*), 919
user_defined_id (*fastdds.RTPSEndpointQos property*), 881
user_transports (*fastdds.TransportConfigQos property*), 882
USERDATA_QOS_POLICY_ID (*class in fastdds*), 878
UserDataQosPolicy (*class in fastdds*), 884

V

valid_data (*fastdds.SampleInfo property*), 975
value (*fastdds.OwnershipStrengthQosPolicy property*), 874
value (*fastdds.TransportPriorityQosPolicy property*), 883
view_state (*fastdds.SampleInfo property*), 975
VOLATILE_DURABILITY_QOS (*class in fastdds*), 868

W

wait_for_acknowledgments() (*fastdds.DataWriter method*), 925
wait_for_acknowledgments() (*fastdds.Publisher method*), 943
wait_for_historical_data() (*fastdds.DataReader method*), 961
wait_for_unread_message() (*fastdds.DataReader method*), 961
wire_protocol() (*fastdds.DomainParticipantQos method*), 919
WIREPROTOCOLCONFIG_QOS_POLICY_ID (*class in fastdds*), 879
WireProtocolConfigQos (*class in fastdds*), 884
write() (*fastdds.DataWriter method*), 926
write_w_timestamp() (*fastdds.DataWriter method*), 927
writer_data_lifecycle() (*fastdds.DataWriterQos method*), 938
writer_resource_limits() (*fastdds.DataWriterQos method*), 938
WRITERDATALIFECYCLE_QOS_POLICY_ID (*class in fastdds*), 878
WriterDataLifecycleQosPolicy (*class in fastdds*), 885
WRITERRESOURCELIMITS_QOS_POLICY_ID (*class in fastdds*), 879
WriterResourceLimitsQos (*class in fastdds*), 885

X

XCDR2_DATA_REPRESENTATION (*class in fastdds*), 863
XCDR_DATA_REPRESENTATION (*class in fastdds*), 863
XML_DATA_REPRESENTATION (*class in fastdds*), 863