Fast DDS Documentation

Release 2.3.1

eProsima

Apr 29, 2021

INTRODUCTION

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).

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.
- **Throughput controllers.** We support user-configurable throughput 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).

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.
- **Throughput controllers.** We support user-configurable throughput 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.

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.

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.
- `fastrtpsgen`, 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
```

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:

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.

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.
- `fastrtpsgen`, 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.

6.10 Linux 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:

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*
- *Gtest* [optional]

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
```

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. For a detailed description of the Gtest installation process, please refer to the Gtest Installation Guide.

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 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
```

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:** 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
mkdir src
vcs import src < fastrtps.repos
```

3. Build the packages:

```
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:

```
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
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:

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

6.10.2 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

In order to compile *Fast DDS-Gen*, the following packages need to be installed in the system:

- *Java JDK*
- *Gradle*

Java JDK

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

Gradle

Gradle is an open-source build automation tool. Download and install the last stable version of Gradle in the preferred way.

Compiling Fast DDS-Gen

Once the requirements above are met, compile *Fast DDS-Gen* by following the steps below:

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

Note: If errors occur during compilation or you do not wish to install gradle, an executable script is included which will download a gradle temporarily for the compilation step.

```
./gradlew 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.

6.11 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:

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]

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. For a detailed description of the Gtest installation process, please refer to the Gtest Installation Guide.

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. Download and install the latest OpenSSL version for Windows at this link. After installing, add the environment variable `OPENSSL_ROOT_DIR` pointing to the installation root directory.

For example:

```
OPENSSL_ROOT_DIR=C:\Program Files\OpenSSL-Win64
```

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 ~\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
cd foonathan_memory_vendor
mkdir build && cd build
cmake ..  -DBUILD_SHARED_LIBS=ON
cmake --build . --target install
```

   - Fast CDR

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

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`.

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-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

In order to compile *Fast DDS-Gen*, the following packages need to be installed in the system:

- *Java JDK*
- *Gradle*

Java JDK

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

Gradle

Gradle is an open-source build automation tool. Download and install the last stable version of Gradle in the preferred way.

Compiling Fast DDS-Gen

Once the requirements above are met, install *Fast DDS-Gen* by following the steps below:

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

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]

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. For a detailed description of the Gtest installation process, please refer to the Gtest Installation Guide.

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

In order to compile *Fast DDS-Gen*, the following packages need to be installed in the system:

- *Java JDK*
- *Gradle*

Java JDK

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

Gradle

Gradle is an open-source build automation tool. Download and install the last stable version of Gradle in the preferred way. with Homebrew it would be running the command:

```
brew install gradle
```

Note: If errors occur during compilation or you do not wish to install gradle, an executable script is included which will download gradle temporarily for the compilation step.

```
./gradlew assemble
```

Compiling Fast DDS-Gen

Once the requirements above are met, compile *Fast DDS-Gen* by following the steps below:

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

6.13.1 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_GTEST is set to ON if	ON OFF	OFF

6.13.2 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 the 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 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

6.13.3 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	ON 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	ON 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	ON 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	android-ifaddrs is an 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.

6.13.4 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	Activate 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
EPROSIMA_GTEST	Activates special set of GTEST_ROOT, i.e. the root directory of the GoogleTest installation.	ON OFF	OFF
EPROSIMA_GMOCK	Activates special set of GMOCK_ROOT, i.e. the root directory of the GoogleTest C++ mocking framework installation. In the latest version of GoogleTest, GoogleMock is integrated into it.	ON OFF	OFF
FASTRTPS_API_TESTS	Enables 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	Enables 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_UDPV6	Disables UDPv6 tests.	ON OFF	OFF

6.14 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.

6.14.1 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.

6.14.2 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.

6.14.3 Writing a simple publisher and subscriber application

This section details how to create an simple Fast DDS application with a publisher and a subscriber 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

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
        ├── 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. The way we will make these accessible from the workspace depends on the installation procedure we have followed in the Installation Manual.

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

If you have followed the Colcon installation there are several ways to import the libraries. If you want these to be accessible only from the current shell session, run one of the following two commands.

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

If you want these to be accessible from any session, you can add the Fast DDS installation directory to your `$PATH` variable in the shell configuration files running the following command.

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

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 3.12.4)

if(NOT CMAKE_VERSION VERSION_LESS 3.0)
    cmake_policy(SET CMP0048 NEW)
endif()

project(DDSHelloWorld)

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

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

# Set C++11
include(CheckCXXCompilerFlag)
if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_COMPILER_IS_CLANG OR
        CMAKE_CXX_COMPILER_ID MATCHES "Clang")
    check_cxx_compiler_flag(-std=c++11 SUPPORTS_CXX11)
    if(SUPPORTS_CXX11)
        add_compile_options(-std=c++11)
```

(continues on next page)

(continued from previous page)

```
    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 your favorite 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/fastrtpsgen HelloWorld.idl
```

This must have generated the following files:

- HelloWorld.cxx: HelloWorld type definition.
- HelloWorld.h: Header file for HelloWorld.cxx.
- HelloWorldPubSubTypes.cxx: Serialization and Deserialization code for the HelloWorld type.
- HelloWorldPubSubTypes.h: Header file for HelloWorldPubSubTypes.cxx.

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
```

Now you have the publisher's source code. The publisher is going to send 10 publications under the topic HelloWorld.

```
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:
34
35     HelloWorld hello_;
36
37     DomainParticipant* participant_;
38
```

(continues on next page)

(continued from previous page)

```
    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_;

public:

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

(continues on next page)

(continued from previous page)

```
95
96      virtual ~HelloWorldPublisher()
97      {
98          if (writer_ != nullptr)
99          {
100             publisher_->delete_datawriter(writer_);
101         }
102         if (publisher_ != nullptr)
103         {
104             participant_->delete_publisher(publisher_);
105         }
106         if (topic_ != nullptr)
107         {
108             participant_->delete_topic(topic_);
109         }
110         DomainParticipantFactory::get_instance()->delete_participant(participant_);
111     }
112
113     //!Initialize the publisher
114     bool init()
115     {
116         hello_.index(0);
117         hello_.message("HelloWorld");
118
119         DomainParticipantQos participantQos;
120         participantQos.name("Participant_publisher");
121         participant_ = DomainParticipantFactory::get_instance()->create_participant(0,
    ↪ participantQos);
122
123         if (participant_ == nullptr)
124         {
125             return false;
126         }
127
128         // Register the Type
129         type_.register_type(participant_);
130
131         // Create the publications Topic
132         topic_ = participant_->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_
    ↪QOS_DEFAULT);
133
134         if (topic_ == nullptr)
135         {
136             return false;
137         }
138
139         // Create the Publisher
140         publisher_ = participant_->create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);
141
142         if (publisher_ == nullptr)
143         {
144             return false;
145         }
146
147         // Create the DataWriter
148         writer_ = publisher_->create_datawriter(topic_, DATAWRITER_QOS_DEFAULT, &
    ↪listener_);
```

(continues on next page)

(continued from previous page)

```
149
150         if (writer_ == nullptr)
151         {
152             return false;
153         }
154         return true;
155     }
156
157     //!Send a publication
158     bool publish()
159     {
160         if (listener_.matched_ > 0)
161         {
162             hello_.index(hello_.index() + 1);
163             writer_->write(&hello_);
164             return true;
165         }
166         return false;
167     }
168
169     //!Run the Publisher
170     void run(
171             uint32_t samples)
172     {
173         uint32_t samples_sent = 0;
174         while (samples_sent < samples)
175         {
176             if (publish())
177             {
178                 samples_sent++;
179                 std::cout << "Message: " << hello_.message() << " with index: " <<␣
↪hello_.index()
180                           << " SENT" << std::endl;
181             }
182             std::this_thread::sleep_for(std::chrono::milliseconds(1000));
183         }
184     }
185 };
186
187 int main(
188         int argc,
189         char** argv)
190 {
191     std::cout << "Starting publisher." << std::endl;
192     int samples = 10;
193
194     HelloWorldPublisher* mypub = new HelloWorldPublisher();
195     if(mypub->init())
196     {
197         mypub->run(static_cast<uint32_t>(samples));
198     }
199
200     delete mypub;
201     return 0;
202 }
```

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*. Is the object responsible for the creation of DataReaders.
- *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>
#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_;
```

(continues on next page)

(continued from previous page)

```
    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 allow us to execute routines in case of an event. The overridden callback *`on_publication_matched`* allows you to define 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 you to track 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(
            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_);
    }
    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);
```

(continues on next page)

(continued from previous page)

```
    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);

    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;
    int samples = 10;

    HelloWorldPublisher* mypub = new HelloWorldPublisher();
    if(mypub->init())
    {
        mypub->run(static_cast<uint32_t>(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 you can build, compile and run the publisher application. From the build directory in the workspace, run the following commands.

```
cmake ..
make
./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
```

Now you have the subscriber's source code. 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.
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 HelloWorldSubscriber.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/subscriber/Subscriber.hpp>
26 #include <fastdds/dds/subscriber/DataReader.hpp>
27 #include <fastdds/dds/subscriber/DataReaderListener.hpp>
28 #include <fastdds/dds/subscriber/qos/DataReaderQos.hpp>
29 #include <fastdds/dds/subscriber/SampleInfo.hpp>
30
31 using namespace eprosima::fastdds::dds;
32
33 class HelloWorldSubscriber
34 {
35 private:
36
37     DomainParticipant* participant_;
38
39     Subscriber* subscriber_;
40
41     DataReader* reader_;
42
43     Topic* topic_;
44
45     TypeSupport type_;
46
```

(continues on next page)

(continued from previous page)

```
    class SubListener : public DataReaderListener
    {
    public:

        SubListener()
            : samples_(0)
        {
        }

        ~SubListener() override
        {
        }

        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:

```

(continues on next page)

(continued from previous page)

```
    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_);
    }

    //!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)
        {
```

(continues on next page)

(continued from previous page)

```
            return false;
        }

        // 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;
    int samples = 10;

    HelloWorldSubscriber* mysub = new HelloWorldSubscriber();
    if(mysub->init())
    {
        mysub->run(static_cast<uint32_t>(samples));
    }

    delete mysub;
    return 0;
}
```

Examining the code

As you have noticed, the source code to implement the subscriber is practically identical to the source code implemented by the publisher. Therefore, we will focus on the main differences between them, without explaining all the code again.

Following the same structure as in the publisher explanation, we start with 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;
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

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_);
}
```

Then we have 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;
    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);
```

(continues on next page)

(continued from previous page)

```
    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;
    int samples = 10;

    HelloWorldSubscriber* mysub = new HelloWorldSubscriber();
    if(mysub->init())
    {
        mysub->run(static_cast<uint32_t>(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 you can build, compile and run the subscriber application. From the build directory in the workspace, run the following commands.

```
cmake ..
make clean && make
./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 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.

6.15.1 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 the implements 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 reads the data from 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.

6.15.2 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 following is a brief overview of how *Fast DDS* multithreading schedule work:

- Main thread: Managed by the application.
- Event thread: Each DomainParticipant owns one of these. It processes periodic and triggered time events.
- Asynchronous writer thread: This thread manages asynchronous writes for all DomainParticipants. Even for synchronous writers, some forms of communication must be initiated in the background.

- Reception threads: DomainParticipants spawn a thread for each reception channel, where the concept of a channel depends on the transport layer (e.g. a UDP port).

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.

6.15.3 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 a hubs for discovery meta traffic.
- **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: `logInfo`, `logWarning` and `logError`. 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*.

6.16 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.

6.16.1 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()*. Within each callback entity hierarchy remains the same.

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.

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

```
<publisher profile_name="publisher_xml_conf_deadline_profile">
    <qos>
        <deadline>
            <period>
                <sec>1</sec>
                <nanosec>0</nanosec>
            </period>
        </deadline>
    </qos>
</publisher>

<subscriber profile_name="subscriber_xml_conf_deadline_profile">
    <qos>
        <deadline>
            <period>
                <sec>1</sec>
                <nanosec>0</nanosec>
            </period>
        </deadline>
    </qos>
</subscriber>
```

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.

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

```
<publisher profile_name="publisher_xml_conf_durability_profile">
    <qos>
        <durability>
            <kind>TRANSIENT_LOCAL</kind>
         </durability>
    </qos>
</publisher>

<subscriber profile_name="subscriber_xml_conf_durability_profile">
    <qos>
        <durability>
            <kind>VOLATILE</kind>
        </durability>
    </qos>
</subscriber>
```

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

eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
group_data.data_vec(vec); //Setter function
```

XML

This QoS Policy cannot be configured using XML for the moment.

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*. If the limit is reached, the behavior of the system depends on the *ReliabilityQosPolicy*, if its kind is BEST_EFFORT the older values will be discarded, but if it is RELIABLE the service blocks the DataWriter until the old values are delivered to all existing Subscribers.

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

```
<publisher profile_name="publisher_xml_conf_lifespan_profile">
    <qos>
        <lifespan>
            <duration>
                <sec>5</sec>
                <nanosec>0</nanosec>
            </duration>
        </lifespan>
    </qos>
</publisher>
```

(continues on next page)

(continued from previous page)

```
<subscriber profile_name="subscriber_xml_conf_lifespan_profile">
    <qos>
        <lifespan>
            <duration>
                <sec>5</sec>
                <nanosec>0</nanosec>
            </duration>
        </lifespan>
    </qos>
</subscriber>
```

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 also 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

```
<publisher profile_name="publisher_xml_conf_liveliness_profile">
    <qos>
      <liveliness>
        <announcement_period>
          <sec>0</sec>
          <nanosec>1000000</nanosec>
        </announcement_period>
        <lease_duration>
          <sec>1</sec>
        </lease_duration>
        <kind>AUTOMATIC</kind>
      </liveliness>
    </qos>
</publisher>

<subscriber profile_name="subscriber_xml_conf_liveliness_profile">
    <qos>
        <liveliness>
            <lease_duration>
                <sec>1</sec>
            </lease_duration>
            <kind>AUTOMATIC</kind>
        </liveliness>
    </qos>
</subscriber>
```

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*.

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

This QoS Policy cannot be configured using XML for the moment.

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

This QoS Policy cannot be configured using XML for the moment.

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.

It can be changed on enabled entities.

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

```
<publisher profile_name="pub_partition_example">
<qos>
    <partition>
        <names>
            <name>part1</name>
            <name>part2</name>
        </names>
    </partition>
</qos>
</publisher>
```

(continues on next page)

(continued from previous page)

```
<subscriber profile_name="sub_partition_example">
<qos>
    <partition>
        <names>
            <name>part1</name>
            <name>part2</name>
        </names>
    </partition>
</qos>
</subscriber>
```

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.

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

```
<publisher profile_name="publisher_xml_conf_reliability_profile">
    <qos>
        <reliability>
            <kind>RELIABLE</kind>
            <max_blocking_time>
                <sec>1</sec>
                <nanosec>0</nanosec>
            </max_blocking_time>
        </reliability>
    </qos>
</publisher>

<subscriber profile_name="subscriber_xml_conf_reliability_profile">
    <qos>
        <reliability>
            <kind>BEST_EFFORT</kind>
        </reliability>
    </qos>
</subscriber>
```

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.
- *max_instances*: Controls the maximum number of instances that a 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.
- *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

```
<publisher profile_name="publisher_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>
```

(continues on next page)

(continued from previous page)

```
    </topic>
</publisher>

<subscriber profile_name="subscriber_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>
</subscriber>
```

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 new octet to topic data vector
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
topic_data.data_vec(vec); //Setter Function
```

XML

This QoS Policy cannot be configured using XML for the moment.

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 new octet to topic data vector
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
user_data.data_vec(vec); //Setter Function
```

XML

This QoS Policy cannot be configured using XML for the moment.

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*
- *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

```
<publisher profile_name="publisher_profile_qos_datasharing">
    <qos>
        <data_sharing>
            <kind>AUTOMATIC</kind>
            <domain_ids>
                <domainId>0x1234</domainId>
                <domainId>0xABCD</domainId>
            </domain_ids>
        </data_sharing>
    </qos>
</publisher>

<subscriber profile_name="subscriber_profile_qos_datasharing">
    <qos>
        <data_sharing>
            <kind>AUTOMATIC</kind>
            <domain_ids>
                <domainId>0x1234</domainId>
                <domainId>0xABCD</domainId>
            </domain_ids>
        </data_sharing>
    </qos>
</subscriber>
```

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 a sufficient 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.

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.

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
//Change the duration to 1 second
disable_acks.duration = {1, 0};
```

XML

```
<publisher profile_name="publisher_xml_conf_disable_positive_acks_profile">
    <qos>
        <disablePositiveAcks>
            <enabled>true</enabled>
            <duration>
                <sec>1</sec>
            </duration>
        </disablePositiveAcks>
    </qos>
</publisher>

<subscriber profile_name="subscriber_xml_conf_disable_positive_acks_profile">
    <qos>
        <disablePositiveAcks>
            <enabled>true</enabled>
        </disablePositiveAcks>
    </qos>
</subscriber>
```

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*

- *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.

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.

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;
```

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>
```

(continues on next page)

(continued from previous page)

```
                <max_properties>512</max_properties>
            </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*.

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"));
```

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>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

PublishModeQosPolicy

This QoS Policy configures how the *DataWriter* sends the data. See *PublishModeQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*PublishModeQosPolicyKind*	*SYNCHRONOUS_PUBLISH_MODE*

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

```
<publisher profile_name="publisher_profile_qos_publishmode">
    <qos>
        <publishMode>
            <kind>ASYNCHRONOUS</kind>
        </publishMode>
    </qos>
</publisher>
```

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

```
<subscriber 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>
</subscriber>
```

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_t*	Empty List
multicast_locator_list	*LocatorList_t*	Empty List
remote_locator_list	*LocatorList_t*	Empty List
user_defined_id	int16_t	-1
entity_id	int16_t	-1
history_memory_policy	*MemoryManagementPolicy*	*PREALLOCATED_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.
- *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);
```

(continues on next page)

(continued from previous page)

```
//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);
//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

```
<publisher profile_name="publisher_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>
    </multicastLocatorList>
    <!-- The history memory policy is changed to DYNAMIC_RESERVE -->
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</publisher>

<subscriber profile_name="subscriber_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>
```

(continues on next page)

(continued from previous page)

```
        </locator>
    </multicastLocatorList>
    <historyMemoryPolicy>PREALLOCATED_WITH_REALLOC</historyMemoryPolicy>
</subscriber>
```

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.

It cannot be changed 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
```

(continues on next page)

(continued from previous page)

```
//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

```
<subscriber profile_name="sub_profile_name">
    <times> <!-- readerTimesType -->
        <initialAcknackDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>70</nanosec>
        </initialAcknackDelay>
        <heartbeatResponseDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>5</nanosec>
        </heartbeatResponseDelay>
    </times>
    <!--You can also change the values of DisablePositiveACKsQosPolicy.-->
    <!--See DisablePositiveACKsQosPolicy section for further details-->
</subscriber>
```

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*

- *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.

Note: This QoS Policy concerns to *DataWriter* entities.

It cannot be changed 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.

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;
```

XML

```
<publisher profile_name="pub_profile_name">
    <times> <!-- writerTimesType -->
        <initialHeartbeatDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>20</nanosec>
        </initialHeartbeatDelay>
        <heartbeatPeriod> <!-- DURATION -->
            <sec>5</sec>
```

(continues on next page)

(continued from previous page)

```
            <nanosec>0</nanosec>
        </heartbeatPeriod>
        <nackResponseDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>10</nanosec>
        </nackResponseDelay>
        <nackSupressionDuration> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>20</nanosec>
        </nackSupressionDuration>
    </times>
    <!--You can also change the values of DisablePositiveACKsQosPolicy.-->
    <!--See DisablePositiveACKsQosPolicy section for further details-->
</publisher>
```

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>
```

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	fastrtps::rtps::GuidPrefix_t	0
participant_id	int32_t	-1
builtin	*BuiltinAttributes*	
throughput_controller	*ThroughputControllerDescriptor*	
default_unicast_locator_list	*LocatorList_t*	Empty List
default_multicast_locator_list	*LocatorList_t*	Empty List

- *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.
- *throughput_controller*: It allows the configuration of the throughput settings.
- *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.

Note: This QoS Policy concerns to DomainParticipant entities.

It cannot be changed on enabled entities.

ThroughputControllerDescriptor

This structure allows to limit the output bandwidth. See *ThroughputControllerDescriptor*.

List of structure members:

Member Name	Type
bytesPerPeriod	uint32_t
periodMillisecs	uint32_t

- *bytesPerPeriod*: This member states the number of bytes that this controller will allow in a given period.
- *periodMillisecs*: It specifies the window of time in which no more than *bytesPerPeriod* bytes are allowed.

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);
// Limit to 300kb per second.
eprosima::fastrtps::rtps::ThroughputControllerDescriptor␣
↪slowPublisherThroughputController{300000, 1000};
wire_protocol.throughput_controller = slowPublisherThroughputController;
```

(continues on next page)

(continued from previous page)

```
//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);
```

XML

```
<participant profile_name="UDP SERVER" 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>
        </builtin>
        <throughputController>
            <bytesPerPeriod>300000</bytesPerPeriod>
            <periodMillisecs>1000</periodMillisecs>
        </throughputController>
        <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>
    </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*

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);
```

XML

```
<publisher 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>
</publisher>
```

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.

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.

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.

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*.

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.

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.

SampleRejectedStatus

This status changes every time an incoming data sample is rejected by the DataReader. The reason for the rejection is stored as a *SampleRejectedStatusKind*. 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`.

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.

SampleRejectedStatusKind

There are four possible values (see `SampleRejectedStatusKind`):

- `NOT_REJECTED`: It means there have been no rejections so far on this DataReader. The rejection reason will have this value only while the total count of rejections is zero.
- `REJECTED_BY_INSTANCES_LIMIT`: The sample was rejected because the *max_instances* limit was reached.
- `REJECTED_BY_SAMPLES_LIMIT`: The sample was rejected because the *max_samples* limit was reached.
- `REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT`: The sample was rejected because the *max_samples_per_instance* limit was reached.

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*.

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.

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*.

6.16.2 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()`	No

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);
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;
}

// 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);
```

(continues on next page)

(continued from previous page)

```
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
// (...)

// 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)
```

(continues on next page)

(continued from previous page)

```
{
    // 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 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.
- *`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).

```
class CustomDomainParticipantListener : public DomainParticipantListener
{

public:

    CustomDomainParticipantListener()
        : DomainParticipantListener()
```

(continues on next page)

(continued from previous page)

```
    {
    }

    virtual ~CustomDomainParticipantListener()
    {
    }

    virtual void on_participant_discovery(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ParticipantDiscoveryInfo&& info)
    {
        if (info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOVERED_PARTICIPANT)
        {
            std::cout << "New participant discovered" << std::endl;
        }
        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
    virtual void onParticipantAuthentication(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ParticipantAuthenticationInfo&& info)
    {
        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

    virtual void on_subscriber_discovery(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info)
    {
        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;
```

(continues on next page)

(continued from previous page)

```
        }
    }

    virtual void on_publisher_discovery(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info)
    {
        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;
        }
    }

    virtual 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)
    {
        (void)participant, (void)request_sample_id, (void)topic, (void)identifier,␣
↪(void)object, (void)dyn_type;
        std::cout << "New data type discovered" << std::endl;

    }

    virtual void on_type_dependencies_reply(
            DomainParticipant* participant,
            const eprosima::fastrtps::rtps::SampleIdentity& request_sample_id,
            const eprosima::fastrtps::types::TypeIdentifierWithSizeSeq& dependencies)
    {
        (void)participant, (void)request_sample_id, (void)dependencies;
        std::cout << "Answer to a request for type dependencies was received" <<␣
↪std::endl;
    }

    virtual 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)
    {
        (void)participant, (void)topic_name, (void)type_name, (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
    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;
}
```

(continues on next page)

(continued from previous page)

```
// 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.
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 =
        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 domain 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.

```
// 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 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">
    <publisher profile_name="pub_11">
        <topic>
            <name>TopicName</name>
            <dataType>TopicDataTypeName</dataType>
        </topic>
        <qos>
            <partition>
                <names>
                    <name>Partition_1</name>
                    <name>Partition_2</name>
                </names>
            </partition>
        </qos>
    </publisher>

    <publisher profile_name="pub_12">
        <topic>
            <name>TopicName</name>
            <dataType>TopicDataTypeName</dataType>
        </topic>
        <qos>
            <partition>
                <names>
                    <name>*</name>
                </names>
            </partition>
        </qos>
    </publisher>

    <publisher profile_name="pub_21">
        <topic>
            <name>TopicName</name>
            <dataType>TopicDataTypeName</dataType>
```

6.16.3 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
PublisherQos qos = publisher->get_qos();
```

(continues on next page)

(continued from previous page)

```
// 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;
}

// Get the current QoS or create a new one from scratch
PublisherQos qos_type2;

// Modify QoS attributes
```

(continues on next page)

(continued from previous page)

```
// (...)

// 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
PublisherQos custom_qos;

// Modify QoS attributes
// (...)

// Create a publisher with a custom PublisherQos
Publisher* publisher = participant->create_publisher(custom_qos);
```

(continues on next page)

(continued from previous page)

```
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.
// 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()` 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()` 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
    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.

```
// 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 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()*	Yes
DeadlineQosPolicy	*deadline()*	Yes
LatencyBudgetQosPolicy	*latency_budget()*	Yes
LivelinessQosPolicy	*liveliness()*	No
ReliabilityQosPolicy	*reliability()*	No (*)
DestinationOrderQosPolicy	*destination_order()*	No
HistoryQosPolicy	*history()*	Yes
ResourceLimitsQosPolicy	*resource_limits()*	Yes
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()*	Yes
DataRepresentationQosPolicy	*representation()*	Yes
PropertyPolicyQos	*properties()*	Yes
RTPSReliableWriterQos	*reliable_writer_qos()*	Yes
RTPSEndpointQos	*endpoint()*	Yes
WriterResourceLimitsQos	*writer_resource_limits()*	Yes
ThroughputControllerDescriptor	*throughput_controller()*	Yes
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*

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.

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;
}
```

(continues on next page)

(continued from previous page)

```
// 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();

// 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;
}
```

(continues on next page)

(continued from previous page)

```
// 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())
        != 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
```

(continues on next page)

(continued from previous page)

```
    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.

Warning: Currently *on_offered_deadline_missed* is not implemented (it will never be called), and will be implemented on a future release of Fast DDS.

- *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*.

```
class CustomDataWriterListener : public DataWriterListener
{

public:

    CustomDataWriterListener()
        : DataWriterListener()
    {
    }

    virtual ~CustomDataWriterListener()
    {
    }

    virtual void on_publication_matched(
            DataWriter* writer,
            const PublicationMatchedStatus& info)
```

(continues on next page)

(continued from previous page)

```
    {
        (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;
        }
    }

    virtual void on_offered_deadline_missed(
            DataWriter* writer,
            const OfferedDeadlineMissedStatus& status)
    {
        (void)writer, (void)status;
        std::cout << "Some data could not be delivered on time" << std::endl;
    }

    virtual void on_offered_incompatible_qos(
            DataWriter* /*writer*/,
            const OfferedIncompatibleQosStatus& status)
    {
        std::cout << "Found a remote Topic with incompatible QoS (QoS ID: " << status.
↪last_policy_id <<
            ")" << std::endl;
    }

    virtual void on_liveliness_lost(
            DataWriter* writer,
            const LivelinessLostStatus& status)
    {
        (void)writer, (void)status;
        std::cout << "Liveliness lost. Matched Subscribers will consider us offline" <
↪< 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
// (...)

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)
{
    // 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;
}
```

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)
{
    // Error
    return;
}

// Use the DataWriter to communicate
// (...)

// Delete the DataWriter
if (publisher->delete_datawriter(data_writer) != ReturnCode_t::RETCODE_OK)
{
    // Error
```

(continues on next page)

(continued from previous page)

```
    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)
{
    // 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)
{
```

(continues on next page)

(continued from previous page)

```
    // 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;
    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
```

(continues on next page)

(continued from previous page)

```
        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);
```

6.16.4 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
// (...)
```

(continues on next page)

(continued from previous page)

```
// 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)
{
    // 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 de-

fault 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)
{
    // 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.

```
class CustomSubscriberListener : public SubscriberListener
{

public:

    CustomSubscriberListener()
        : SubscriberListener()
    {
    }

    virtual ~CustomSubscriberListener()
    {
    }

    virtual void on_data_on_readers(
            Subscriber* sub)
    {
        (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 =
        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()* 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() 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.

```
// 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
Subscriber* subscriber =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}

// Use the Subscriber to communicate
// (...)

// 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
RTPSEndpoinTimeBasedFilterQosPolicytQos	*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

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.

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
    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
// (...)

// 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
```

(continues on next page)

(continued from previous page)

```
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)
{
    // 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.

Warning: Currently *on_requested_deadline_missed()* is not implemented (it will never be called), and will be implemented on a future release of Fast DDS.

- *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.

Warning: Currently *on_sample_rejected()* is not implemented (it will never be called), and will be implemented on a future release of Fast DDS.

- *on_sample_lost()*: A data sample was lost and will never be received.

Warning: Currently *on_sample_lost()* is not implemented (it will never be called), and will be implemented on a future release of Fast DDS.

```
class CustomDataReaderListener : public DataReaderListener
{

public:

    CustomDataReaderListener()
        : DataReaderListener()
    {
    }

    virtual ~CustomDataReaderListener()
    {
    }
```

(continues on next page)

(continued from previous page)

```
    virtual void on_data_available(
            DataReader* reader)
    {
        (void)reader;
        std::cout << "Received new data message" << std::endl;
    }

    virtual void on_subscription_matched(
            DataReader* reader,
            const SubscriptionMatchedStatus& info)
    {
        (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;
        }
    }

    virtual void on_requested_deadline_missed(
            DataReader* reader,
            const eprosima::fastrtps::RequestedDeadlineMissedStatus& info)
    {
        (void)reader, (void)info;
        std::cout << "Some data was not received on time" << std::endl;
    }

    virtual void on_liveliness_changed(
            DataReader* reader,
            const eprosima::fastrtps::LivelinessChangedStatus& info)
    {
        (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;
        }
    }

    virtual void on_sample_rejected(
            DataReader* reader,
            const eprosima::fastrtps::SampleRejectedStatus& info)
    {
        (void)reader, (void)info;
        std::cout << "A received data sample was rejected" << std::endl;
    }

    virtual void on_requested_incompatible_qos(
            DataReader* /*reader*/,
            const RequestedIncompatibleQosStatus& info)
```

(continues on next page)

(continued from previous page)

```
    {
        std::cout << "Found a remote Topic with incompatible QoS (QoS ID: " << info.
↪last_policy_id <<
            ")" << std::endl;
    }

    virtual void on_sample_lost(
            DataReader* reader,
            const SampleLostStatus& info)
    {
        (void)reader, (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);
```

(continues on next page)

(continued from previous page)

```
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;
}
```

(continues on next page)

(continued from previous page)

```
// 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;
}
```

Deleting a DataReader

A DataReader can be deleted with the `delete_datareader()` member function on the *Subscriber* instance where the DataReader was created.

```
// 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 DataReader
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.

Note: Currently the `sample_state` is not implemented, and its value is always set to **NOT_READ**. It will be implemented on a future release of *Fast DDS*.

---

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.

Note: Currently the `view_state` is not implemented, and its value is always set to **NOT_NEW**. It will be implemented on a future release of *Fast DDS*.

---

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*.

Note: Currently the *instance_state* is partially implemented, and the value **NOT_ALIVE_NO_WRITERS** will never be set. It will be fully implemented on a future release of *Fast DDS*.

---

disposed_generation_count

disposed_generation_count indicates the number of times the instance had become alive after it was disposed.

Note: Currently the *disposed_generation_count* is not implemented, and its value is always set to `0`. It will be implemented on a future release of *Fast DDS*.

---

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`.

Note: Currently the *no_writers_generation_count* is not implemented, and its value is always set to `1`. It will be implemented on a future release of *Fast DDS*.

---

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);

    // 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.

```
    // 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 for which valid_data is true should be accessed
            if (info_seq[n].valid_data)
            {
                // 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()
    {
    }

    virtual void on_data_available(
            DataReader* reader)
    {
        // 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

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 with the `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 threat 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 (data_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 "
                          << topic->get_name()
                          << 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;
    }
}
```

6.16.5 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

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.

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)
{
    // 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
// (...)
```

(continues on next page)

(continued from previous page)

```
// Set as the new default TopicQos
if (participant->set_default_topic_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // 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()`*.

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 specialization implemented is *Topic*.

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()
    {
    }

    virtual void on_inconsistent_topic(
            Topic* topic,
            InconsistentTopicStatus status)
    {
        (void)topic, (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());

// 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);
```

(continues on next page)

(continued from previous page)

```
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.
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.
```

(continues on next page)

(continued from previous page)

```
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);
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 =
```

(continues on next page)

(continued from previous page)

```
        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;
}
```

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 *fastrtpsgen* on Linux or *fastrtpsgen.bat* on Windows. The IDL file containing the data type definition is given with the `<IDLfile>` argument.

Linux
`fastrtpsgen [<options>] <IDLfile> [<IDLfile> ...]`
Windows
`fastrtpsgen.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.

6.17 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.

6.17.1 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

6.17.2 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/C++/RTPSTest_as_socket and examples/C++/RTPSTest_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);
    }

};
```

6.17.3 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.

6.17.4 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.

6.17.5 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;
    }
```

(continues on next page)

(continued from previous page)

```
    bool get_payload(
            SerializedPayload_t& data,
            IPayloadPool*& /* data_owner */,
            CacheChange_t& cache_change) override
    {
        // Reserve new memory for the payload buffer
        octet* payload = new octet[data.length];

        // Copy the data
        memcpy(payload, data.data, data.length);

        // 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;

        // Tell the CacheChange who needs to release its payload
        cache_change.payload_owner(this);

        return true;
    }

    bool release_payload(
            CacheChange_t& cache_change) override
    {
        // Ensure precondition
        assert(this == cache_change.payload_owner());

        // 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;
```

(continues on next page)

(continued from previous page)

```
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.
// 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);
```

6.18 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.

6.18.1 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.

6.18.2 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, act as a hub for discovery meta traffic.
- **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.

6.18.3 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 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 the 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

```
<participant profile_name="participant_discovery_protocol">
    <rtps>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SIMPLE</discoveryProtocol>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

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

```
<participant profile_name="participant_discovery_ignore_flags">
    <rtps>
        <builtin>
            <discovery_config>
                <ignoreParticipantFlags>FILTER_DIFFERENT_PROCESS | FILTER_SAME_
↪PROCESS</ignoreParticipantFlags>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

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

```
<participant profile_name="participant_discovery_lease_duration">
    <rtps>
        <builtin>
            <discovery_config>
                <leaseDuration>
                    <sec>10</sec>
                    <nanosec>20</nanosec>
                </leaseDuration>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

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

```
<participant profile_name="participant_discovery_lease_announcement">
    <rtps>
        <builtin>
            <discovery_config>
                <leaseAnnouncement>
                    <sec>1</sec>
                    <nanosec>2</nanosec>
                </leaseAnnouncement>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

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>
                        <sec>0</sec>
                        <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 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 with 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.

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>
```

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>`	It 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`
`<partitionName>`	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 the *ReliabilityQosPolicy* section.	*BEST_EFFORT_RELIABILITY_QOS* *RELIABLE_RELIABILITY_QOS*	*BEST_EFFORT_RELIABILITY_QOS*
`<durabilityQos>`	See the *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*.		

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`

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"/>
        </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"/>
        </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

```
<publisher profile_name="publisher_xml_conf_static_discovery">
    <userDefinedID>3</userDefinedID>
</publisher>

<subscriber profile_name="subscriber_xml_conf_static_discovery">
    <userDefinedID>5</userDefinedID>
</subscriber>
```

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.

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 on 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 a beforehand 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 all the discovery information known by the *server*, in opposition to *clients*, which only receive the information they need.
- *Servers* do not require any beforehand 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*.

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">
                            <metatrafficUnicastLocatorList>
                                <locator/>
                            </metatrafficUnicastLocatorList>
                        </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*.

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>
```

Full example

The following constitutes a full example on how to configure *server* and *client* both programmatically and using XML.

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.
                        This list may contain one or more <RemoteServer> tags
                    -->
                    <discoveryServersList>
                        <!--
                            Set remote server configuration:
```

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">
                            <metatrafficUnicastLocatorList>
                                <!-- Set SERVER's listening locator for PDP -->
                                <locator>
                                    <udpv4>
```

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 */
    virtual void on_participant_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::ParticipantDiscoveryInfo&& info)
    {
        (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:
                /* 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 */
    virtual void on_subscriber_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info)
    {
        (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";
```

(continues on next page)

(continued from previous page)

```
                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 */
    virtual void on_publisher_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info)
    {
        (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 */
                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 */
    virtual 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)
    {
        (void)participant, (void)request_sample_id, (void)topic, (void)identifier,␣
↪(void)object, (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);
```

6.19 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*.

6.19.1 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*

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 `transport_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-api`	`0`	None. Reserved value for internal use.
`LOCATOR_KIND_UDPv4-api`	`1`	*UDP Transport* over IPv4.
`LOCATOR_KIND_UDPv6-api`	`2`	*UDP Transport* over IPv6.
`LOCATOR_KIND_TCPv4-api`	`4`	*TCP Transport* over IPv4.
`LOCATOR_KIND_TCPv6-api`	`8`	*TCP Transport* over IPv6.
`LOCATOR_KIND_SHM-api`	`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 stored in the two most significant bytes of the member `port`.

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`.

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");
```

6.19.2 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 `TransportDescriptor`. However, all their `TransportDescriptor` 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	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 *UDPTransportDescriptor*.

Note: The *kind* value for a UDPv4TransportDescriptor is given by the value `eprosima::fastrtps::rtps::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 `eprosima::fastrtps::rtps::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>
    </participant>
</profiles>
```

6.19.3 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*. 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 `TransportDescriptor`. However, they share many of their features, and most of the `TransportDescriptor` 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	List of allowed interfaces. See *Interface Whitelist*
`TTL`	`uint8_t`	`1`	Time to live, in number of hops.
`listening_ports`	`vector<uint16_t>`	empty	List of ports to listen as *server*.
`keep_alive_frequency_ms`	`uint32_t`	5000	Frequency of RTCP keep alive requests (in ms).
`keep_alive_timeout_ms`	`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]`	empty	Configuration for TLS. See *WAN or Internet Communication over TCPv4*.

Note: The *kind* value for a TCPv4TransportDescriptor is given by the value

`eprosima::fastrtps::rtps::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 `eprosima::fastrtps::rtps::LOCATOR_KIND_TCPv6`

Enabling TCP Transport

To enable TCP transport in a DomainParticipant, 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.

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++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<TCPv4TransportDescriptor>();
tcp_transport->sendBufferSize = 9216;
tcp_transport->receiveBufferSize = 9216;
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;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tcp_transport</transport_id>
            <type>TCPv4</type>
            <sendBufferSize>9216</sendBufferSize>
            <receiveBufferSize>9216</receiveBufferSize>
            <listening_ports>
                <port>5100</port>
            </listening_ports>
            <wan_addr>80.80.99.45</wan_addr>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TCPParticipant">
        <rtps>
            <userTransports>
                <transport_id>tcp_transport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
        </rtps>
    </participant>
</profiles>
```

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++

```
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<TCPv4TransportDescriptor>();
qos.transport().user_transports.push_back(tcp_transport);

// Set initial peers.
Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
IPLocator::setIPv4(initial_peer_locator, "80.80.99.45");
initial_peer_locator.port = 5100;

qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer_locator);

// Avoid using the default transport
qos.transport().use_builtin_transports = false;
```

XML

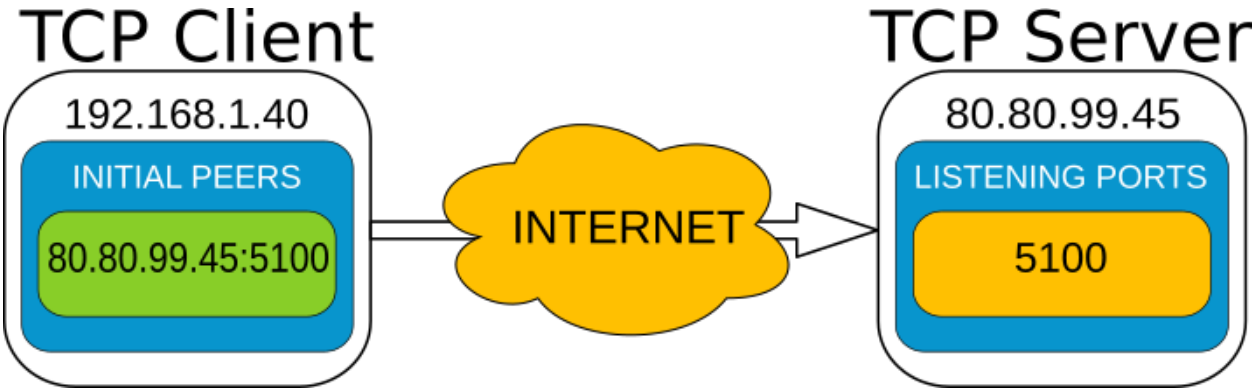
```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tcp2_transport</transport_id>
            <type>TCPv4</type>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TCP2Participant">
        <rtps>
            <userTransports>
                <transport_id>tcp2_transport</transport_id>
            </userTransports>
             <useBuiltinTransports>false</useBuiltinTransports>
            <builtin>
                <initialPeersList>
                    <locator>
                        <tcpv4>
                            <address>80.80.99.45</address>
                            <physical_port>5100</physical_port>
                        </tcpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

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_peer` list.

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++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<TCPv4TransportDescriptor>();
tcp_transport->sendBufferSize = 9216;
tcp_transport->receiveBufferSize = 9216;
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;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>tcp_transport</transport_id>
            <type>TCPv4</type>
            <sendBufferSize>9216</sendBufferSize>
            <receiveBufferSize>9216</receiveBufferSize>
            <listening_ports>
                <port>5100</port>
            </listening_ports>
            <wan_addr>80.80.99.45</wan_addr>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TCPParticipant">
        <rtps>
            <userTransports>
                <transport_id>tcp_transport</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
        </rtps>
    </participant>
</profiles>
```

On the client side, the DomainParticipant must be configured with the **public** IP address and `listening_port` of the *TCP server* as `initial_peer`.

C++

```
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<TCPv4TransportDescriptor>();
qos.transport().user_transports.push_back(tcp_transport);

// Set initial peers.
Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
IPLocator::setIPv4(initial_peer_locator, "80.80.99.45");
initial_peer_locator.port = 5100;

qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer_locator);

// 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>tcp2_transport</transport_id>
            <type>TCPv4</type>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TCP2Participant">
        <rtps>
            <userTransports>
                <transport_id>tcp2_transport</transport_id>
            </userTransports>
             <useBuiltinTransports>false</useBuiltinTransports>
            <builtin>
                <initialPeersList>
                    <locator>
                        <tcpv4>
                            <address>80.80.99.45</address>
                            <physical_port>5100</physical_port>
                        </tcpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

HelloWorldExampleTCP

A TCP version of helloworld example can be found in the `examples/C++/DDS/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*.

6.19.4 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.

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 that will help understanding how the Shared Memory Transport works in order to deliver the data messages to the appropriate *DomainParticipant*. The purpose is not to be a 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.

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()`	Timeout for the health check of ports (in milliseconds).
`rtps_dump_file_`	`string`	Empty	`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 `eprosima::fastrtps::rtps::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>
```

HelloWorldExampleSharedMem

A Shared Memory version of helloworld example can be found in the `examples/C++/DDS/HelloWorldExampleSharedMem` folder. It shows a publisher and a subscriber that communicate through Shared Memory.

6.19.5 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.

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

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`.

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.

Warning: Currently the configuration of shared memory directory is not supported. As a result, any directory set by the user will be discarded, and the default directory configured for the current system is used.

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. 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.

6.19.6 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*. 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. 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

C++ - Option 1: Manual setting of the `unsigned char` in 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++ - Option 2: Using the >> operator and the `std::istringstream` type.

```
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>
```

6.19.7 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 TransportDescriptor. 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);
tls_transport->set_WAN_address("80.80.99.45");

// 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>
            <wan_addr>80.80.99.45</wan_addr>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TLSServerParticipant">
        <rtps>
            <userTransports>
                <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, "80.80.99.45");
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);

// 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>
                <options>
                    <option>DEFAULT_WORKAROUNDS</option>
                    <option>SINGLE_DH_USE</option>
                    <option>NO_SSLV2</option>
                </options>
            </tls>
        </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="TLSClientParticipant">
        <rtps>
            <userTransports>
                <transport_id>tls_transport_client</transport_id>
            </userTransports>
            <builtin>
                <initialPeersList>
                    <locator>
                        <tcpv4>
                            <address>80.80.99.45</address>
```

The following table describes the data members that are configurable on `TLSConfig`.

Member	Data type	Default	Description
`password`	`string`	empty	Password of the `private_key_file` or `rsa_private_key_file`.
`private_key_file`	`string`	empty	Path to the private key certificate file.
`rsa_private_key_file`	`string`	empty	Path to the private key RSA certificate file.
`cert_chain_file`	`string`	empty	Path to the public certificate chain file.
`tmp_dh_file`	`string`	empty	Path to the Diffie-Hellman parameters file.
`verify_file`	`string`	empty	Path to the CA (Certification- Authority) file.
`verify_mode`	`TLSVerifyMode`	empty	Establishes the verification mode mask. See *TLS Verification Mode*
`options`	`TLSOptions`	empty	Establishes the SSL Context options mask. See *TLS Options*
`verify_paths`	`vector<string>`	empty	Paths where the system will look for verification files.
`verify_depth`	`int32_t`	empty	Maximum allowed depth for verifying intermediate certificates.
`default_verify_path`	`bool`	empty	Look for verification files on the default paths.
`handshake_role`	`TLSHandShakeRole`	`DEFAULT`	Role that the transport will take on handshaking. See *TLS Handshake Role*

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 TransportDescriptor using the `add_verify_mode()` member function.

Value	Description
`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.

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
`DEFAULT_WORKAROUNDS`	Implement various bug workarounds. See Boost.Asio context
`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 when using *Diffie-Hellman* parameters.

TLS Handshake Role

The role can take the following values:

Value	Description
`DEFAULT`	Configured as client if connector, and as server if acceptor
`CLIENT`	Configured as client.
`SERVER`	Configured as server.

6.19.8 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 using the field `wire_protocol().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 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 using the field `wire_protocol().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 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>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

User-traffic Multicast Locators

Users can set their own user-traffic multicast locators using the field `wire_protocol().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 using the field `wire_protocol().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 *user-traffic 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 1: 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 corresponding field on the *DomainParticipantQos*.

Table 2: 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`

6.19.9 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 `TransportDescriptor` 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);
```

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>
            <userTransports>
                <transport_id>CustomTcpTransport</transport_id>
            </userTransports>
        </rtps>
    </participant>
</profiles>
```

Warning: The interface whitelist feature applies to network interfaces. Therefore, it is only available on *TCP Transport* and *UDP Transport*.

6.19.10 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/>
                </metatrafficUnicastLocatorList>
                <initialPeersList>
                    <locator>
                        <udpv4>
                            <address>192.168.0.1</address>
                        </udpv4>
                    </locator>
                </initialPeersList>
            </builtin>
        </rtps>
    </participant>
</profiles>
```

6.20 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.

6.20.1 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*):

6.20.2 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 3: **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.

6.20.3 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;
 *     };
 */

// 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;
```

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.

6.21 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.

6.21.1 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.
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.

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>
    </rtps>
</participant>
```

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

# For the CA policy
[ policy_match ]
countryName = match
```

(continues on next page)

(continued from previous page)

```
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 \
```

(continues on next page)

(continued from previous page)

```
  -revoke partcert.pem

openssl ca -gencrl \
  -config maincaconf.cnf \
  -cert maincacert.pem \
  -keyfile maincakey.pem \
  -out crl.pem
```

6.21.2 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.auth.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	*Domains*	`<domains>`	`false`
			`true`
	Allow Unauthenticated Participants	`<allow_unauthenticated_participants>`	`false`
			`true`
	Enable Join Access Control	`<enable_join_access_control>`	`SIGN`
			`ENCRYPT`
			`NONE`
	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	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.

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>`	`true`
		`false`
Data protection Kind	`<data_protection_kind>`	`true`
		`false`

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.

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.

- false: the RTPS submessages shall not be encrypted.
- true: the RTPS submessages 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.

- false: the data payload shall not be encrypted.
- true: the data payload 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>
    <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" />
```

(continues on next page)

(continued from previous page)

```
            <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" />
            <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="PublisherPermissions">
            <subject_name>emailAddress=mainpub@eprosima.com, CN=Main Publisher, OU=eProsima, 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>
            <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>
        <grant name="SubscriberPermissions">
            <subject_name> emailAddress=mainsub@eprosima.com, CN=Main Subscriber, OU=eProsima, 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>
            <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>
```

(continues on next page)

(continued from previous page)

```
    </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" />
            <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">
```

(continues on next page)

(continued from previous page)

```
        <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" />
            <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

# Permissions document: permissions.xml
openssl smime -sign -in permissions.xml -text -out permissions.smime -signer␣
↪maincacert.pem -inkey maincakey.pem
```

6.21.3 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. If the *Access control plugin: DDS:Access:Permissions* plugin will not be used, you can configure the DDS:Crypto:AES-GCM-GMAC plugin manually with the properties outlined in the following table.

Property name	Description	Property Value
rtps.participant.rtps_protection_kind	Encrypt whole RTPS messages	`ENCRYPT`
rtps.endpoint.submessage_protection_kind	Encrypt RTPS submessages of a particular entity	`ENCRYPT`
rtps.endpoint.payload_protection_kind	Encrypt payload of a particular Writer	`ENCRYPT`

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");

// Only if DDS:Access:Permissions plugin is not enabled
// Configure DDS:Crypto:AES-GCM-GMAC plugin
pqos.properties().properties().emplace_back(
    "rtps.participant.rtps_protection_kind",
    "ENCRYPT");
```

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>
                <!-- Only if DDS:Access:Permissions plugin is not enabled -->
                <!-- Configure DDS:Crypto:AES-GCM-GMAC plugin -->
                <property>
                    <name>rtps.participant.rtps_protection_kind</name>
                    <value>ENCRYPT</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Next example shows how to configure DataWriters to encrypt their RTPS submessages and the RTPS message payload, i.e. the user data. This is done by setting the DDS:Crypto:AES-GCM-GMAC properties (*properties()*) corresponding to the DataWriters in the *DataWriterQos*.

C++

```
DataWriterQos wqos;

// Only if DDS:Access:Permissions plugin is not enabled
// Configure DDS:Crypto:AES-GCM-GMAC plugin
wqos.properties().properties().emplace_back(
    "rtps.endpoint.submessage_protection_kind",
    "ENCRYPT");
wqos.properties().properties().emplace_back(
    "rtps.endpoint.payload_protection_kind",
    "ENCRYPT");
```

XML

```
<publisher profile_name="secure_datawriter_conf_crypto_plugin_xml_profile">
    <propertiesPolicy>
        <properties>
            <!-- Only if DDS:Access:Permissions plugin is not enabled -->
            <!-- Configure DDS:Crypto:AES-GCM-GMAC plugin -->
            <property>
                <name>rtps.endpoint.submessage_protection_kind</name>
                <value>ENCRYPT</value>
            </property>
            <property>
                <name>rtps.endpoint.payload_protection_kind</name>
                <value>ENCRYPT</value>
            </property>
        </properties>
    </propertiesPolicy>
</publisher>
```

The last example shows how to configure DataReader to encrypt their RTPS submessages. This is done by setting the DDS:Crypto:AES-GCM-GMAC properties (*properties()*) corresponding to the DataReaders in the *DataReaderQos*.

C++

```
DataWriterQos rqos;

// Only if DDS:Access:Permissions plugin is not enabled
// Configure DDS:Crypto:AES-GCM-GMAC plugin
rqos.properties().properties().emplace_back(
    "rtps.endpoint.submessage_protection_kind",
    "ENCRYPT");
```

XML

```
<subscriber profile_name="secure_datareader_conf_crypto_plugin_xml_profile">
    <propertiesPolicy>
        <properties>
            <!-- Only if DDS:Access:Permissions plugin is not enabled -->
            <!-- Configure DDS:Crypto:AES-GCM-GMAC plugin -->
            <property>
                <name>rtps.endpoint.submessage_protection_kind</name>
                <value>ENCRYPT</value>
            </property>
        </properties>
    </propertiesPolicy>
</subscriber>
```

6.21.4 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 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.

6.22.1 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*.

6.22.2 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.

6.22.3 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()`.

6.22.4 Logging Messages

The logging of messages is handled by three dedicated macros, one for each available verbosity level (see *Verbosity Level*):

- `logInfo`: Logs messages with `Log::Kind::Info` verbosity.
- `logWarning`: Logs messages with `Log::Kind::Warning` verbosity.
- `logError`: 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*).

```
logInfo(DOCUMENTATION_CATEGORY, "This is an info message");
logWarning(DOCUMENTATION_CATEGORY, "This is an warning message");
logError(DOCUMENTATION_CATEGORY, "This is an error message");
```

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*.

6.22.5 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

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 `logError` 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.

6.22.6 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
logError(CATEGORY_1, "First log entry");
// Would be consumed
logError(CATEGORY_2, "Second log entry");
// Would NOT be consumed
logError(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
logError(CATEGORY, "First log entry");

// Set filter using regular expression so filename must match "other"
Log::SetFilenameFilter(std::regex("other"));
// Would NOT be consumed
logError(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
logError(CATEGORY, "First log entry");
// Would NOT be consumed
logError(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`.

6.22.7 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));
```

6.22.8 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.

- `logInfo` 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`.
- `logWarning` is fully disabled by either:
  - Setting CMake option `LOG_NO_WARNING` to `ON`.
  - Defining macro `HAVE_LOG_NO_WARNING` to `1`.
- `logError` 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.
- `logInfo` has a special behaviour to simplify working with Multi-Config capability IDEs. If `LOG_NO_INFO` is `OFF` or `HAVE_LOG_NO_INFO` is `0` the logging is enabled only for `Debug` configuration. By setting `FASTDDS_ENFORCE_LOG_INFO` to `ON` the logging will always be enabled.

Warning: `INTERNAL_DEBUG` can be automatically set to `ON` if CMake option `EPROSIMA_BUILD` is set to `ON`.

6.23 Statistics Module

The *Fast DDS Statistics module* is an extension to Fast DDS that enables the recollection of data concerning the DDS communication. The collected data is published using DDS over some specific 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 performance. 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 those topics that may be of interest for the user's application. Therefore, the standard *DDS Layer* has been extended.

The following section explains this DDS extended API.

6.23.1 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. Finally, the recommended QoS for enabling the DataWriters and creating the user's DataReaders that subscribe to the Statistics topics are described.

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.

Warning: This statistics topic is not yet implemented.

NETWORK_LATENCY_TOPIC

The `_fastdds_statistics_network_latency` statistics topic collects data related with the network latency 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`.

Warning: This statistics topic is not yet implemented.

PUBLICATION_THROUGHPUT_TOPIC

The `_fastdds_statistics_publication_throughput` statistics topic collects the amount of data that is being sent by each DataWriter. This measurement provides information about the publication's throughput.

Warning: This statistics topic is not yet implemented.

SUBSCRIPTION_THROUGHPUT_TOPIC

The `_fastdds_statistics_subscription_throughput` statistics topic collects the amount of data that is being received by each DataReader. This measurement provides information about the subscription's throughput.

Warning: This statistics topic is not yet implemented.

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.

Warning: This statistics topic is not yet implemented.

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.

Warning: This statistics topic is not yet implemented.

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.

Warning: This statistics topic is not yet implemented.

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).

Warning: This statistics topic is not yet implemented.

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).

Warning: This statistics topic is not yet implemented.

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*).

PHYSICAL_DATA_TOPIC

The `_fastdds_statistics_physical_data` statistics topic reports the host, user and process where the *Fast DDS Statistics module* is running.

Warning: This statistics topic is not yet implemented.

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

Statistics DataWriters are enabled using the method `enable_statistics_datawriter()`. This 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).

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.

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);

// 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;
}
```

(continues on next page)

(continued from previous page)

```
// 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
```

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*.

Warning: Currently, the following *statistics topics* have not been implemented yet. They will be available in future releases:

- *HISTORY_LATENCY_TOPIC*
- *NETWORK_LATENCY_TOPIC*
- *PUBLICATION_THROUGHPUT_TOPIC*
- *SUBSCRIPTION_THROUGHPUT_TOPIC*
- *RTPS_LOST_TOPIC*
- *RESENT_DATAS_TOPIC*
- *SAMPLE_DATAS_TOPIC*

- *PDP_PACKETS_TOPIC*
- *EDP_PACKETS_TOPIC*
- *PHYSICAL_DATA_TOPIC*

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*
HistoryQosPolicyKind	*KEEP_LAST_HISTORY_QOS*
history_depth	100
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

6.24 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*).

An XML profile is defined by a unique name that is used to reference the XML profile during the creation of an *Entity*, the *Trasport* 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.

6.24.1 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*. The available profile types are:

- *DomainParticipant profiles*,
- *DataWriter profiles*,
- *DataReader profiles*,
- *Transport descriptors*,
- *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>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <participant profile_name="participant_profile">
            <!-- ... -->
        </participant>

        <data_writer profile_name="datawriter_profile">
            <!-- ... -->
        </data_writer>

        <data_reader profile_name="datareader_profile">
            <!-- ... -->
        </data_reader>

        <transport_descriptors>
            <!-- ... -->
        </transport_descriptors>

        <log>
            <!-- ... -->
        </log>

        <types>
            <!-- ... -->
        </types>
    </profiles>
</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. Moreover, *create_participant_with_profile()*, *create_publisher()*, and *create_subscriber()* 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"))
{
    DomainParticipant* participant =
            DomainParticipantFactory::get_instance()->create_participant_with_profile(
        0, "participant_xml_profile");
    Publisher* publisher = participant->create_publisher_with_profile("publisher_xml_
↪profile");
    Subscriber* subscriber = participant->create_subscriber_with_profile("subscriber_
↪xml_profile");
}
```

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>`, `<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>`, 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
<?xml version="1.0" encoding="UTF-8" ?>
<types>
    <type>
        <!-- Type definition -->
    </type>

    <type>
        <!-- Type definition -->
        <!-- Type definition -->
    </type>
</types>
```

Rooted

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
    <types>
        <type>
            <!-- Type definition -->
        </type>

        <type>
            <!-- Type definition -->
            <!-- Type definition -->
        </type>
    </types>
</dds>
```

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(
```

(continues on next page)

(continued from previous page)

```
        0, participant_qos);
}
```

6.24.2 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: `<domain_id>` and `<rtps>`. All the DomainParticipant configuration options belong to the `<rtps>` element, except for the DDS `DomainId` which is defined by the `<domain_id>` element. Below a list with the configuration XML elements is presented:

Name	Description	Values	Default
`<domainId>`	DomainId to be used by the 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>	
<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
<throughputController>	Limits middleware's bandwidth usage. See the *Throughput Configuration* section.	*Throughput Configuration*	
<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* transport instead of its <userTransports>.	bool	true
<propertiesPolicy>	Additional configuration properties. It expects a *PropertiesPolicyType*.	*PropertiesPolicyType*	
<allocation>	Configuration regarding allocation behavior. It expects a *DomainParticipantAllocationType*.	*DomainParticipantAllocationType*	

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="domainparticipant_profile_name">
        <domainId>80</domainId>

        <rtps>
            <name>DomainParticipant Name</name>

            <defaultUnicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <port>7400</port>
                        <address>192.168.1.41</address>
                    </udpv4>
                </locator>
            </defaultUnicastLocatorList>
```

(continues on next page)

(continued from previous page)

```
            <defaultMulticastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4>
                        <port>7400</port>
                        <address>192.168.2.41</address>
                    </udpv4>
                </locator>
            </defaultMulticastLocatorList>

            <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>
            </port>

            <participantID>99</participantID>

            <throughputController>
                <bytesPerPeriod>8192</bytesPerPeriod>
                <periodMillisecs>1000</periodMillisecs>
            </throughputController>

            <userTransports>
                <transport_id>TransportId1</transport_id>
                <transport_id>TransportId2</transport_id>
            </userTransports>

            <useBuiltinTransports>false</useBuiltinTransports>

            <propertiesPolicy>
                <!-- PROPERTIES_POLICY -->
                <properties>
                    <property>
                        <name>Property1Name</name>
                        <value>Property1Value</value>
                        <propagate>false</propagate>
                    </property>
                </properties>
            </propertiesPolicy>

            <allocation>
                <!-- ALLOCATION -->
            </allocation>
```

(continues on next page)

(continued from previous page)

```
    </participant>
```

Note:

- `LOCATOR_LIST` means a *LocatorListType* is expected.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- For `BUILTIN` details, please refer to *Builtin parameters*.
- For `ALLOCATION` details, please refer to *ParticipantAllocationType*.

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*.	`<max_unicast_locators>` `<max_multicast_locators>`	
`<max_unicast_locators>`	Child element of `<remote_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>`	Child element of `<remote_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
`<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. See *SendBuffersAllocationAttributes*.	`uint32_t`	
`<max_user_data>`	Maximum size of the user data submessage. Set to zero for no limit. See *SendBuffersAllocationAttributes*.	`uint32_t`	
`<max_properties>`	Maximum size of the properties submessage. Set to zero for no limit. See *SendBuffersAllocationAttributes*.	`uint32_t`	

Example

```
<allocation>
    <remote_locators>
        <max_unicast_locators>4</max_unicast_locators>
        <max_multicast_locators>1</max_multicast_locators>
    </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>
```

(continues on next page)

(continued from previous page)

```
        <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>
</allocation>
```

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>	Restricts multicast metatraffic to PDP only.	bool	true
<use_WriterLiveliness>	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*	
<DataReaderHistoryMemoryPolicy>	Memory policy for DataReaders. See *HistoryQosPolicyKind*.	*HistoryMemoryPolicy*	*PREALLOCATED*
<DataWriterHistoryMemoryPolicy>	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

Example

```
<builtin>
    <discovery_config>
        <discoveryProtocol>NONE</discoveryProtocol>

        <ignoreParticipantFlags>FILTER_DIFFERENT_HOST</ignoreParticipantFlags>

        <EDP>SIMPLE</EDP>

        <leaseDuration>
            <!-- DURATION -->
            <sec>20</sec>
            <nanosec>0</nanosec>
        </leaseDuration>

        <leaseAnnouncement>
            <!-- DURATION -->
            <sec>3</sec>
            <nanosec>0</nanosec>
        </leaseAnnouncement>

        <initialAnnouncements>
            <!-- INITIAL_ANNOUNCEMENTS -->
        </initialAnnouncements>

        <simpleEDP>
            <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
            <PUBREADER_SUBWRITER>true</PUBREADER_SUBWRITER>
        </simpleEDP>

        <static_edp_xml_config>file://filename.xml</static_edp_xml_config>
    </discovery_config>

    <avoid_builtin_multicast>true</avoid_builtin_multicast>

    <use_WriterLivelinessProtocol>false</use_WriterLivelinessProtocol>

    <metatrafficUnicastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udpv4/>
        </locator>
    </metatrafficUnicastLocatorList>

    <metatrafficMulticastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udpv4/>
        </locator>
    </metatrafficMulticastLocatorList>

    <initialPeersList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udpv4/>
        </locator>
    </initialPeersList>
```

(continues on next page)

(continued from previous page)

```
    <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</readerHistoryMemoryPolicy>

    <readerPayloadSize>512</readerPayloadSize>

    <writerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</writerHistoryMemoryPolicy>

    <writerPayloadSize>512</writerPayloadSize>

    <mutation_tries>55</mutation_tries>
</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*.	*SIMPLE* *CLIENT* *SERVER* *BACKUP* *NONE*	*SIMPLE*
`<ignoreParticipantFlags>`	Restricts the 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 `<staticEndpointXMLFilename>`. 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*.	*Initial Announcements*	
`<staticEndpointXMLFilename>`	The XML filename with the static EDP configuration. Only necessary if the `<EDP>` member is set to `STATIC`. See *STATIC Discovery Settings*.	`string`	

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.
FILTER_DIFFERENT_PROCESS | *FILTER_SAME_PROCESS*	Discovery traffic from DomainParticipant's own host is discarded.

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

6.24.3 DataWriter profiles

The DataWriter profiles allow for configuring *DataWriters* from an XML file. These profiles are defined within the `<data_writer>` or `<publisher>` XML tags. Thus, the following XML code snippets are equivalent.

DataWriter profile - Definition method 1	**DataWriter profile** - Definition method 2

`<data_writer profile_name="my_datawriter_profile">`
`  <topic>`
`  <!-- TOPIC_TYPE -->`
`  </topic>`
`  <qos>`
`  <!-- QOS -->`
`  </qos>`
`  <!-- Other elements -->`
`</data_writer>`
	`<publisher profile_name="my_publisher_profile">`
	`  <topic>`
	`  <!-- TOPIC_TYPE -->`
	`  </topic>`
	`  <qos>`
	`  <!-- QOS -->`
	`  </qos>`
	`  <!-- Other elements -->`
	`</publisher>`

Important: The `<publisher>` and `<data_writer>` XML tags are equivalent. Therefore, XML profiles in which the DataWriters are defined with the `<publisher>` tag are fully compatible with *Fast DDS*.

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.	*Times*	
`<unicastLocatorList>`	List of input unicast locators. It expects a *LocatorListType*.	`<locator>`	
`<multicastLocatorList>`	List of input multicast locators. It expects a *LocatorListType*.	`<locator>`	
`<throughputController>`	Limits the output bandwidth of the DataWriter.	*Throughput Configuration*	
`<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*.	*Allocation Configuration*	

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <data_writer profile_name="datawriter_profile_name">
            <topic>
                <!-- TOPIC_TYPE -->
            </topic>

            <qos>
                <!-- QOS -->
            </qos>

            <times> <!-- writerTimesType -->
                <initialHeartbeatDelay>
                    <sec>0</sec>
                    <nanosec>12</nanosec>
                </initialHeartbeatDelay>

                <heartbeatPeriod>
                    <sec>3</sec>
                    <nanosec>0</nanosec>
                </heartbeatPeriod>

                <nackResponseDelay>
                    <sec>0</sec>
                    <nanosec>5</nanosec>
                </nackResponseDelay>

                <nackSupressionDuration>
```

(continues on next page)

(continued from previous page)

```
                    <sec>0</sec>
                    <nanosec>0</nanosec>
                </nackSupressionDuration>
            </times>

            <unicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4/>
                </locator>
            </unicastLocatorList>

            <multicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4/>
                </locator>
            </multicastLocatorList>

            <throughputController>
                <bytesPerPeriod>8192</bytesPerPeriod>
                <periodMillisecs>1000</periodMillisecs>
            </throughputController>

            <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

            <propertiesPolicy>
                <!-- PROPERTIES_POLICY -->
            </propertiesPolicy>

            <userDefinedID>55</userDefinedID>

            <entityID>66</entityID>

            <matchedSubscribersAllocation>
                <initial>0</initial>
                <maximum>0</maximum>
                <increment>1</increment>
            </matchedSubscribersAllocation>
        </data_writer>
    </profiles>
<dds>
```

Note:

- `LOCATOR_LIST` means a *LocatorListType* 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*.

Times

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

6.24.4 DataReader profiles

The DataReader profiles allow declaring *DataReaders* from an XML file. These profiles are defined within the `<data_reader>` or `<subscriber>` XML tags. Thus, the following XML codes are equivalent.

DataReader profile - Definition method 1

```
<data_reader profile_name="my_datareader_
↪profile">
    <topic>
        <!-- TOPIC_TYPE -->
    </topic>
    <qos>
        <!-- QOS -->
    </qos>
    <!-- Other elements -->
</data_reader>
```

DataReader profile - Definition method 2

```
<subscriber profile_name="my_subscriber_
↪profile">
    <topic>
        <!-- TOPIC_TYPE -->
    </topic>
    <qos>
        <!-- QOS -->
    </qos>
    <!-- Other elements -->
</subscriber>
```

Important: The `<subscriber>` and `<data_reader>` XML tags are equivalent. Therefore, XML profiles in which the DataReaders are defined with the `<subscriber>` tag are fully compatible with *Fast DDS*.

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 this `<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.	*Times*	
`<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*	
`<expectsInlineQos>`	It indicates if QoS is expected inline.	`bool`	`false`
`<historyMemoryPolicy>`	Memory allocation kind for DataReaders's history.	*MemoryManagementPolicy*	*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>`	Sets the limits of the collection of matched DataWriters. See *ParticipantResourceLimitsQos*.	*Allocation Configuration*	

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <data_reader profile_name="sub_profile_name">
            <topic>
                <!-- TOPIC_TYPE -->
            </topic>

            <qos>
                <!-- QOS -->
            </qos>

            <times> <!-- readerTimesType -->
                <initialAcknackDelay>
                    <sec>0</sec>
                    <nanosec>70</nanosec>
                </initialAcknackDelay>

                <heartbeatResponseDelay>
                    <sec>0</sec>
                    <nanosec>5</nanosec>
                </heartbeatResponseDelay>
            </times>

            <unicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4/>
                </locator>
            </unicastLocatorList>
```

(continues on next page)

(continued from previous page)

```
            <multicastLocatorList>
                <!-- LOCATOR_LIST -->
                <locator>
                    <udpv4/>
                </locator>
            </multicastLocatorList>

            <expectsInlineQos>true</expectsInlineQos>

            <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

            <propertiesPolicy>
                <!-- PROPERTIES_POLICY -->
            </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*.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- For `QOS` details, please refer to *QoS*.
- `TOPIC_TYPE` is detailed in section *TopicType*.

Times

Name	Description	Values	Default
`<initialAcknackDelay>`	Initial ACKNACK delay.	*DurationType*	70 ms
`<heartbeatResponseDelay>`	Response time delay when receiving a Heartbeat.	*DurationType*	5 ms

6.24.5 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 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 (**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>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>TransportId1</transport_id>
                <type>UDPv4</type>
                <sendBufferSize>8192</sendBufferSize>
                <receiveBufferSize>8192</receiveBufferSize>
                <TTL>250</TTL>
                <non_blocking_send>false</non_blocking_send>
                <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>
                <output_port>5101</output_port>
                <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>
                <calculate_crc>false</calculate_crc>
                <check_crc>false</check_crc>
                <enable_tcp_nodelay>false</enable_tcp_nodelay>
                <tls><!-- TLS Section --></tls>
                <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.	`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`
		`SERVER`
		`CLIENT`
			`DEFAULT`

An example of TLS protocol parameter configuration is shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <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>
```

(continues on next page)

(continued from previous page)

```
                        <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>
                </tls>
            </transport_descriptor>
        </transport_descriptors>
    </profiles>
</dds>
```

6.24.6 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*.	*Consumer-DataType*	

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>
    <log>
        <!--
        Clear consumers
        -->
        <use_default>FALSE</use_default>

        <!--
        StdoutConsumer does not have any properties
        -->
        <consumer>
            <class>StdoutConsumer</class>
        </consumer>

        <!--
```

(continues on next page)

(continued from previous page)

```
        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
        -->
        <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`

6.24.7 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>
    <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.

- *Enum*
- *Typedef*
- *Struct*
- *Union*
- *Bitset*
- *Bitmask*
- *Member types*
  - *Primitive types*
  - *Arrays*
  - *Sequences*
  - *Maps*
- *Complex types*

Enum

The <enum> type is defined by its attribute name and a set of <enumerator> child elements. Each <enumerator> is defined by two attributes: a name and an optional 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" value="2"/>
</enum>
```

Typedef

The <typedef> XML element is defined by a name and a type mandatory attributes, and various optional attributes for complex types definition. These optional attributes are: key_type, arrayDimensions, nonBasicTypeName, sequenceMaxLength, and mapMaxLength. See *Complex types attributes* for more information on these attributes. The <typedef> element corresponds to *Alias* in *Supported Types* section.

```
<typedef name="MyAliasEnum" type="nonBasic" nonBasicTypeName="MyEnum"/>
<typedef name="MyAliasArray" type="int32" arrayDimension="2,2"/>
```

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 structs. 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. Each <case> element has one or more <caseDiscriminator> and a <member> child elements. 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 can not be higher than 64, and two optional attributes: name and type. A <bitfield> with a blank 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 type will manage. The maximum value allowed for the bit_bound is 64. The <bit_value> element can 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>
```

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.

bool	int32_t	float32
byte	int64_t	float64
char	uint16_t	float128
wchar	uint32_t	string
int16_t	uint64_t	wstring

All of them are defined as follows:

```
<struct name="primitive_types_example">
    <!-- Primitive type definitions inside a struct -->
    <member name="my_long" type="int64"/>
    <member name="my_bool" type="boolean"/>
    <member name="my_string" type="string"/>
</struct>
```

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.

```
<struct name="arrays_example">
    <member name="long_array" type="int32" arrayDimensions="2,3,4"/>
</struct>
```

Sequences

The sequence type is implemented by setting three attributes: `name`, the `type`, and the `sequenceMaxLength`. The type of its content should be defined by the `type` attribute. The following example shows the implementation of a sequence of maximum length equal to 3. In turn, this is a sequence of sequences of maximum length of 2 and contents of type `int32`. Please, refer to *Sequence* section for more information on sequence type.

```
<typedef name="my_sequence_inner" type="int32" sequenceMaxLength="2"/>
<struct name="SeqSeqStruct">
    <member name="my_sequence_sequence" type="nonBasic" nonBasicTypeName="my_sequence_
↪inner" sequenceMaxLength="3"/>
</struct>
```

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. Again, both types can be defined as attributes of a `<typedef>` element, or as a `<member>` child element of a `<struct>` or `<union>` elements. See section *Map* for more information on map type.

```
<typedef name="my_map_inner" type="int32" key_type="int32" mapMaxLength="2"/>
<struct name="MapMapStruct">
    <member name="my_map_map" type="nonBasic" nonBasicTypeName="my_map_inner" key_
↪type="int32" mapMaxLength="2"/>
</struct>
```

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. Please, refer to *Complex Types* section for more information on complex types.

```
<struct name="OtherStruct">
    <member name="my_enum" type="nonBasic" nonBasicTypeName="MyEnum"/>
    <member name="my_struct" type="nonBasic" nonBasicTypeName="MyStruct"␣
↪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`.
`arrayDimensions`	Dimensions of an array.
`sequenceMaxLength`	Maximum length of a *Sequences*.
`mapMaxLength`	Maximum length of a *Maps*.
`key_type`	Data type of a map key.

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;
}
```

6.24.8 Common

The preceding XML profiles define some XML elements that are common to several profiles. This section aims to explain these common elements.

- *LocatorListType*
- *PropertiesPolicyType*
- *DurationType*
- *TopicType*
  - *HistoryQoS*
  - *ResourceLimitsQos*
- *QoS*
  - *Durability*
  - *Liveliness*
  - *Partition*
  - *Deadline*
  - *Lifespan*
  - *DisablePositiveAcks*
  - *LatencyBudget*
- *Throughput Configuration*
- *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.	`uint32_t`	0
`<physical_port>`	TCP's *physical port*.	`uint32_t`	0
`<address>`	IP address of the locator.	`string` (IPv4/IPv6 format)	""
`<unique_lan_id>`	The LAN ID uniquely identifies the LAN the locator belongs to (**TCPv4 only**).	`string` (16 bytes)	
`<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>
    </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>
    </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>
```

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>
```

(continues on next page)

(continued from previous page)

```
            <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 two mandatory elements `<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>

    <leaseDuration>
        <sec>500</sec>
        <nanosec>0</nanosec>
    </leaseDuration>

    <leaseAnnouncement>
        <sec>1</sec>
        <nanosec>856000</nanosec>
    </leaseAnnouncement>
</discovery_config>
```

TopicType

The *Topic* name and data type are used to determine whether Datawriters and DataReaders can exchange messages. Please refer to *Topic* section for a a deeper explanation on the *Topic* class.

Name	Description	Values	Default
`<kind>`	It defines the Topic's key kind. See *Definition of data types*.		
`<name>`	It defines the Topic's name. It must be unique.	`string_255`	
`<dataType>`	It references the Topic's data type.	`string_255`	
`<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*	

Warning: The `<kind>` child element is only used if the Topic is defined using the *Fast DDS* RTPS-layer API, and will be ignored if the Topic is defined via the *Fast DDS* DDS-layer API.

Example

```
<topic>
    <kind>NO_KEY</kind>
    <name>TopicName</name>
    <dataType>TopicDataTypeName</dataType>
    <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>
    </resourceLimitsQos>
</topic>
```

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
`<durability>`	See *DurabilityQosPolicy*.	*Durability*
`<liveliness>`	See *LivelinessQosPolicy*.	*Liveliness*
`<reliability>`	See *ReliabilityQosPolicy*.	*ReliabilityQosPolicy*
`<partition>`	See *PartitionQosPolicy*.	*Partition*
`<deadline>`	See *DeadlineQosPolicy*.	*Deadline*
`<lifespan>`	See *LifespanQosPolicy*.	*Lifespan*
`<disablePositiveAcks>`	See *DisablePositiveACKsQosPolicy*.	*Durability*
`<latencyBudget>`	See *LatencyBudgetQosPolicy*.	*Durability*

Example

```
<qos> <!-- readerQosPoliciesType -->
    <durability>
        <kind>VOLATILE</kind>
    </durability>

    <liveliness>
        <kind>AUTOMATIC</kind>

        <lease_duration>
            <sec>1</sec>
        </lease_duration>

        <announcement_period>
            <sec>1</sec>
        </announcement_period>
    </liveliness>

    <reliability>
        <kind>BEST_EFFORT</kind>
    </reliability>
```

(continues on next page)

(continued from previous page)

```
    <partition>
        <names>
            <name>part1</name>
            <name>part2</name>
        </names>
    </partition>

    <deadline>
        <period>
            <sec>1</sec>
        </period>
    </deadline>

    <lifespan>
        <duration>
            <sec>1</sec>
        </duration>
    </lifespan>

    <disablePositiveAcks>
        <enabled>true</enabled>
    </disablePositiveAcks>
</qos>
```

Durability

Name	Description	Values	Default
<kind>	See *DurabilityQosPolicyKind*.	*VOLATILE*	*VOLATILE*
		TRANSIENT_LOCAL	
		TRANSIENT	
		PERSISTENT	

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*.		*c_TimeInfinite*

ReliabilityQosPolicy

Name	Description	Values	Default
`<kind>`	See *ReliabilityQosPolicyKind*.	*BEST_EFFORT*	DataReaders: *BEST_EFFORT*
		RELIABLE	DataWriters: *RELIABLE*
`<max_blocking_time>`	See *ReliabilityQosPolicy*.	*DurationType*	100 ms

Partition

Name	Description	Values
`<names>`	It comprises a set of `<name>` elements containing the name of each partition. See *PartitionQosPolicy*.	`<name>`

Deadline

Name	Description	Values	Default
`<period>`	See *DeadlineQosPolicy*.	*DurationType*	*c_TimeInfinite*

Lifespan

Name	Description	Values	Default
`<duration>`	See *LifespanQosPolicy*.	*DurationType*	*c_TimeInfinite*

DisablePositiveAcks

Name	Description	Values	Default
`<enabled>`	See *DisablePositiveACKsQosPolicy*.	`bool`	`false`
`<duration>`	See *DisablePositiveACKsQosPolicy*.	*DurationType*	*c_TimeInfinite*

LatencyBudget

Name	Description	Values	Default
`<duration>`	See *LatencyBudgetQosPolicy*.	*DurationType*	0

Throughput Configuration

The `<throughputController>` element allows to limit the output bandwidth. It contains two child elements which are explained in the following table.

Name	Description	Values	Default
`<bytesPerPeriod>`	Packet size in bytes that the throughput controller will allow to send in a given period.	`uint32_t`	4294967295 bytes
`<periodMillisecs>`	Window of time in which no more than `<bytesPerPeriod>` bytes are allowed.	`uint32_t`	0

Example

```
<participant profile_name="participant_thoughput">
    <rtps>
        <throughputController>
            <bytesPerPeriod>8192</bytesPerPeriod>
            <periodMillisecs>1000</periodMillisecs>
        </throughputController>
    </rtps>
</participant>
```

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*
		PREALLOCATED_WITH_REALLOC	
		DYNAMIC	
		DYNAMIC_REUSABLE	

Example

```
<data_writer profile_name="data_writer_historyMemoryPolicy">
    <!-- ...  -->
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</data_writer>

<data_reader profile_name="data_reader_historyMemoryPolicy">
    <!-- ...  -->
    <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

6.24.9 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>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>ExampleTransportId1</transport_id>
                <type>TCPv4</type>
                <sendBufferSize>8192</sendBufferSize>
                <receiveBufferSize>8192</receiveBufferSize>
                <TTL>250</TTL>
                <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>
                <listening_ports>
                    <port>5100</port>
                    <port>5200</port>
                </listening_ports>
            </transport_descriptor>
            <transport_descriptor>
                <transport_id>ExampleTransportId2</transport_id>
                <type>UDPv6</type>
```

(continues on next page)

(continued from previous page)

```
            </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-->
            </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>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>
                </defaultUnicastLocatorList>

                <defaultMulticastLocatorList>
                    <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>
                </defaultMulticastLocatorList>

                <sendSocketBufferSize>8192</sendSocketBufferSize>

                <listenSocketBufferSize>8192</listenSocketBufferSize>

                <builtin>
                    <discovery_config>

                        <discoveryProtocol>NONE</discoveryProtocol>

                        <EDP>SIMPLE</EDP>

                        <leaseDuration>
                            <sec>DURATION_INFINITY</sec>
                        </leaseDuration>

                        <leaseAnnouncement>
                            <sec>1</sec>
                            <nanosec>856000</nanosec>
                        </leaseAnnouncement>

                        <simpleEDP>
                            <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
                            <PUBREADER_SUBWRITER>true</PUBREADER_SUBWRITER>
                        </simpleEDP>

                        <staticEndpointXMLFilename>filename.xml</
↪staticEndpointXMLFilename>

                    </discovery_config>

                    <use_WriterLivelinessProtocol>false</use_WriterLivelinessProtocol>

                    <metatrafficUnicastLocatorList>
                        <locator>
                            <udpv4>
                                <!-- Access as physical, like UDP -->
                                <port>7400</port>
                                <address>192.168.1.41</address>
                            </udpv4>
```

(continues on next page)

(continued from previous page)

```
                        </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>
                    </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>
                        <locator>
                            <tcpv4>
                                <!-- Both physical and logical (port), like TCP -->
                                <physical_port>5100</physical_port>
```

(continues on next page)

(continued from previous page)

```
                                <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>

                    <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</
↪readerHistoryMemoryPolicy>

                    <writerHistoryMemoryPolicy>PREALLOCATED</
↪writerHistoryMemoryPolicy>
                </builtin>

                <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>
                </allocation>

                <port>
                    <portBase>7400</portBase>
                    <domainIDGain>200</domainIDGain>
                    <participantIDGain>10</participantIDGain>
                    <offsetd0>0</offsetd0>
                    <offsetd1>1</offsetd1>
                    <offsetd2>2</offsetd2>
                    <offsetd3>3</offsetd3>
                </port>

```

(continues on next page)

(continued from previous page)

```
                <participantID>99</participantID>

                <throughputController>
                    <bytesPerPeriod>8192</bytesPerPeriod>
                    <periodMillisecs>1000</periodMillisecs>
                </throughputController>

                <userTransports>
                    <transport_id>ExampleTransportId1</transport_id>
                    <transport_id>ExampleTransportId1</transport_id>
                </userTransports>

                <useBuiltinTransports>false</useBuiltinTransports>

                <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>
            </rtps>
        </participant>

        <data_writer profile_name="datawriter_profile_example">
            <topic>
                <kind>WITH_KEY</kind>
                <name>TopicName</name>
                <dataType>TopicDataTypeName</dataType>
                <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>
                </resourceLimitsQos>
            </topic>

            <qos> <!-- dataWriterQosPoliciesType -->
                <durability>
                    <kind>VOLATILE</kind>
                </durability>
                <liveliness>
                    <kind>AUTOMATIC</kind>
                    <lease_duration>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
```

(continues on next page)

(continued from previous page)

```
                    </lease_duration>
                    <announcement_period>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </announcement_period>
                </liveliness>
                <reliability>
                    <kind>BEST_EFFORT</kind>
                    <max_blocking_time>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </max_blocking_time>
                </reliability>
                <lifespan>
                    <duration>
                        <sec>5</sec>
                        <nanosec>0</nanosec>
                    </duration>
                </lifespan>
                <partition>
                    <names>
                        <name>part1</name>
                        <name>part2</name>
                    </names>
                </partition>
                <publishMode>
                    <kind>ASYNCHRONOUS</kind>
                </publishMode>
                <disablePositiveAcks>
                    <enabled>true</enabled>
                        <duration>
                        <sec>1</sec>
                    </duration>
                </disablePositiveAcks>
            </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>

            <unicastLocatorList>
                <locator>
```

(continues on next page)

(continued from previous page)

```
                    <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>

            <throughputController>
                <bytesPerPeriod>8192</bytesPerPeriod>
                <periodMillisecs>1000</periodMillisecs>
            </throughputController>

            <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

```

(continues on next page)

(continued from previous page)

```
            <matchedSubscribersAllocation>
                <initial>3</initial>
                <maximum>3</maximum>
                <increment>0</increment>
            </matchedSubscribersAllocation>

            <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>
        </data_writer>

        <data_reader profile_name="datareader_profile_example">
            <topic>
                <kind>WITH_KEY</kind>
                <name>TopicName</name>
                <dataType>TopicDataTypeName</dataType>
                <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>
                </resourceLimitsQos>
            </topic>

            <qos> <!-- dataReaderQosPoliciesType -->
                <durability>
                    <kind>PERSISTENT</kind>
                </durability>
                <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>
```

(continues on next page)

(continued from previous page)

```
                </liveliness>
                <reliability>
                    <kind>BEST_EFFORT</kind>
                    <max_blocking_time>
                        <sec>1</sec>
                        <nanosec>856000</nanosec>
                    </max_blocking_time>
                </reliability>
                <lifespan>
                    <duration>
                        <sec>5</sec>
                        <nanosec>0</nanosec>
                    </duration>
                </lifespan>
                <partition>
                    <names>
                        <name>part1</name>
                        <name>part2</name>
                    </names>
                </partition>
            </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>
```

(continues on next page)

(continued from previous page)

```
            </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>

            <expectsInlineQos>true</expectsInlineQos>

            <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

            <matchedPublishersAllocation>
                <initial>1</initial>
                <maximum>1</maximum>
                <increment>0</increment>
            </matchedPublishersAllocation>

            <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>
        </data_reader>
```

(continues on next page)

(continued from previous page)

```
    </profiles>

    <log>
        <use_default>FALSE</use_default>

        <consumer>
            <class>StdoutConsumer</class>
        </consumer>

        <consumer>
            <class>FileConsumer</class>
            <property>
                <name>filename</name>
                <value>execution.log</value>
            </property>
            <property>
                <name>append</name>
                <value>TRUE</value>
            </property>
        </consumer>
    </log>

    <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" value="2"/>
            </enum>
        </type>
        <type>
            <enum name="MyEnum">
                <enumerator name="A" value="0"/>
                <enumerator name="B" value="1"/>
                <enumerator name="C" value="2"/>
            </enum>

            <typedef name="MyAlias1" type="nonBasic" nonBasicTypeName="MyEnum"/>

            <typedef name="MyAlias2" type="int32" arrayDimensions="2,2"/>

            <typedef name="my_map_inner" type="int32" key_type="int32" mapMaxLength="2"/>

            <bitset name="MyBitSet">
                <bitfield name="a" bit_bound="3"/>
                <bitfield name="b" bit_bound="10"/>
                <bitfield name="c" bit_bound="12" type="int16"/>
            </bitset>

            <bitmask name="MyBitMask" bit_bound="8">
                <bit_value name="flag0" position="0"/>
                <bit_value name="flag1"/>
            </bitmask>

            <struct name="MyStruct">
```

(continues on next page)

(continued from previous page)

```
647                 <member name="first" type="int32"/>
648                 <member name="second" type="int64"/>
649             </struct>
650
651             <struct name="OtherStruct">
652                 <member name="my_enum" type="nonBasic" nonBasicTypeName="MyEnum"/>
653                 <member name="my_struct" type="nonBasic" nonBasicTypeName="MyStruct"␣
↪arrayDimensions="5"/>
654             </struct>
655
656             <union name="MyUnion1">
657                 <discriminator type="byte"/>
658                 <case>
659                     <caseDiscriminator value="0"/>
660                     <caseDiscriminator value="1"/>
661                     <member name="first" type="int32"/>
662                 </case>
663                 <case>
664                     <caseDiscriminator value="2"/>
665                     <member name="second" type="nonBasic" nonBasicTypeName="MyStruct"/
↪>
666                 </case>
667                 <case>
668                     <caseDiscriminator value="default"/>
669                     <member name="third" type="int64"/>
670                 </case>
671             </union>
672
673             <!-- All possible members struct type -->
674             <struct name="MyFullStruct">
675                 <!-- Primitives & basic -->
676                 <member name="my_bool" type="boolean"/>
677                 <member name="my_byte" type="byte"/>
678                 <member name="my_char" type="char8"/>
679                 <member name="my_wchar" type="char16"/>
680                 <member name="my_short" type="int16"/>
681                 <member name="my_long" type="int32"/>
682                 <member name="my_longlong" type="int64"/>
683                 <member name="my_unsignedshort" type="uint16"/>
684                 <member name="my_unsignedlong" type="uint32"/>
685                 <member name="my_unsignedlonglong" type="uint64"/>
686                 <member name="my_float" type="float32"/>
687                 <member name="my_double" type="float64"/>
688                 <member name="my_longdouble" type="float128"/>
689                 <member name="my_string" type="string"/>
690                 <member name="my_wstring" type="wstring"/>
691                 <member name="my_boundedString" type="string" stringMaxLength="41925"/
↪>
692                 <member name="my_boundedWString" type="wstring" stringMaxLength="41925
↪"/>
693
694                 <!-- long long_array[2][3][4]; -->
695                 <member name="long_array" arrayDimensions="2,3,4" type="int32"/>
696
697                 <!-- map<long,map<long,long,2>,2> my_map_map; -->
698                 <member name="my_map_map" type="nonBasic" nonBasicTypeName="my_map_
↪inner" key_type="int32" mapMaxLength="2"/>
```

(continues on next page)

(continued from previous page)

```
                <!-- Complex types -->
                <member name="my_other_struct" type="nonBasic" nonBasicTypeName=
↪"OtherStruct"/>
            </struct>
        </type>
    </types>
</dds>
```

6.25 Environment variables

This is the list of environment variables that affect the behavior of *Fast DDS*:

6.25.1 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
```

6.25.2 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
```

6.25.3 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 `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 the IP address (e.g., '192.168.2.23') or IP-port pair (e.g., '192.168.2.23:24353').
- If no port is specified, the default port 11811 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.

The following example shows how to set the address of two remote discovery servers with addresses '84.22.259.329:8888' and '81.41.17.102:1234' and IDs 0 and 2 respectively.

Linux

```
export ROS_DISCOVERY_SERVER=84.22.259.329:8888;;81.41.17.102:1234
```

Windows

```
set ROS_DISCOVERY_SERVER=84.22.259.329:8888;;81.41.17.102:1234
```

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`.

6.25.4 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 PropertyPolicyQos Options

This section contains the list of *PropertyPolicyQos* that can be set with *Fast DDS*:

6.26.1 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>
```

6.26.2 Persistence Service Settings

Warning: This section is still under work.

6.26.3 Security Plugins Settings

Warning: This section is still under work.

6.26.4 Logging Module Settings

Warning: This section is still under work.

6.27 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.

6.27.1 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

6.27.2 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());
DynamicType_ptr struct_type(builder->build());
```

(continues on next page)

(continued from previous page)

```
// 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);
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
```

(continues on next page)

(continued from previous page)

```
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 },
        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
```

(continues on next page)

(continued from previous page)

```
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);
```

(continues on next page)

(continued from previous page)

```
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);
```

6.27.3 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);
```

(continues on next page)

(continued from previous page)

```
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());

// 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(
```

(continues on next page)

(continued from previous page)

```
    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);

// 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);
```

6.27.4 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.

6.27.5 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*:

```
DomainParticipantQos qos;
qos.wire_protocol().builtin.typelookup_config.use_client = true;
qos.wire_protocol().builtin.typelookup_config.use_server = true;
```

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.

6.27.6 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`.

6.27.7 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.

6.27.8 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.

6.27.9 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/C++/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/C++/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/C++/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.

6.28 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:

- *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:
  - 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.
- *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 the OSRF Robot Operation System 2 (ROS 2), 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.

6.28.1 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.

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>
                        </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>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
```

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>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
```

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 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 client of the other.

Consider two servers, each one managing an isolated network:

Network	Prefix	UDPv4 address
A	75.63.2D.73.76.72	192.168.1.45:7413
B	75.63.2D.73.76.72	192.168.1.57:7412

In order to communicate both networks we can set server A to act as 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>
        </rtps>
    </participant>
</profiles>
```

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.41	192.168.1.43
B	75.63.2D.73.76.72.2D.42	192.168.1.42
C	75.63.2D.73.76.72.2D.43	192.168.1.41

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>
                        </RemoteServer>
                        <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.31">
                            <metatrafficUnicastLocatorList>
```

6.28.2 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 case.

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/C++/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

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_static_pub">
        <rtps>
            <name>HelloWorldPublisher</name>
```

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

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <participant profile_name="participant_profile_static_sub">
        <rtps>
            <name>HelloWorldSubscriber</name>
```

6.28.3 Large Data Rates

When the amount of data exchanged between a *Publisher* and a *Subscriber* is large, some tuning 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 limitations have to be taken into account:

- 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 be fragmented into several transport packets.

Warning: The loss of a fragment means the loss of the entire message. This has 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
```

Flow Controllers

eProsima Fast DDS provides a mechanism to limit the rate at which the data is sent by a DataWriter. These controllers can be configured at DataWriter or DomainParticipant level. On the DomainParticipant the throughput controller is configured on the *wire_protocol()* member function, while the *DataWriterQos* uses the *throughput_controller()* member function.

C++

```
// Limit to 300kb per second.
ThroughputControllerDescriptor slowPublisherThroughputController{300000, 1000};

DataWriterQos qos;
qos.throughput_controller(slowPublisherThroughputController);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <publisher profile_name="publisher_profile_qos_flowcontroller">
        <throughputController>
            <bytesPerPeriod>300000</bytesPerPeriod>
            <periodMillisecs>1000</periodMillisecs>
        </throughputController>
    </publisher>
</profiles>
```

Warning: Specifying a throughput 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_ALL_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.

6.28.4 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">
    <subscriber profile_name="subscriber_xml_conf_multicast_locators_profile">
        <multicastLocatorList>
            <locator>
                <udpv4>
                    <address>239.255.0.4</address>
                    <port>7900</port>
                </udpv4>
            </locator>
        </multicastLocatorList>
    </subscriber>
</profiles>
```

6.28.5 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.

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;
```

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>
            </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 an internal collection with information about every DataReader to which it matches. By default, this collection is 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 a data member `matched_subscriber_allocation` of type *ResourceLimitedContainerConfig* that allows configuring the maximum expected size of the collection of matched DataReader, so that it 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 this data member.

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);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <publisher profile_name="writer_profile_qos_resource_limit">
        <!-- Limit to 3 matching readers -->
        <matchedSubscribersAllocation>
            <initial>3</initial>
            <maximum>3</maximum>
            <increment>0</increment>
        </matchedSubscribersAllocation>
    </publisher>
</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">
    <subscriber profile_name="reader_profile_qos_resource_limit">
        <!-- Limit to 1 matching writer -->
        <matchedPublishersAllocation>
            <initial>1</initial>
            <maximum>1</maximum>
            <increment>0</increment>
        </matchedPublishersAllocation>
    </subscriber>
</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 4: **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 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;


// DataWriter configuration for Topic 1
////////////////////////////////////////
DataWriterQos writer1_qos;

// We know we will only have three matching subscribers
writer1_qos.writer_resource_limits().matched_subscriber_allocation =
        eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);

// 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);


// DataReader configuration for both Topics
///////////////////////////////////////////
DataReaderQos reader_qos;

// We know we will only have one matching publisher
reader_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">
    <participant profile_name="participant_alloc_qos_example">
        <rtps>
            <allocation>
                <!-- We know we have 3 participants on the domain -->
                <total_participants>
```

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 5: **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.	

6.28.6 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_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_reader>
</profiles>
```

Set Dynamic Allocation

By default *MemoryManagementPolicy* is set to *PREALLOCATED_MEMORY_MODE*, meaning that the amount of memory required by the configured *ResourceLimitsQosPolicy* will be allocated at initialization.

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>
```

6.28.7 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

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
// prepared for sending.
if (ReturnCode_t::RETCODE_OK == writer->loan_sample(sample))
```

(continues on next page)

(continued from previous page)

```
{
    // 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();

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`.

- 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)
{
    // 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 << 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.
- Suitable for `PREALLOCATED_MEMORY_MODE` and `PREALLOCATED_WITH_REALLOC_MEMORY_MODE` memory configurations only.

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.

6.28.8 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

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;
```

(continues on next page)

(continued from previous page)

```
}

// 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;
}
```

6.28.9 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

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. 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;
}
```

(continues on next page)

(continued from previous page)

```
// 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)
{
    // 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 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.

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 is `rmw_fastrtps_cpp`. However, it is still possible to select `rmw_fastrtps_dynamic_cpp` by using the environment variable `RMW_IMPLEMENTATION`:

1. Exporting `RMW_IMPLEMENTATION` environment variable:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_dynamic_cpp
```

2. When launching your ROS 2 application:

```
RMW_IMPLEMENTATION=rmw_fastrtps_dynamic_cpp ros2 run <package> <application>
```

6.29.1 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

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_MEM*
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 `<publisher>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic before mangling, i.e., the topic name used to create the publisher. If such profile is not defined, *rmw_fastrtps* attempts to load the `<publisher>` profile with attribute `is_default_profile="true"`.
- To configure a subscriber, define a `<subscriber>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic before mangling. If such profile is not defined, *rmw_fastrtps* attempts to load the `<subscriber>` profile with attribute `is_default_profile="true"`.

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 `<subscriber>` 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 `<subscriber>` profile with attribute `profile_name="service"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<subscriber>` profile with attribute `is_default_profile="true"`.
- To configure the reply publisher, define a `<publisher>` 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 `<publisher>` profile with `attribute profile_name="service"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<publisher>` 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 `<publisher>` 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 `<publisher>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<publisher>` profile with attribute `is_default_profile="true"`.
- To configure the reply subscription, define a `<subscriber>` 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 `<subscriber>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<subscriber>` 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 in *Foxy*	*Fast DDS* entity in *Eloquent & below*
Context	Participant	*Not DDS direct mapping*
Node	*Not DDS direct mapping*	Participant

This means that on *Foxy*, 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* would be specified as:

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>
</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 -->
    <publisher profile_name="default publisher profile" is_default_profile=
↪"true">
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </publisher>

    <!-- Publisher profile for topic helloworld -->
    <publisher profile_name="helloworld">
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </publisher>

    <!-- Request subscriber profile for services -->
    <subscriber profile_name="service">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </subscriber>

    <!-- Request publisher profile for clients -->
    <publisher profile_name="client">
        <qos>
            <publishMode>
                <kind>ASYNCHRONOUS</kind>
            </publishMode>
        </qos>
    </publisher>

    <!-- Request subscriber profile for server of service "add_two_ints" -->
    <subscriber profile_name="rq/add_two_intsRequest">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </subscriber>

    <!-- Reply subscriber profile for client of service "add_two_ints" -->
    <subscriber profile_name="rr/add_two_intsReply">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </subscriber>
</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
```

6.29.2 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.

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 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*.

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
$ 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.

6.30 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.

6.30.1 DDS DCPS PIM

Data Distribution Service (DDS) Data-Centric Publish-Subscribe (DCPS) Platform Independent Model (PIM) API

Core

Entity

class eprosima::fastdds::dds::**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

Entity(**const** *StatusMask* &*mask* = *StatusMask*::*all*())
Constructor.

Parameters

- `mask`: *StatusMask* (default: all)

fastrtps::types::ReturnCode_t **enable**()
This operation enables the *Entity*.

Return RETCODE_OK

void **close**()
This operation disables the *Entity* before closing it.

const *StatusMask* &**get_status_mask**() **const**
Retrieves the set of relevant statuses for the *Entity*.

Return 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.

Return const reference to the *StatusMask* with the triggered statuses set to 1

const InstanceHandle_t &**get_instance_handle**() **const**
Retrieves the instance handler that represents the *Entity*.

Return Reference to the InstanceHandle

bool **is_enabled**() **const**
Checks if the *Entity* is enabled.

Return true if enabled, false if not

const StatusCondition &**get_statuscondition**() **const**
Allows access to the StatusCondition associated with the *Entity*.

Return Reference to StatusCondition object

DomainEntity

class eprosima::fastdds::dds::**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

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 = 0

Extended CDR Encoding version 1.

enumerator XML_DATA_REPRESENTATION = 1

XML Data Representation (Unsupported)

enumerator XCDR2_DATA_REPRESENTATION = 2

Extended CDR Encoding version 2.

DataRepresentationQosPolicy

class eprosima::fastdds::dds::**DataRepresentationQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **p**

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.

Note Immutable Qos Policy

Public Functions

DataRepresentationQosPolicy()
Constructor.

~DataRepresentationQosPolicy() **override** = default
Destructor.

bool **operator==**(**const** *DataRepresentationQosPolicy* &*b*) **const**
Compares the given policy to check if it's equal.

Return True if the policy is equal.

Parameters

- b: QoS Policy.

void **clear**() **override**
Clears the *QosPolicy* object.

Public Members

std::vector<DataRepresentationId_t> **m_value**
List of DataRepresentationId. By default, empty list.

DataSharingQosPolicy

class eprosima::fastdds::dds::**DataSharingQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** ep
Qos Policy to configure the data sharing

Note Immutable Qos Policy

Public Functions

DataSharingQosPolicy()
Constructor.

~DataSharingQosPolicy() = default
Destructor.

DataSharingQosPolicy(**const** *DataSharingQosPolicy* &*b*)
Copy constructor.

Parameters

- b: Another *DataSharingQosPolicy* instance

void **clear**() **override**
Clears the *QosPolicy* object.

const *DataSharingKind* &**kind**() **const**

Return the current DataSharing configuration mode

const std::string &**shm_directory**() **const**

Return the current DataSharing shared memory directory

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.

Return the current DataSharing domain IDs

void **set_max_domains**(uint32_t *size*)

Parameters

- `size`: the new maximum number of domain IDs

const uint32_t &**max_domains**() **const**

Return the current configured maximum number of domain IDs

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.

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).

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.

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).

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.

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).

void **off**()
Configures the DataSharing in disabled mode.

void **add_domain_id**(uint16_t *id*)
Adds a user-specific DataSharing domain ID.

Parameters

- `id`: 16 bit identifier

DataSharingKind

enum `eprosima::fastdds::dds::`**DataSharingKind**
Data sharing configuration kinds

Values:

enumerator AUTO = 0x01
Automatic configuration. DataSharing will be used if requirements are met.

enumerator ON = 0x02
Activate the use of DataSharing. *Entity* creation will fail if requirements for DataSharing are not met

enumerator OFF = 0x03
Disable the use of DataSharing

DeadlineQosPolicy

class `eprosima::fastdds::dds::`**DeadlineQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosi
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

DeadlineQosPolicy()
Constructor.

~DeadlineQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**DestinationOrderQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **pub**
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.

Note Immutable Qos Policy

Public Functions

DestinationOrderQosPolicy()
Constructor.

~DestinationOrderQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**DisablePositiveACKsQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t,
Class *DisablePositiveACKsQosPolicy* to disable sending of positive ACKs

Note Immutable Qos Policy

Public Functions

DisablePositiveACKsQosPolicy()
Constructor.

~DisablePositiveACKsQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**DurabilityQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** epr
This policy expresses if the data should 'outlive' their writing time.

Note Immutable Qos Policy

Public Functions

DurabilityQosPolicy()
Constructor.

~DurabilityQosPolicy() = default
Destructor.

fastrtps::rtps::*DurabilityKind_t* **durabilityKind**() **const**
Translates kind to rtps layer equivalent

Return fastrtps::rtps::DurabilityKind_t

void **durabilityKind**(**const** fastrtps::rtps::*DurabilityKind_t* *new_kind*)
Set kind passing the rtps layer equivalent kind

Parameters

- `new_kind`: fastrtps::rtps::DurabilityKind_t

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 eprosima::fastdds::dds::**DurabilityServiceQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **pu**

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.

Note Immutable Qos Policy

Public Functions

DurabilityServiceQosPolicy()
Constructor.

~DurabilityServiceQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**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

EntityFactoryQosPolicy()

Constructor without parameters.

EntityFactoryQosPolicy(bool *autoenable*)

Constructor.

Parameters

- `autoenable`: Value for the autoenable_created_entities boolean

~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 eprosima::fastdds::dds::**GenericDataQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** ep

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

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

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

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.

Return reference to the current object.

Parameters

- `b`: object to be copied

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.

Return reference to the current object.

Parameters

- `b`: object to be copied

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

const collection_type &**dataVec**() **const**

Return const reference to the internal raw data.

void **clear**() **override**
Clears the *QosPolicy* object.

const collection_type &**data_vec**() **const**
Returns raw data vector.

Return raw data as vector of octets.

collection_type &**data_vec**()
Returns raw data vector.

Return raw data as vector of octets.

void **data_vec**(**const** collection_type &*vec*)
Sets raw data vector.

Parameters

- `vec`: raw data to set.

const collection_type &**getValue**() **const**
Returns raw data vector.

Return raw data as vector of octets.

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 eprosima::fastdds::dds::**HistoryQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosim
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

HistoryQosPolicy()
Constructor.

~HistoryQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**LatencyBudgetQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public**
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.

Note Mutable Qos Policy

Public Functions

LatencyBudgetQosPolicy()
Constructor.

~LatencyBudgetQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**LifespanQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosi
Specifies the maximum duration of validity of the data written by the *DataWriter*.

Note Mutable Qos Policy

Public Functions

LifespanQosPolicy()
Constructor.

~LifespanQosPolicy() = default
Destructor.

void **clear**() **override**
Clears the *QosPolicy* object.

Public Members

fastrtps::*Duration_t* **duration**
Period of validity. By default, c_TimeInfinite.

LivelinessQosPolicy

class eprosima::fastdds::dds::**LivelinessQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** epr
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

LivelinessQosPolicy()
Constructor.

~LivelinessQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**OwnershipQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** epros

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

OwnershipQosPolicy()

Constructor.

~OwnershipQosPolicy() = default

Destructor.

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 eprosima::fastdds::dds::**OwnershipStrengthQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **pu**

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

OwnershipStrengthQosPolicy()
Constructor.

~OwnershipStrengthQosPolicy() = default
Destructor.

void **clear**() **override**
Clears the *QosPolicy* object.

Public Members

uint32_t **value**
Strength By default, 0.

ParticipantResourceLimitsQos

using eprosima::fastdds::dds::**ParticipantResourceLimitsQos** = fastrtps::rtps::*RTPSParticipantAllocationAttri*
Holds allocation limits affecting collections managed by a participant.

Partition_t

class eprosima::fastdds::dds::**Partition_t**

Public Functions

Partition_t(**const** void *_ptr_)
Constructor using a pointer.

Parameters

- ptr: Pointer to be set

uint32_t **size**() **const**
Getter for the size.

Return uint32_t with the size

const char ***name**() **const**
Getter for the partition name.

Return name

PartitionQosPolicy

class eprosima::fastdds::dds::**PartitionQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** epros

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

PartitionQosPolicy()

Constructor without parameters.

PartitionQosPolicy(uint16_t *in_length*)

Constructor using Parameter length.

Parameters

- `in_length`: Length of the parameter

PartitionQosPolicy(**const** *PartitionQosPolicy* &*b*)

Copy constructor.

Parameters

- `b`: Another *PartitionQosPolicy* instance

~PartitionQosPolicy() = default

Destructor.

const_iterator **begin**() **const**

Getter for the first position of the partition list.

Return *const_iterator*

const_iterator **end**() **const**

Getter for the end of the partition list.

Return *const_iterator*

uint32_t **size**() **const**

Getter for the number of partitions.

Return uint32_t with the size

uint32_t **empty**() **const**

Check if the set is empty.

Return true if it is empty, false otherwise

void **set_max_size**(uint32_t *size*)

Setter for the maximum size.

Parameters

- `size`: Size to be set

uint32_t **max_size**() **const**
Getter for the maximum size.

Return uint32_t with the maximum size

void **push_back**(**const** char ***name*)
Appends a name to the list of partition names.

Parameters

- `name`: Name to append.

void **clear**() **override**
Clears list of partition names

const std::vector<std::string> **getNames**() **const**
Returns partition names.

Return Vector of partition name strings.

void **setNames**(std::vector<std::string> &*nam*)
Overrides partition names

Parameters

- `nam`: Vector of partition name strings.

const std::vector<std::string> **names**() **const**
Returns partition names.

Return Vector of partition name strings.

void **names**(std::vector<std::string> &*nam*)
Overrides partition names

Parameters

- `nam`: Vector of partition name strings.

class const_iterator

Public Functions

const_iterator(**const** fastrtps::rtps::*octet* **ptr*)
Constructor using a pointer.

Parameters
- `ptr`: Pointer to be set

PresentationQosPolicy

class eprosima::fastdds::dds::**PresentationQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** e

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.

Note Immutable Qos Policy

Public Functions

PresentationQosPolicy()
Constructor without parameters.

~PresentationQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**PublishModeQosPolicy** : **public** eprosima::fastdds::dds::*QosPolicy*
Class *PublishModeQosPolicy*, defines the publication mode for a specific writer.

Public Functions

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 Functionality not implemented yet. Coming soon.

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 eprosima::fastdds::dds::**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

QosPolicy()

Constructor without parameters.

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*

~QosPolicy() = default

Destructor.

bool **send_always**() **const**

Whether it should always be sent.

Return True if it should always be sent.

void **clear**() = 0

Clears the *QosPolicy* object.

Public Members

bool **hasChanged**
Boolean that indicates if the Qos has been changed.

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 = 0

enumerator USERDATA_QOS_POLICY_ID = 1

enumerator DURABILITY_QOS_POLICY_ID = 2

enumerator PRESENTATION_QOS_POLICY_ID = 3

enumerator DEADLINE_QOS_POLICY_ID = 4

enumerator LATENCYBUDGET_QOS_POLICY_ID = 5

enumerator OWNERSHIP_QOS_POLICY_ID = 6

enumerator OWNERSHIPSTRENGTH_QOS_POLICY_ID = 7

enumerator LIVELINESS_QOS_POLICY_ID = 8

enumerator TIMEBASEDFILTER_QOS_POLICY_ID = 9

enumerator PARTITION_QOS_POLICY_ID = 10

enumerator RELIABILITY_QOS_POLICY_ID = 11

enumerator DESTINATIONORDER_QOS_POLICY_ID = 12

enumerator HISTORY_QOS_POLICY_ID = 13

enumerator RESOURCELIMITS_QOS_POLICY_ID = 14

enumerator ENTITYFACTORY_QOS_POLICY_ID = 15

enumerator WRITERDATALIFECYCLE_QOS_POLICY_ID = 16

enumerator READERDATALIFECYCLE_QOS_POLICY_ID = 17

enumerator TOPICDATA_QOS_POLICY_ID = 18

enumerator GROUPDATA_QOS_POLICY_ID = 19

enumerator TRANSPORTPRIORITY_QOS_POLICY_ID = 20

enumerator LIFESPAN_QOS_POLICY_ID = 21

enumerator DURABILITYSERVICE_QOS_POLICY_ID = 22

enumerator DATAREPRESENTATION_QOS_POLICY_ID = 23

enumerator TYPECONSISTENCYENFORCEMENT_QOS_POLICY_ID = 24

enumerator DISABLEPOSITIVEACKS_QOS_POLICY_ID = 25

enumerator PARTICIPANTRESOURCELIMITS_QOS_POLICY_ID = 26

enumerator PROPERTYPOLICY_QOS_POLICY_ID = 27

enumerator PUBLISHMODE_QOS_POLICY_ID = 28

enumerator READERRESOURCELIMITS_QOS_POLICY_ID = 29

enumerator RTPSENDPOINT_QOS_POLICY_ID = 30

enumerator RTPSRELIABLEREADER_QOS_POLICY_ID = 31

enumerator RTPSRELIABLEWRITER_QOS_POLICY_ID = 32

enumerator TRANSPORTCONFIG_QOS_POLICY_ID = 33

enumerator TYPECONSISTENCY_QOS_POLICY_ID = 34

enumerator WIREPROTOCOLCONFIG_QOS_POLICY_ID = 35

enumerator WRITERRESOURCELIMITS_QOS_POLICY_ID = 36

enumerator NEXT_QOS_POLICY_ID

ReaderDataLifecycleQosPolicy

class eprosima::fastdds::dds::**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.

Note Mutable Qos Policy

Public Functions

ReaderDataLifecycleQosPolicy()

Constructor.

~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 eprosima::fastdds::dds::**ReliabilityQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** ep

Indicates the reliability of the endpoint.

Note Immutable Qos Policy

Public Functions

ReliabilityQosPolicy()

Constructor.

~ReliabilityQosPolicy() = default

Destructor.

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 = 0x01

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 = 0x02

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 eprosima::fastdds::dds::**ResourceLimitsQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **publi**

Specifies the resources that the Service can consume in order to meet the requested QoS

Note Immutable Qos Policy

Public Functions

ResourceLimitsQosPolicy()

Constructor.

~ResourceLimitsQosPolicy() = default

Destructor.

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*). By default, 5000.

Warning It is inconsistent for this value to be less than max_samples_per_instance.

int32_t **max_instances**

Represents the maximum number of instances *DataWriter* (or *DataReader*) can manage. By default, 10.

int32_t **max_samples_per_instance**

Represents the maximum number of samples of any one instance a DataWriter(or DataReader) can manage. By default, 400.

Warning It is inconsistent for this value to be greater than 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 eprosima::fastdds::dds::**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.

int16_t **user_defined_id**
User Defined ID, used for StaticEndpointDiscovery. By default, -1.

int16_t **entity_id**
Entity ID, if the user wants to specify the EntityID of the endpoint. By default, -1.

fastrtps::rtps::MemoryManagementPolicy_t **history_memory_policy**
Underlying History memory policy. By default, PREALLOCATED_MEMORY_MODE.

TimeBasedFilterQosPolicy

class eprosima::fastdds::dds::**TimeBasedFilterQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **publ**
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.

Note Mutable Qos Policy

Public Functions

TimeBasedFilterQosPolicy()
Constructor.

~TimeBasedFilterQosPolicy() = default
Destructor.

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 eprosima::fastdds::dds::**TransportConfigQos** : **public** eprosima::fastdds::dds::*QosPolicy*
Qos Policy to configure the transport layer.

Public Functions

TransportConfigQos()
Constructor.

~TransportConfigQos() = default
Destructor.

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.

TransportPriorityQosPolicy

class eprosima::fastdds::dds::**TransportPriorityQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **pu**

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.

Note Mutable Qos Policy

Public Functions

TransportPriorityQosPolicy()
Constructor.

~TransportPriorityQosPolicy() = default
Destructor.

void **clear**() **override**
Clears the *QosPolicy* object.

Public Members

uint32_t **value**
Priority By default, 0.

TypeConsistencyEnforcementQosPolicy

class eprosima::fastdds::dds::**TypeConsistencyEnforcementQosPolicy** : **public** eprosima::fastdds::dds::Pa

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

TypeConsistencyEnforcementQosPolicy()
Constructor.

~TypeConsistencyEnforcementQosPolicy() **override** = default
Destructor.

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 eprosima::fastdds::dds::**WireProtocolConfigQos** : **public** eprosima::fastdds::dds::*QosPolicy*

Qos Policy that configures the wire protocol.

Public Functions

WireProtocolConfigQos()

Constructor.

~WireProtocolConfigQos() = default

Destructor.

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.

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 UnicastLocators. This is usually left empty.

WriterDataLifecycleQosPolicy

class eprosima::fastdds::dds::**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.

Note Mutable Qos Policy

Public Functions

WriterDataLifecycleQosPolicy()

Constructor.

~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 eprosima::fastdds::dds::**WriterResourceLimitsQos**

Qos Policy to configure the limit of the writer resources.

Public Functions

WriterResourceLimitsQos()

Constructor.

~WriterResourceLimitsQos() = default

Destructor.

Public Members

fastrtps::ResourceLimitedContainerConfig **matched_subscriber_allocation**

Matched subscribers allocation limits.

Status

BaseStatus

struct eprosima::fastdds::dds::**BaseStatus**
A struct storing the base status.

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 eprosima::fastdds::dds::**DeadlineMissedStatus**
A struct storing the deadline status.

Public Functions

DeadlineMissedStatus()
Constructor.

~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 eprosima::fastdds::dds::**IncompatibleQosStatus**
A struct storing the requested incompatible QoS status.

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 eprosima::fastdds::dds::**LivelinessChangedStatus**
: A struct storing the liveliness changed status.

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 eprosima::fastdds::dds::**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 eprosima::fastdds::dds::**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 eprosima::fastdds::dds::**QosPolicyCount**
A struct storing the id of the incompatible QoS Policy and the number of times it fails.

Public Functions

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 eprosima::fastdds::dds::**SampleRejectedStatus**
A struct storing the sample lost 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 eprosima::fastdds::dds::**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

StatusMask()

Construct an *StatusMask* with no flags set.

StatusMask(uint32_t *mask*)

Construct an *StatusMask* with an uint32_t bit mask.

Parameters

- `mask`: the bit array to initialize the bitset with

StatusMask &**operator<<**(**const** *StatusMask* &*mask*)

Add given *StatusMask* bits into this *StatusMask* bitset.

Return *StatusMask* this

StatusMask &**operator>>**(**const** *StatusMask* &*mask*)

Remove given *StatusMask* bits into this *StatusMask* bitset.

Return *StatusMask* this

bool **is_active**(*StatusMask status*) **const**

Checks if the status passed as parameter is 1 in the actual *StatusMask*.

Return true if the status is active and false if not

Parameters

- `status`: Status that need to be checked

Public Static Functions

StatusMask **all**()
Get all StatusMasks

Return *StatusMask* all

StatusMask **none**()
Get no StatusMasks

Return *StatusMask* none

StatusMask **inconsistent_topic**()
Get the *StatusMask* associated with dds::core::status::InconsistentTopicStatus

Return *StatusMask* inconsistent_topic

StatusMask **offered_deadline_missed**()
Get the *StatusMask* associated with dds::core::status::OfferedDeadlineMissedStatus

Return *StatusMask* offered_deadline_missed

StatusMask **requested_deadline_missed**()
Get the *StatusMask* associated with dds::core::status::RequestedDeadlineMissedStatus

Return *StatusMask* requested_deadline_missed

StatusMask **offered_incompatible_qos**()
Get the *StatusMask* associated with dds::core::status::OfferedIncompatibleQosStatus

Return *StatusMask* offered_incompatible_qos

StatusMask **requested_incompatible_qos**()
Get the *StatusMask* associated with dds::core::status::RequestedIncompatibleQosStatus

Return *StatusMask* requested_incompatible_qos

StatusMask **sample_lost**()
Get the *StatusMask* associated with dds::core::status::SampleLostStatus

Return *StatusMask* sample_lost

StatusMask **sample_rejected**()
Get the *StatusMask* associated with dds::core::status::SampleRejectedStatus

Return *StatusMask* sample_rejected

StatusMask **data_on_readers**()
Get the *StatusMask* associated with dds::core::status::data_on_readers

Return *StatusMask* data_on_readers

StatusMask **data_available**()
get the statusmask associated with dds::core::status::data_available

Return statusmask data_available

StatusMask **liveliness_lost**()
Get the *StatusMask* associated with dds::core::status::LivelinessLostStatus

Return *StatusMask* liveliness_lost

StatusMask **liveliness_changed**()
Get the *StatusMask* associated with dds::core::status::LivelinessChangedStatus

Return *StatusMask* liveliness_changed

StatusMask **publication_matched**()
Get the statusmask associated with dds::core::status::PublicationMatchedStatus

Return *StatusMask* publication_matched

StatusMask **subscription_matched**()
Get the statusmask associated with dds::core::status::SubscriptionMatchedStatus

Return *StatusMask* subscription_matched

FASTDDS_STATUS_COUNT
Alias of size_t(16)

SubscriptionMatchedStatus

struct eprosima::fastdds::dds::**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.

LoanableArray

template<typename **T**, std::size_t **num_items**>
struct eprosima::fastdds::dds::**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

void **buffer_for_loans() const
Get a buffer pointer that could be used on *LoanableCollection::loan*.

Return buffer pointer for loans.

LoanableCollection

class eprosima::fastdds::dds::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 >, eprosima::fastdds::dds::UserAllocatedSequence

Public Functions

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.

Return the pointer to the elements buffer.

bool has_ownership() const
Get the ownership flag.

Return whether the collection has ownership of the buffer.

size_type maximum() const
Get the maximum number of elements currently allocated.

Return the maximum number of elements currently allocated.

size_type length() const
Get the number of elements currently accessible.

Return the number of elements currently accessible.

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.

Pre new_length >= 0

Return true if the new length was correctly set.

Post *length()* == new_length

Post *maximum()* >= new_length

Parameters

- `[in] new_length`: New number of elements to be accessible.

bool **loan**(element_type *_buffer_, size_type _new_maximum_, size_type _new_length_)
Loan a buffer to the collection.

Pre (_has_ownership()_ == false) || (_maximum()_ == 0)

Pre new_maximum > 0

Pre new_maximum >= new_length

Pre buffer != nullptr

Return false if preconditions are not met.

Return true if operation succeeds.

Post _buffer()_ == buffer

Post _has_ownership()_ == false

Post _maximum()_ == new_maximum

Post _length()_ == new_length

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.

element_type ***unloan**(size_type &_maximum_, size_type &_length_)
Remove the loan from the collection.

Pre _has_ownership()_ == false

Return nullptr if preconditions are not met.

Return pointer to the previously loaned buffer of elements.

Post _buffer()_ == nullptr

Post _has_ownership()_ == true

Post _length()_ == 0

Post _maximum()_ == 0

Parameters

- `[out] maximum`: number of allocated elements on the returned buffer.
- `[out] length`: number of accessible elements on the returned buffer.

element_type ***unloan**()
Remove the loan from the collection.

Pre _has_ownership()_ == false

Return nullptr if preconditions are not met.

Return 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**>
class `eprosima::fastdds::dds::`**LoanableSequence** : **public** eprosima::fastdds::dds::LoanableTypedCollection<*T*>

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

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.

Post *buffer()* != nullptr

Post *has_ownership()* == true

Post *length()* == 0

Post *maximum()* == max

Parameters

- `[in] max`: Number of elements to pre-allocate.

~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.

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

Post *has_ownership()* == true

Post *maximum()* == other.length()

Post *length()* == other.length()

Post *buffer()* != nullptr when other.length() > 0

Parameters

- `[in] other`: The sequence from where contents are to be copied.

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()*

Post *has_ownership()* == true

Post *maximum()* >= other.length()

Post *length()* == other.length()

Post *buffer()* != nullptr when other.length() > 0

Parameters

- `[in] other`: The sequence from where contents are to be copied.

FASTDDS_SEQUENCE(*FooSeq*, *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 eprosima::fastdds::dds::**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

~DomainParticipant()
Destructor.

ReturnCode_t **get_qos**(*DomainParticipantQos* &*qos*) **const**
This operation returns the value of the *DomainParticipant* QoS policies

Return RETCODE_OK

Parameters

- qos: *DomainParticipantQos* reference where the qos is going to be returned

const *DomainParticipantQos* &**get_qos**() **const**
This operation returns the value of the *DomainParticipant* QoS policies.

Return A reference to the *DomainParticipantQos*

ReturnCode_t **set_qos**(**const** *DomainParticipantQos* &*qos*) **const**
This operation sets the value of the *DomainParticipant* QoS policies.

Return 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.

Parameters

- qos: *DomainParticipantQos* to be set

const *DomainParticipantListener* ***get_listener**() **const**
Allows accessing the *DomainParticipantListener*.

Return *DomainParticipantListener* pointer

ReturnCode_t **set_listener**(*DomainParticipantListener* **listener*)
Modifies the *DomainParticipantListener*, sets the mask to *StatusMask::all()*

Return RETCODE_OK

Parameters

- `listener`: new value for the *DomainParticipantListener*

ReturnCode_t **set_listener**(*DomainParticipantListener* *listener, **const** *StatusMask* &mask)
Modifies the *DomainParticipantListener*.

Return RETCODE_OK

Parameters

- `listener`: new value for the *DomainParticipantListener*
- `mask`: *StatusMask* that holds statuses the listener responds to

ReturnCode_t **enable**() **override**
This operation enables the *DomainParticipant*.

Return RETCODE_OK

Publisher ***create_publisher**(**const** *PublisherQos* &qos, *PublisherListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())
Create a *Publisher* in this Participant.

Return Pointer to the created *Publisher*.

Parameters

- `qos`: QoS of the *Publisher*.
- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all)

Publisher ***create_publisher_with_profile**(**const** std::string &profile_name, *PublisherListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())
Create a *Publisher* in this Participant.

Return Pointer to the created *Publisher*.

Parameters

- `profile_name`: *Publisher* profile name.
- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all)

ReturnCode_t **delete_publisher**(**const** *Publisher* *publisher)
Deletes an existing *Publisher*.

Return 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.

Parameters

- `publisher`: to be deleted.

Subscriber ***create_subscriber**(**const** *SubscriberQos* &qos, *SubscriberListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())
Create a *Subscriber* in this Participant.

Return Pointer to the created *Subscriber*.

Parameters

- `qos`: QoS of the *Subscriber*.

- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all)

Subscriber ***create_subscriber_with_profile**(**const** std::string &*profile_name*, *SubscriberListener* **listener* = nullptr, **const** *StatusMask* &*mask* = *StatusMask*::*all*())

Create a *Subscriber* in this Participant.

Return Pointer to the created *Subscriber*.

Parameters

- `profile_name`: *Subscriber* profile name.
- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all)

ReturnCode_t **delete_subscriber**(**const** *Subscriber* **subscriber*)

Deletes an existing *Subscriber*.

Return 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.

Parameters

- `subscriber`: to be deleted.

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.

Return Pointer to the created *Topic*.

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)

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.

Return Pointer to the created *Topic*.

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)

ReturnCode_t **delete_topic**(**const** *Topic* *topic)
Deletes an existing *Topic*.

Return 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.

Parameters

- `topic`: to be deleted.

ContentFilteredTopic *__create_contentfilteredtopic__(**const** std::string &*name*, **const** *Topic* *related_topic, **const** std::string &*filter_expression*, **const** std::vector<std::string> &*expression_parameters*)
Create a ContentFilteredTopic in this Participant.

Return Pointer to the created ContentFilteredTopic, nullptr in error case

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

ReturnCode_t **delete_contentfilteredtopic**(**const** ContentFilteredTopic *a_contentfilteredtopic)
Deletes an existing ContentFilteredTopic.

Return 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.

Parameters

- `a_contentfilteredtopic`: ContentFilteredTopic to be 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.

Return Pointer to the created ContentFilteredTopic, nullptr in error case

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

ReturnCode_t **delete_multitopic**(**const** MultiTopic *a_multitopic)
Deletes an existing MultiTopic.

Return 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.

Parameters

- `a_multitopic`: MultiTopic to be 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*. Topics obtained by this method must be destroyed by delete_topic.

Return Pointer to the existing *Topic*, nullptr in error case

Parameters

- `topic_name`: *Topic* name
- `timeout`: Maximum time to wait for the *Topic*

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.

Return Pointer to the topic description, if it has been created locally. Otherwise, nullptr is returned.

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.

const *Subscriber* ***get_builtin_subscriber**() **const**
Allows access to the builtin *Subscriber*.

Return 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 required to be reversible.

Return RETURN_OK code if everything correct, error code otherwise

Parameters

- `handle`: Identifier of the remote participant to ignore

ReturnCode_t **ignore_topic**(**const** InstanceHandle_t &*handle*)
Locally ignore a topic.

Note This action is not required to be reversible.

Return RETURN_OK code if everything correct, error code otherwise

Parameters

- `handle`: Identifier of the topic to ignore

ReturnCode_t **ignore_publication**(**const** InstanceHandle_t &*handle*)
Locally ignore a datawriter.

Note This action is not required to be reversible.

Return RETURN_OK code if everything correct, error code otherwise

Parameters

- `handle`: Identifier of the datawriter to ignore

ReturnCode_t **ignore_subscription**(**const** InstanceHandle_t &*handle*)
Locally ignore a datareader.

Note This action is not required to be reversible.

Return RETURN_OK code if everything correct, error code otherwise

Parameters

- `handle`: Identifier of the datareader to ignore

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.

Return The Participant's domain_id

ReturnCode_t **`delete_contained_entities`**`()`

Deletes all the entities that were created by means of the "create" methods

Return 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.

Return 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.

Return RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *PublisherQos* to be set

`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.

Return 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.

Return RETCODE_OK

Parameters

- `qos`: *PublisherQos* reference where the default_publisher_qos is returned

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.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *Publisher* profile name.
- `qos`: *PublisherQos* object where the qos is returned.

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.

Return RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *SubscriberQos* to be set

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.

Return 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.

Return RETCODE_OK

Parameters

- `qos`: *SubscriberQos* reference where the default_subscriber_qos is returned

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.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *Subscriber* profile name.
- `qos`: *SubscriberQos* object where the qos is returned.

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.

Return RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *TopicQos* to be set

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.

Return 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.

Return RETCODE_OK

Parameters

- `qos`: *TopicQos* reference where the default_topic_qos is returned

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.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *Topic* profile name.
- `qos`: *TopicQos* object where the qos is returned.

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".

Return RETCODE_OK if everything correct, error code otherwise

Parameters

- `[out] participant_handles`: Reference to the vector where discovered participants will be returned

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.

Return RETCODE_OK if everything correct, PRECONDITION_NOT_MET if participant does not exist

Parameters

- `[out] participant_data`: Reference to the ParticipantBuiltinTopicData object to return the data
- `participant_handle`: InstanceHandle of *DomainParticipant* to retrieve the data from

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".

Return RETCODE_OK if everything correct, error code otherwise

Parameters

- `[out] topic_handles`: Reference to the vector where discovered topics will be returned

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.

Return RETCODE_OK if everything correct, PRECONDITION_NOT_MET if topic does not exist

Parameters

- `[out] topic_data`: Reference to the TopicBuiltinTopicData object to return the data
- `topic_handle`: InstanceHandle of *Topic* to retrieve the data from

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*.

Return True if entity is contained. False otherwise.

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)

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.

Return RETCODE_OK

Parameters

- `current_time`: Time_t reference where the current time is returned

ReturnCode_t **register_type**(*TypeSupport type*, **const** std::string &*type_name*)
Register a type in this participant.

Return 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.

Parameters

- `type`: *TypeSupport*.
- `type_name`: The name that will be used to identify the Type.

ReturnCode_t **register_type**(*TypeSupport type*)
Register a type in this participant.

Return 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.

Parameters

- `type`: *TypeSupport*.

ReturnCode_t **unregister_type**(**const** std::string &*typeName*)
Unregister a type in this participant.

Return 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.

Parameters

- `typeName`: Name of the type

TypeSupport **find_type**(**const** std::string &*type_name*) **const**
This method gives access to a registered type based on its name.

Return *TypeSupport* corresponding to the type_name

Parameters

- `type_name`: Name of the type

const InstanceHandle_t &**get_instance_handle**() **const**
Returns the *DomainParticipant*'s handle.

Return InstanceHandle of this *DomainParticipant*.

const fastrtps::rtps::*GUID_t* &**guid**() **const**
Getter for the Participant GUID.

Return A reference to the GUID

std::vector<std::string> **get_participant_names**() **const**
Getter for the participant names.

Return 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.

Return True if correctly found and activated.

Parameters

- `partguid`: Participant GUID_t.
- `userId`: User defined ID as shown in the XML file.
- `kind`: EndpointKind (WRITER or READER)

fastrtps::rtps::ResourceEvent &**get_resource_event**() **const**

Getter for the resource event.

Return 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.

Return SampleIdentity

Parameters

- `in`: TypeIdentifier sequence

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.

Return SampleIdentity

Parameters

- `in`: TypeIdentifier sequence

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 typelookup service and registers the resulting dynamic type. The registration will be perform asynchronously and the user will be notified through the given callback, which receives the type_name as unique argument. If the type is already registered, the function will return true, but the callback will not be called. If the given type_information is enough to build the type without using the typelookup service, it will return true and the callback will be never called.

Return true if type is already available (callback will not be called). false if type isn't available yet (the callback will be called if negotiation is success, and ignored in other case).

Parameters

- `type_information`:
- `type_name`:
- `callback`:

bool **has_active_entities**()
Check if the Participant has any *Publisher*, *Subscriber* or *Topic*.

Return true if any, false otherwise.

DomainParticipantFactory

class eprosima::fastdds::dds::**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.

Return *DomainParticipant* pointer. (nullptr if not created.)

Parameters

- `domain_id`: Domain Id.
- `qos`: *DomainParticipantQos* Reference.
- `listener`: *DomainParticipantListener* Pointer (default: nullptr)
- `mask`: *StatusMask* Reference (default: all)

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.

Return *DomainParticipant* pointer. (nullptr if not created.)

Parameters

- `domain_id`: Domain Id.
- `profile_name`: Participant profile name.
- `listener`: *DomainParticipantListener* Pointer (default: nullptr)
- `mask`: *StatusMask* Reference (default: all)

DomainParticipant ***create_participant_with_profile**(**const** std::string *&profile_name*, *DomainParticipantListener* **listener* = nullptr, **const** *StatusMask* *&mask* = *StatusMask*::*all*())
Create a Participant.

Return *DomainParticipant* pointer. (nullptr if not created.)

Parameters

- `profile_name`: Participant profile name.
- `listener`: *DomainParticipantListener* Pointer (default: nullptr)

- `mask`: *StatusMask* Reference (default: all)

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.

Return previously created *DomainParticipant* within the specified domain

Parameters

- `domain_id`:

std::vector<*DomainParticipant**> **lookup_participants**(DomainId_t *domain_id*) **const**

Returns all participants that belongs to the specified domain_id.

Return previously created DomainParticipants within the specified domain

Parameters

- `domain_id`:

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.

Return RETCODE_OK

Parameters

- `qos`: *DomainParticipantQos* where the qos is returned

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.

Return 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.

Return RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *DomainParticipantQos* to be set

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.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *DomainParticipant* profile name.
- `qos`: *DomainParticipantQos* object where the qos is returned.

ReturnCode_t **delete_participant**(*DomainParticipant* **part*)

Remove a Participant and all associated publishers and subscribers.

Return RETCODE_PRECONDITION_NOT_MET if the participant has active entities, RETCODE_OK if the participant is correctly deleted and RETCODE_ERROR otherwise.

Parameters

- `part`: Pointer to the participant.

ReturnCode_t **load_profiles**()

Load profiles from default XML file.

Return RETCODE_OK

ReturnCode_t **load_XML_profiles_file**(**const** std::string &*xml_profile_file*)

Load profiles from XML file.

Return RETCODE_OK if it is correctly loaded, RETCODE_ERROR otherwise.

Parameters

- `xml_profile_file`: XML profile file.

ReturnCode_t **get_qos**(*DomainParticipantFactoryQos* &*qos*) **const**

This operation returns the value of the *DomainParticipantFactory* QoS policies.

Return RETCODE_OK

Parameters

- `qos`: DomaParticipantFactoryQos reference where the qos is returned

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.

Return 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.

Parameters

- `qos`: *DomainParticipantFactoryQos* to be set.

Public Static Functions

DomainParticipantFactory ***get_instance**()
Returns the *DomainParticipantFactory* singleton.

Return The *DomainParticipantFactory* singleton.

DomainParticipantFactoryQos

class eprosima::fastdds::dds::**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

DomainParticipantFactoryQos()
Constructor.

~DomainParticipantFactoryQos()
Destructor.

const *EntityFactoryQosPolicy* &**entity_factory**() **const**
Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

EntityFactoryQosPolicy &**entity_factory**()
Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

void **entity_factory**(**const** *EntityFactoryQosPolicy* &*entity_factory*)
Setter for *EntityFactoryQosPolicy*

Parameters

- entity_factory: *EntityFactoryQosPolicy*

DomainParticipantListener

class eprosima::fastdds::dds::**DomainParticipantListener** : **public** eprosima::fastdds::dds::*PublisherListener*
Class *DomainParticipantListener*, overrides behaviour towards certain events.

Public Functions

DomainParticipantListener()
Constructor.

~DomainParticipantListener()
Destructor.

void **on_participant_discovery**(*DomainParticipant* *&participant*, fastrtps::rtps::*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`: Pointer to the Participant which discovered the remote participant.
- `info`: Remote participant information. User can take ownership of the object.

void **onParticipantAuthentication**(*DomainParticipant* *participant, fastrtps::rtps::*ParticipantAuthenticationInfo* &&*info*)

This method is called when a new Participant is authenticated.

Parameters

- `participant`: Pointer to the authenticated Participant.
- `info`: Remote participant authentication information. User can take ownership of the object.

void **on_subscriber_discovery**(*DomainParticipant* *participant, fastrtps::rtps::*ReaderDiscoveryInfo* &&*info*)

This method is called when a new *Subscriber* is discovered, or a previously discovered subscriber changes its QOS or is removed.

Parameters

- `participant`: Pointer to the Participant which discovered the remote subscriber.
- `info`: Remote subscriber information. User can take ownership of the object.

void **on_publisher_discovery**(*DomainParticipant* *participant, fastrtps::rtps::*WriterDiscoveryInfo* &&*info*)

This method is called when a new *Publisher* is discovered, or a previously discovered publisher changes its QOS or is removed.

Parameters

- `participant`: Pointer to the Participant which discovered the remote publisher.
- `info`: Remote publisher information. User can take ownership of the object.

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;

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.

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 eprosima::fastdds::dds::**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 Functions

DomainParticipantQos()

Constructor.

~DomainParticipantQos()

Destructor.

const *UserDataQosPolicy* &**user_data**() **const**

Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

UserDataQosPolicy &**user_data**()

Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

void **user_data**(**const** *UserDataQosPolicy* &*value*)

Setter for *UserDataQosPolicy*

Parameters

- value: *UserDataQosPolicy*

const *EntityFactoryQosPolicy* &**entity_factory**() **const**

Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

EntityFactoryQosPolicy &**entity_factory**()

Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

void **entity_factory**(**const** *EntityFactoryQosPolicy* &*value*)

Setter for *EntityFactoryQosPolicy*

Parameters

- value: *EntityFactoryQosPolicy*

const *ParticipantResourceLimitsQos* &**allocation**() **const**

Getter for ParticipantResourceLimitsQos

Return ParticipantResourceLimitsQos reference

ParticipantResourceLimitsQos &**allocation**()

Getter for ParticipantResourceLimitsQos

Return ParticipantResourceLimitsQos reference

void **allocation**(**const** *ParticipantResourceLimitsQos* &*allocation*)

Setter for ParticipantResourceLimitsQos

Parameters

- allocation: ParticipantResourceLimitsQos

const *PropertyPolicyQos* &**properties**() **const**
Getter for PropertyPolicyQos

Return PropertyPolicyQos reference

PropertyPolicyQos &**properties**()
Getter for PropertyPolicyQos

Return PropertyPolicyQos reference

void **properties**(**const** *PropertyPolicyQos* &*properties*)
Setter for PropertyPolicyQos

Parameters

- `properties`: PropertyPolicyQos

const *WireProtocolConfigQos* &**wire_protocol**() **const**
Getter for *WireProtocolConfigQos*

Return *WireProtocolConfigQos* reference

WireProtocolConfigQos &**wire_protocol**()
Getter for *WireProtocolConfigQos*

Return *WireProtocolConfigQos* reference

void **wire_protocol**(**const** *WireProtocolConfigQos* &*wire_protocol*)
Setter for *WireProtocolConfigQos*

Parameters

- `wire_protocol`: *WireProtocolConfigQos*

const *TransportConfigQos* &**transport**() **const**
Getter for *TransportConfigQos*

Return *TransportConfigQos* reference

TransportConfigQos &**transport**()
Getter for *TransportConfigQos*

Return *TransportConfigQos* reference

void **transport**(**const** *TransportConfigQos* &*transport*)
Setter for *TransportConfigQos*

Parameters

- `transport`: *TransportConfigQos*

const fastrtps::string_255 &**name**() **const**
Getter for the Participant name

Return name

fastrtps::string_255 &**name**()
Getter for the Participant name

Return name

void **name**(**const** fastrtps::string_255 &*value*)
Setter for the Participant name

Return value New name to be set

const *DomainParticipantQos* `eprosima::fastdds::dds::`**PARTICIPANT_QOS_DEFAULT**

Publisher

DataWriter

class eprosima::fastdds::dds::**DataWriter** : **public** eprosima::fastdds::dds::*DomainEntity*

Class *DataWriter*, contains the actual implementation of the behaviour of the *DataWriter*.

Public Types

enum 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

ReturnCode_t **enable**() **override**

This operation enables the *DataWriter*.

Return 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.

Return True if correct, false otherwise

Parameters

- `data`: Pointer to the data

bool **write**(void **data*, fastrtps::rtps::*WriteParams* *&params*)

Write data with params to the topic.

Return True if correct, false otherwise

Parameters

- `data`: Pointer to the data
- `params`: Extra write parameters.

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).

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.

Parameters

- `data`: Pointer to the data
- `handle`: InstanceHandle_t.

ReturnCode_t **write_w_timestamp**(void **data*, **const** InstanceHandle_t &*handle*, **const** fastrtps::rtps::*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.

NOT YET IMPLEMENTED

Return Any of the standard return codes.

Parameters

- `data`: Pointer to the data
- `handle`: InstanceHandle_t
- `timestamp`: Time_t used to set the source_timestamp.

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.

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.

Parameters

- `[in] instance`: Sample used to get the instance's key.

InstanceHandle_t **register_instance_w_timestamp**(void **instance*, **const** fastrtps::rtps::*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*.

NOT YET IMPLEMENTED 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.

Return Handle containing the instance's key.

Parameters

- `instance`: Sample used to get the instance's key.
- `timestamp`: Time_t used to set the source_timestamp.

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.

Return Returns the operation's result. If the operation finishes successfully, ReturnCode_t::RETCODE_OK is returned.

Parameters

- `[in] instance`: Sample used to deduce instance's key in case of `handle` parameter is HANDLE_NIL.
- `[in] handle`: Instance's key to be unregistered.

ReturnCode_t **`unregister_instance_w_timestamp`**(void **instance*, **`const`** InstanceHandle_t &*handle*, **`const`** fastrtps::rtps::*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*.

NOT YET IMPLEMENTED 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

Return Handle containing the instance's key.

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.

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 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.

Return Any of the standard return codes.

Parameters

- `[inout] key_holder`:

- `[in] handle`:

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.

Return handle of the given instance

Parameters

- `[in] instance`: Data pointer to the sample

const fastrtps::rtps::*GUID_t* &**guid**() **const**
Returns the *DataWriter*'s GUID

Return Reference to the *DataWriter* GUID

InstanceHandle_t **get_instance_handle**() **const**
Returns the *DataWriter*'s InstanceHandle

Return Copy of the *DataWriter* InstanceHandle

TypeSupport **get_type**() **const**
Get data type associated to the *DataWriter*

Return 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.

Return RETCODE_OK if the *DataWriter* receive the acknowledgments before the time expires and RETCODE_ERROR otherwise

Parameters

- `max_wait`: Maximum blocking time for this operation

ReturnCode_t **get_offered_deadline_missed_status**(*OfferedDeadlineMissedStatus* &*status*)
Returns the offered deadline missed status.

Return RETCODE_OK

Parameters

- `[out] status`: Deadline missed status struct

ReturnCode_t **get_offered_incompatible_qos_status**(*OfferedIncompatibleQosStatus* &*status*)
Returns the offered incompatible qos status.

Return RETCODE_OK

Parameters

- `[out] status`: Offered incompatible qos status struct

ReturnCode_t **get_publication_matched_status**(*PublicationMatchedStatus* &*status*) **const**
Returns the publication matched status.

Return RETCODE_OK

Parameters

- `[out] status`: publication matched status struct

ReturnCode_t **set_qos**(**const** *DataWriterQos* &*qos*)
Establishes the *DataWriterQos* for this *DataWriter*.

Return 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.

Parameters

- `qos`: *DataWriterQos* to be set

const *DataWriterQos* &**get_qos**() **const**
Retrieves the *DataWriterQos* for this *DataWriter*.

Return Reference to the current *DataWriterQos*

ReturnCode_t **get_qos**(*DataWriterQos* &*qos*) **const**
Fills the *DataWriterQos* with the values of this *DataWriter*.

Return RETCODE_OK

Parameters

- `qos`: *DataWriterQos* object where the qos is returned.

Topic ***get_topic**() **const**
Retrieves the topic for this *DataWriter*.

Return Pointer to the associated *Topic*

const *DataWriterListener* ***get_listener**() **const**
Retrieves the listener for this *DataWriter*.

Return Pointer to the *DataWriterListener*

ReturnCode_t **set_listener**(*DataWriterListener* **listener*)
Modifies the *DataWriterListener*, sets the mask to *StatusMask::all()*

Return RETCODE_OK

Parameters

- `listener`: new value for the *DataWriterListener*

ReturnCode_t **set_listener**(*DataWriterListener* **listener*, **const** *StatusMask* &*mask*)
Modifies the *DataWriterListener*.

Return RETCODE_OK

Parameters

- `listener`: new value for the *DataWriterListener*
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all).

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.

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.

Parameters

- `[in] data`: Sample used to deduce instance's key in case of `handle` parameter is HANDLE_NIL.
- `[in] handle`: InstanceHandle of the data

ReturnCode_t **dispose_w_timestamp**(void *_data_, **const** InstanceHandle_t &_handle_)

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.

Return RTPS_DllAPI

Parameters

- `data`: Pointer to the data.
- `handle`: InstanceHandle_t

ReturnCode_t **get_liveliness_lost_status**(*LivelinessLostStatus* &_status_)

Returns the liveliness lost status.

Return RETCODE_OK

Parameters

- `status`: Liveliness lost status struct

const *Publisher* ***get_publisher**() **const**

Getter for the *Publisher* that creates this *DataWriter*.

Return 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.

Return RETCODE_OK if asserted, RETCODE_ERROR otherwise

ReturnCode_t **get_matched_subscription_data**(builtin::SubscriptionBuiltinTopicData *&subscription_data*, **const** fastrtps::rtps::*InstanceHandle_t* *&subscription_handle*) **const**

Retrieves in a subscription associated with the *DataWriter*.

Return RETCODE_OK

Parameters

- `[out] subscription_data`: subscription data struct
- `subscription_handle`: InstanceHandle_t of the subscription

ReturnCode_t **get_matched_subscriptions**(std::vector<fastrtps::rtps::*InstanceHandle_t**> *&subscription_handles*) **const**

Fills the given vector with the InstanceHandle_t of matched DataReaders.

Return RETCODE_OK

Parameters

- `[out] subscription_handles`: Vector where the InstanceHandle_t are returned

ReturnCode_t **clear_history**(size_t **removed*)

Clears the *DataWriter* history.

Return RETCODE_OK if the samples are removed and RETCODE_ERROR otherwise

Parameters

- `removed`: size_t pointer to return the size of the data removed

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*.

Return ReturnCode_t::RETCODE_ILLEGAL_OPERATION when the data type does not support loans.

Return ReturnCode_t::RETCODE_NOT_ENABLED if the writer has not been enabled.

Return ReturnCode_t::RETCODE_OUT_OF_RESOURCES if the pool has been exhausted.

Return ReturnCode_t::RETCODE_OK if a pointer to a sample is successfully obtained.

Parameters

- `[out] sample`: Pointer to the sample on the internal pool.

- `[in] initialization`: How to initialize the loaned sample.

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.

Return ReturnCode_t::RETCODE_ILLEGAL_OPERATION when the data type does not support loans.

Return ReturnCode_t::RETCODE_NOT_ENABLED if the writer has not been enabled.

Return ReturnCode_t::RETCODE_BAD_PARAMETER if the pointer does not correspond to a loaned sample.

Return ReturnCode_t::RETCODE_OK if the loan is successfully discarded.

Parameters

- `[inout] sample`: Pointer to the previously loaned sample.

ReturnCode_t **get_sending_locators**(rtps::*LocatorList* &*locators*) **const**
Get the list of locators from which this *DataWriter* may send data.

Return NOT_ENABLED if the reader has not been enabled.

Return OK if a list of locators is returned.

Parameters

- `[out] locators`: LocatorList where the list of locators will be stored.

DataWriterListener

class `eprosima::fastdds::dds::`**DataWriterListener**
Class *DataWriterListener*, allows the end user to implement callbacks triggered by certain events.

Subclassed by *eprosima::fastdds::dds::PublisherListener*

Public Functions

DataWriterListener()
Constructor.

~DataWriterListener()
Destructor.

void **on_publication_matched**(*DataWriter* *writer, **const** *PublicationMatchedStatus* &*info*)
This method is called when the *Publisher* is matched (or unmatched) against an endpoint.

Parameters

- `writer`: Pointer to the associated *Publisher*
- `info`: Information regarding the matched subscriber

void **on_offered_deadline_missed**(*DataWriter* *writer, **const** *OfferedDeadlineMissedStatus* &*status*)
A method called when a deadline is missed

Parameters

- `writer`: Pointer to the associated *Publisher*
- `status`: The deadline missed status

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 *Publisher*
- `status`: The deadline missed status

void **on_liveliness_lost**(*DataWriter* **writer*, **const** *LivelinessLostStatus* &*status*)
Method called when the liveliness of a publisher is lost.

Parameters

- `writer`: The publisher
- `status`: The liveliness lost status

DataWriterQos

class `eprosima::fastdds::dds::`**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.

DurabilityQosPolicy &**durability**()
Getter for *DurabilityQosPolicy*

Return *DurabilityQosPolicy* reference

const *DurabilityQosPolicy* &**durability**() **const**
Getter for *DurabilityQosPolicy*

Return *DurabilityQosPolicy* reference

void **durability**(**const** *DurabilityQosPolicy* &*durability*)
Setter for *DurabilityQosPolicy*

Parameters

- `durability`: new value for the *DurabilityQosPolicy*

DurabilityServiceQosPolicy &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* reference

const *DurabilityServiceQosPolicy* &**durability_service**() **const**
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* reference

void **durability_service**(**const** *DurabilityServiceQosPolicy* &*durability_service*)
Setter for *DurabilityServiceQosPolicy*

Parameters

- `durability_service`: new value for the *DurabilityServiceQosPolicy*

DeadlineQosPolicy &**deadline**()
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* reference

const *DeadlineQosPolicy* &**deadline**() **const**
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* reference

void **deadline**(**const** *DeadlineQosPolicy* &*deadline*)
Setter for *DeadlineQosPolicy*

Parameters

- `deadline`: new value for the *DeadlineQosPolicy*

LatencyBudgetQosPolicy &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* reference

const *LatencyBudgetQosPolicy* &**latency_budget**() **const**
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* reference

void **latency_budget**(**const** *LatencyBudgetQosPolicy* &*latency_budget*)
Setter for *LatencyBudgetQosPolicy*

Parameters

- `latency_budget`: new value for the *LatencyBudgetQosPolicy*

LivelinessQosPolicy &**liveliness**()
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* reference

const *LivelinessQosPolicy* &**liveliness**() **const**
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* reference

void **liveliness**(**const** *LivelinessQosPolicy* &*liveliness*)
Setter for *LivelinessQosPolicy*

Parameters

- `liveliness`: new value for the *LivelinessQosPolicy*

ReliabilityQosPolicy &**reliability**()
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* reference

const *ReliabilityQosPolicy* &**reliability**() **const**
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* reference

void **reliability**(**const** *ReliabilityQosPolicy* &*reliability*)
Setter for *ReliabilityQosPolicy*

Parameters

- `reliability`: new value for the *ReliabilityQosPolicy*

DestinationOrderQosPolicy &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* reference

const *DestinationOrderQosPolicy* &**destination_order**() **const**
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* reference

void **destination_order**(**const** *DestinationOrderQosPolicy* &*destination_order*)
Setter for *DestinationOrderQosPolicy*

Parameters

- `destination_order`: new value for the *DestinationOrderQosPolicy*

HistoryQosPolicy &**history**()
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* reference

const *HistoryQosPolicy* &**history**() **const**
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* reference

void **history**(**const** *HistoryQosPolicy* &*history*)
Setter for *HistoryQosPolicy*

Parameters

- `history`: new value for the *HistoryQosPolicy*

ResourceLimitsQosPolicy &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* reference

const *ResourceLimitsQosPolicy* &**resource_limits**() **const**
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* reference

void **resource_limits**(**const** *ResourceLimitsQosPolicy* &*resource_limits*)
Setter for *ResourceLimitsQosPolicy*

Parameters

- `resource_limits`: new value for the *ResourceLimitsQosPolicy*

TransportPriorityQosPolicy &**transport_priority**()
Getter for *TransportPriorityQosPolicy*

Return *TransportPriorityQosPolicy* reference

const *TransportPriorityQosPolicy* &**transport_priority**() **const**
Getter for *TransportPriorityQosPolicy*

Return *TransportPriorityQosPolicy* reference

void **transport_priority**(**const** *TransportPriorityQosPolicy* &*transport_priority*)
Setter for *TransportPriorityQosPolicy*

Parameters

- `transport_priority`: new value for the *TransportPriorityQosPolicy*

LifespanQosPolicy &**lifespan**()
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* reference

const *LifespanQosPolicy* &**lifespan**() **const**
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* reference

void **lifespan**(**const** *LifespanQosPolicy* &*lifespan*)
Setter for *LifespanQosPolicy*

Parameters

- `lifespan`: new value for the *LifespanQosPolicy*

UserDataQosPolicy &**user_data**()
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

const *UserDataQosPolicy* &**user_data**() **const**
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

void **user_data**(**const** *UserDataQosPolicy* &*user_data*)
Setter for *UserDataQosPolicy*

Parameters

- `user_data`: new value for the *UserDataQosPolicy*

OwnershipQosPolicy &**ownership**()
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* reference

const *OwnershipQosPolicy* &**ownership**() **const**
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* reference

void **ownership**(**const** *OwnershipQosPolicy* &*ownership*)
Setter for *OwnershipQosPolicy*

Parameters

- `ownership`: new value for the *OwnershipQosPolicy*

OwnershipStrengthQosPolicy &**ownership_strength**()
Getter for *OwnershipStrengthQosPolicy*

Return *OwnershipStrengthQosPolicy* reference

const *OwnershipStrengthQosPolicy* &**ownership_strength**() **const**
Getter for *OwnershipStrengthQosPolicy*

Return *OwnershipStrengthQosPolicy* reference

void **ownership_strength**(**const** *OwnershipStrengthQosPolicy* &*ownership_strength*)
Setter for *OwnershipStrengthQosPolicy*

Parameters

- `ownership_strength`: new value for the *OwnershipStrengthQosPolicy*

WriterDataLifecycleQosPolicy &**writer_data_lifecycle**()
Getter for *WriterDataLifecycleQosPolicy*

Return *WriterDataLifecycleQosPolicy* reference

const *WriterDataLifecycleQosPolicy* &**writer_data_lifecycle**() **const**
Getter for *WriterDataLifecycleQosPolicy*

Return *WriterDataLifecycleQosPolicy* reference

void **writer_data_lifecycle**(**const** *WriterDataLifecycleQosPolicy* &*writer_data_lifecycle*)
Setter for *WriterDataLifecycleQosPolicy*

Parameters

- `writer_data_lifecycle`: new value for the *WriterDataLifecycleQosPolicy*

PublishModeQosPolicy &**publish_mode**()
Getter for *PublishModeQosPolicy*

Return *PublishModeQosPolicy* reference

const *PublishModeQosPolicy* &**publish_mode**() **const**
Getter for *PublishModeQosPolicy*

Return *PublishModeQosPolicy* reference

void **publish_mode**(**const** *PublishModeQosPolicy* &*publish_mode*)
Setter for *PublishModeQosPolicy*

Parameters

- `publish_mode`: new value for the *PublishModeQosPolicy*

DataRepresentationQosPolicy &**representation**()
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

const *DataRepresentationQosPolicy* &**representation**() **const**
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

void **representation**(**const** *DataRepresentationQosPolicy* &*representation*)
Setter for *DataRepresentationQosPolicy*

Parameters

- `representation`: new value for the *DataRepresentationQosPolicy*

PropertyPolicyQos &**properties**()
Getter for PropertyPolicyQos

Return PropertyPolicyQos reference

const *PropertyPolicyQos* &**properties**() **const**
Getter for PropertyPolicyQos

Return PropertyPolicyQos reference

void **properties**(**const** *PropertyPolicyQos* &*properties*)
Setter for PropertyPolicyQos

Parameters

- `properties`: new value for the PropertyPolicyQos

RTPSReliableWriterQos &**reliable_writer_qos**()
Getter for *RTPSReliableWriterQos*

Return *RTPSReliableWriterQos* reference

const *RTPSReliableWriterQos* &**reliable_writer_qos**() **const**
Getter for *RTPSReliableWriterQos*

Return *RTPSReliableWriterQos* reference

void **reliable_writer_qos**(**const** *RTPSReliableWriterQos* &*reliable_writer_qos*)
Setter for *RTPSReliableWriterQos*

Parameters

- `reliable_writer_qos`: new value for the *RTPSReliableWriterQos*

RTPSEndpointQos &**endpoint**()
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* reference

const *RTPSEndpointQos* &**endpoint**() **const**
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* reference

void **endpoint**(**const** *RTPSEndpointQos* &*endpoint*)
Setter for *RTPSEndpointQos*

Parameters

- `endpoint`: new value for the *RTPSEndpointQos*

WriterResourceLimitsQos &**writer_resource_limits**()
Getter for *WriterResourceLimitsQos*

Return *WriterResourceLimitsQos* reference

const *WriterResourceLimitsQos* &**writer_resource_limits**() **const**
Getter for *WriterResourceLimitsQos*

Return *WriterResourceLimitsQos* reference

void **writer_resource_limits**(**const** *WriterResourceLimitsQos* &*writer_resource_limits*)
Setter for *WriterResourceLimitsQos*

Parameters

- `writer_resource_limits`: new value for the *WriterResourceLimitsQos*

fastrtps::rtps::*ThroughputControllerDescriptor* &**throughput_controller**()
Getter for ThroughputControllerDescriptor

Return ThroughputControllerDescriptor reference

const fastrtps::rtps::*ThroughputControllerDescriptor* &**throughput_controller**() **const**
Getter for ThroughputControllerDescriptor

Return ThroughputControllerDescriptor reference

void **throughput_controller**(**const** fastrtps::rtps::*ThroughputControllerDescriptor* *&throughput_controller*)
Setter for ThroughputControllerDescriptor

Parameters

- `throughput_controller`: new value for the ThroughputControllerDescriptor

DataSharingQosPolicy &**data_sharing**()
Getter for *DataSharingQosPolicy*

Return *DataSharingQosPolicy* reference

const *DataSharingQosPolicy* &**data_sharing**() **const**
Getter for *DataSharingQosPolicy*

Return *DataSharingQosPolicy* reference

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 `eprosima::fastdds::dds::`**Publisher** : **public** eprosima::fastdds::dds::*DomainEntity*
Class *Publisher*, used to send data to associated subscribers.

Public Functions

~Publisher()
Destructor.

ReturnCode_t **enable**() **override**
This operation enables the *Publisher*.

Return 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.

Return *PublisherQos* reference

ReturnCode_t **get_qos**(*PublisherQos* *&qos*) **const**
Retrieves the *Publisher* Qos.

Return RETCODE_OK

ReturnCode_t **set_qos**(**const** *PublisherQos* *&qos*)
Allows modifying the *Publisher* Qos. The given Qos must be supported by the *PublisherQos*.

Return 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.

Parameters

- `qos`: *PublisherQos* to be set

const *PublisherListener* ***get_listener**() **const**
Retrieves the attached *PublisherListener*.

Return *PublisherListener* pointer

ReturnCode_t **set_listener**(*PublisherListener* *listener)
Modifies the *PublisherListener*, sets the mask to *StatusMask::all()*

Return RETCODE_OK

Parameters

- `listener`: new value for the *PublisherListener*

ReturnCode_t **set_listener**(*PublisherListener* *listener, **const** *StatusMask* &mask)
Modifies the *PublisherListener*.

Return RETCODE_OK

Parameters

- `listener`: new value for the *PublisherListener*
- `mask`: *StatusMask* that holds statuses the listener responds to

DataWriter ***create_datawriter**(*Topic* *topic, **const** *DataWriterQos* &qos, *DataWriterListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())
This operation creates a *DataWriter*. The returned *DataWriter* will be attached and belongs to the *Publisher*.

Return Pointer to the created *DataWriter*. nullptr if failed.

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).

DataWriter ***create_datawriter_with_profile**(*Topic* *topic, **const** std::string &profile_name, *DataWriterListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())
This operation creates a *DataWriter*. The returned *DataWriter* will be attached and belongs to the *Publisher*.

Return Pointer to the created *DataWriter*. nullptr if failed.

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).

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.

Return RETCODE_PRECONDITION_NOT_MET if it does not belong to this *Publisher*, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Parameters

- `writer`: *DataWriter* to delete

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.

Return Pointer to a previously created *DataWriter* associated to a *Topic* with the requested topic_name

Parameters

- `topic_name`: Name of the *Topic*

ReturnCode_t **suspend_publications**()
Indicates to FastDDS that the contained DataWriters are about to be modified.

Return RETCODE_OK if successful, an error code otherwise

ReturnCode_t **resume_publications**()
Indicates to FastDDS that the modifications to the DataWriters are complete.

Return 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.

Return RETCODE_OK if successful, an error code otherwise

ReturnCode_t **end_coherent_changes**()
Signals the end of a set of coherent cache changes.

Return 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.

Return 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.

Parameters

- `max_wait`: Maximum blocking time for this operation

const *DomainParticipant* ***get_participant**() **const**
This operation returns the *DomainParticipant* to which the *Publisher* belongs.

Return Pointer to the *DomainParticipant*

ReturnCode_t **delete_contained_entities**()
Deletes all contained DataWriters.

Return 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.

Return RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *DataWriterQos* to be set

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.

Return 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.

Return RETCODE_OK

Parameters

- `qos`: Reference to the current default WriterQos.

ReturnCode_t **copy_from_topic_qos**(fastdds::dds::*DataWriterQos* &*writer_qos*, **const** fastdds::dds::*TopicQos* &*topic_qos*) **const**
Copies *TopicQos* into the corresponding *DataWriterQos*.

Return RETCODE_OK if successful, an error code otherwise

Parameters

- `[out] writer_qos`:
- `[in] topic_qos`:

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.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *DataWriter* profile name.
- `qos`: *DataWriterQos* object where the qos is returned.

const InstanceHandle_t &**get_instance_handle**() **const**

Returns the *Publisher*'s handle.

Return InstanceHandle of this *Publisher*.

bool **get_datawriters**(std::vector<*DataWriter**> &*writers*) **const**

Fills the given vector with all the datawriters of this publisher.

Return true

Parameters

- `writers`: Vector where the DataWriters are returned

bool **has_datawriters**() **const**

This operation checks if the publisher has DataWriters

Return true if the publisher has one or several DataWriters, false otherwise

PublisherListener

class `eprosima::fastdds::dds::`**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

PublisherListener()

Constructor.

~PublisherListener()

Destructor.

PublisherQos

class eprosima::fastdds::dds::**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

PublisherQos()

Constructor.

~PublisherQos() = default

Destructor.

const *PresentationQosPolicy* &**presentation**() **const**

Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

PresentationQosPolicy &**presentation**()

Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

void **presentation**(**const** *PresentationQosPolicy* &*presentation*)

Setter for *PresentationQosPolicy*

Parameters

- presentation: *PresentationQosPolicy*

const *PartitionQosPolicy* &**partition**() **const**

Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

PartitionQosPolicy &**partition**()

Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

void **partition**(**const** *PartitionQosPolicy* &*partition*)

Setter for *PartitionQosPolicy*

Parameters

- partition: *PartitionQosPolicy*

const *GroupDataQosPolicy* &**group_data**() **const**

Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

GroupDataQosPolicy &**group_data**()

Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

void **group_data**(**const** *GroupDataQosPolicy* &*group_data*)

Setter for *GroupDataQosPolicy*

Parameters

- `group_data`: *GroupDataQosPolicy*

const *EntityFactoryQosPolicy* &**entity_factory**() **const**
Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

EntityFactoryQosPolicy &**entity_factory**()
Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

void **entity_factory**(**const** *EntityFactoryQosPolicy* &*entity_factory*)
Setter for *EntityFactoryQosPolicy*

Parameters

- `entity_factory`: *EntityFactoryQosPolicy*

const *PublisherQos* eprosima::fastdds::dds::**PUBLISHER_QOS_DEFAULT**

RTPSReliableWriterQos

class eprosima::fastdds::dds::**RTPSReliableWriterQos**
Qos Policy to configure the DisablePositiveACKsQos and the writer timing attributes.

Public Functions

RTPSReliableWriterQos()
Constructor.

~RTPSReliableWriterQos() = default
Destructor.

Public Members

fastrtps::rtps::*WriterTimes* **times**
Writer Timing Attributes.

DisablePositiveACKsQosPolicy **disable_positive_acks**
Disable positive acks QoS, implemented in the library.

Subscriber

DataReader

class eprosima::fastdds::dds::**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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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.

ReturnCode_t **`read_w_condition`**(*LoanableCollection &data_values*, SampleInfoSeq *&sample_infos*, int32_t *max_samples* = LENGTH_UNLIMITED, ReadCondition **a_condition* = nullptr)

NOT YET IMPLEMENTED This operation accesses via 'read' the samples that match the criteria speci-

fied 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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] sample_infos`: A SampleInfoSeq object where the received sample info will be returned.
- `[in] max_samples`: The maximum number of samples to be returned.
- `[in] a_condition`: A ReadCondition that returned `sample_states` must pass

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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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.

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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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.

ReturnCode_t **`read_next_instance_w_condition`**(*LoanableCollection &data_values*, SampleInfoSeq &*sample_infos*, int32_t *max_samples* = LENGTH_UNLIMITED, **`const`** InstanceHandle_t &*previous_handle* = HANDLE_NIL, ReadCondition **a_condition* = nullptr)

NOT YET IMPLEMENTED 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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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] a_condition`: A ReadCondition that returned `sample_states` must pass

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

Return Any of the standard return codes.

Parameters

- `[out] data`: Data pointer to store the sample
- `[out] info`: *SampleInfo* pointer to store the sample information

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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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.

ReturnCode_t **`take_w_condition`**(*LoanableCollection &data_values*, SampleInfoSeq *&sample_infos*, int32_t *max_samples* = LENGTH_UNLIMITED, ReadCondition **a_condition* = nullptr)

NOT YET IMPLEMENTED 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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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.
- `[in] a_condition`: A ReadCondition that returned `sample_states` must pass

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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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.

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.

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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.

ReturnCode_t **take_next_instance_w_condition**(*LoanableCollection &data_values*, SampleInfoSeq &*sample_infos*, int32_t *max_samples* = LENGTH_UNLIMITED, **const** InstanceHandle_t &*previous_handle* = HANDLE_NIL, ReadCondition **a_condition* = nullptr)

NOT YET IMPLEMENTED 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

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples will be returned.
- `[inout] 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] a_condition`: A ReadCondition that returned `sample_states` must pass

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.

Return Any of the standard return codes.

Parameters

- `[out] data`: Data pointer to store the sample
- `[out] info`: *SampleInfo* pointer to store the sample information

Public Functions

`~DataReader`()

Destructor.

ReturnCode_t **`enable`**() **`override`**

This operation enables the *DataReader*.

Return 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.

Return true if there is new unread message, false if timeout

Parameters

- `[in] timeout`: Max blocking time for this operation.

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.

Return RETCODE_OK if there is new unread message, ReturnCode_t::RETCODE_TIMEOUT if timeout

Parameters

- `[in] max_wait`: Max blocking time for this operation.

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` .

Return Any of the standard return codes.

Parameters

- `[inout] data_values`: A *LoanableCollection* object where the received data samples were obtained from an earlier invocation of read or take on this *DataReader*.
- `[inout] sample_infos`: A SampleInfoSeq object where the received sample infos were obtained from an earlier invocation of read or take on this *DataReader*.

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.

Return Any of the standard return codes.

Parameters

- `[inout] key_holder`:
- `[in] handle`:

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.

Return handle of the given instance

Parameters

- `[in] instance`: Data pointer to the sample

ReturnCode_t **get_first_untaken_info**(*SampleInfo* **info*)

Returns information about the first untaken sample.

Return RETCODE_OK if sample info was returned. RETCODE_NO_DATA if there is no sample to take.

Parameters

- `[out] info`: Pointer to a SampleInfo_t structure to store first untaken sample information.

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.

Return the number of samples on the reader history that have never been read.

const fastrtps::rtps::*GUID_t* &**guid**()

Get associated GUID.

Return Associated GUID

InstanceHandle_t **get_instance_handle**() **const**

Getter for the associated InstanceHandle.

Return Copy of the InstanceHandle

TypeSupport **type**()

Getter for the data type.

Return *TypeSupport* associated to the *DataReader*.

const *TopicDescription* ***get_topicdescription**() **const**

Get *TopicDescription*.

Return *TopicDescription* pointer.

ReturnCode_t **get_requested_deadline_missed_status**(*RequestedDeadlineMissedStatus* &*status*)

Get the requested deadline missed status.

Return The deadline missed status.

ReturnCode_t **get_requested_incompatible_qos_status**(*RequestedIncompatibleQosStatus &status*)

Get the requested incompatible qos status.

Return RETCODE_OK

Parameters

- `[out] status`: Requested incompatible qos status.

ReturnCode_t **set_qos**(**const** *DataReaderQos &qos*)

Setter for the *DataReaderQos*.

Return 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.

Parameters

- `[in] qos`: new value for the *DataReaderQos*.

const *DataReaderQos* &**get_qos**() **const**

Getter for the *DataReaderQos*.

Return Pointer to the *DataReaderQos*.

ReturnCode_t **get_qos**(*DataReaderQos &qos*) **const**

Getter for the *DataReaderQos*.

Return RETCODE_OK

Parameters

- `[in] qos`: *DataReaderQos* where the qos is returned.

ReturnCode_t **set_listener**(*DataReaderListener *listener*)

Modifies the *DataReaderListener*, sets the mask to *StatusMask::all()*.

Return RETCODE_OK

Parameters

- `[in] listener`: new value for the *DataReaderListener*.

ReturnCode_t **set_listener**(*DataReaderListener *listener*, **const** *StatusMask &mask*)

Modifies the *DataReaderListener*.

Return RETCODE_OK

Parameters

- `[in] listener`: new value for the *DataReaderListener*.
- `[in] mask`: *StatusMask* that holds statuses the listener responds to (default: all).

const *DataReaderListener* ***get_listener**() **const**

Getter for the *DataReaderListener*.

Return Pointer to the *DataReaderListener*

ReturnCode_t **get_liveliness_changed_status**(*LivelinessChangedStatus* &*status*) **const**
Get the liveliness changed status.

Return RETCODE_OK

Parameters

- `[out] status`: *LivelinessChangedStatus* object where the status is returned.

ReturnCode_t **get_sample_lost_status**(*SampleLostStatus* &*status*) **const**
Get the SAMPLE_LOST communication status.

Return RETCODE_OK

Parameters

- `[out] status`: SampleLostStatus object where the status is returned.

ReturnCode_t **get_sample_rejected_status**(*SampleRejectedStatus* &*status*) **const**
Get the SAMPLE_REJECTED communication status.

Return RETCODE_OK

Parameters

- `[out] status`: *SampleRejectedStatus* object where the status is returned.

ReturnCode_t **get_subscription_matched_status**(*SubscriptionMatchedStatus* &*status*) **const**
Returns the subscription matched status.

Return RETCODE_OK

Parameters

- `[out] status`: subscription matched status struct

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*.

Return RETCODE_OK

Parameters

- `[out] publication_data`: publication data struct
- `publication_handle`: InstanceHandle_t of the publication

ReturnCode_t **get_matched_publications**(std::vector<fastrtps::rtps::*InstanceHandle_t*> &*publication_handles*) **const**
Fills the given vector with the InstanceHandle_t of matched DataReaders.

Return RETCODE_OK

Parameters

- `[out] publication_handles`: Vector where the InstanceHandle_t are returned

ReadCondition *`create_readcondition`(`const` std::vector<*SampleStateKind*> &*sample_states*, `const` std::vector<*ViewStateKind*> &*view_states*, `const` std::vector<*InstanceStateKind*> &*instance_states*)

This operation creates a ReadCondition. The returned ReadCondition will be attached and belong to the *DataReader*.

Return ReadCondition pointer

Parameters

- `sample_states`: Vector of SampleStateKind
- `view_states`: Vector of ViewStateKind
- `instance_states`: Vector of InstanceStateKind

QueryCondition *`create_querycondition`(`const` std::vector<*SampleStateKind*> &*sample_states*, `const` std::vector<*ViewStateKind*> &*view_states*, `const` std::vector<*InstanceStateKind*> &*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*.

Return QueryCondition pointer

Parameters

- `sample_states`: Vector of SampleStateKind
- `view_states`: Vector of ViewStateKind
- `instance_states`: Vector of InstanceStateKind
- `query_expression`: string containing query
- `query_parameters`: Vector of strings containing parameters of query expression

ReturnCode_t `delete_readcondition`(`const` ReadCondition *a_condition*)

This operation deletes a ReadCondition attached to the *DataReader*.

Return RETCODE_OK

Parameters

- `a_condition`: pointer to a ReadCondition belonging to the *DataReader*

`const` *Subscriber* *`get_subscriber`() `const`

Getter for the *Subscriber*.

Return *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.

Return Any of the standard return codes.

bool **is_sample_valid**(**const** void **data*, **const** *SampleInfo* **info*) **const**
Checks whether the sample is still valid or is corrupted

Return true if the sample is valid

Parameters

- `data`: Pointer to the sample data to check
- `info`: Pointer to the *SampleInfo* related to `data`

ReturnCode_t **get_listening_locators**(rtps::*LocatorList* &*locators*) **const**
Get the list of locators on which this *DataReader* is listening.

Return NOT_ENABLED if the reader has not been enabled.

Return OK if a list of locators is returned.

Parameters

- `[out] locators`: LocatorList where the list of locators will be stored.

DataReaderListener

class `eprosima::fastdds::dds::`**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

DataReaderListener()
Constructor.

~DataReaderListener()
Destructor.

void **on_data_available**(*DataReader* **reader*)
Virtual function to be implemented by the user containing the actions to be performed when a new Data Message is received.

Parameters

- `reader`: *DataReader*

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

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

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

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

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

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 `eprosima::fastdds::dds::`**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*

Public Functions

DataReaderQos()
Constructor.

DurabilityQosPolicy &**durability**()
Getter for *DurabilityQosPolicy*

Return *DurabilityQosPolicy* reference

const *DurabilityQosPolicy* &**durability**() **const**
Getter for *DurabilityQosPolicy*

Return *DurabilityQosPolicy* const reference

void **durability**(**const** *DurabilityQosPolicy* &*new_value*)
Setter for *DurabilityQosPolicy*

Parameters

- `new_value`: new value for the *DurabilityQosPolicy*

DeadlineQosPolicy &**deadline**()
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* reference

const *DeadlineQosPolicy* &**deadline**() **const**
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* const reference

void **deadline**(**const** *DeadlineQosPolicy* &*new_value*)
Setter for *DeadlineQosPolicy*

Parameters

- `new_value`: new value for the *DeadlineQosPolicy*

LatencyBudgetQosPolicy &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* reference

const *LatencyBudgetQosPolicy* &**latency_budget**() **const**
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* const reference

void **latency_budget**(**const** *LatencyBudgetQosPolicy* &*new_value*)
Setter for *LatencyBudgetQosPolicy*

Parameters

- `new_value`: new value for the *LatencyBudgetQosPolicy*

LivelinessQosPolicy &**liveliness**()
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* reference

const *LivelinessQosPolicy* &**liveliness**() **const**
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* const reference

void **liveliness**(**const** *LivelinessQosPolicy* &*new_value*)
Setter for *LivelinessQosPolicy*

Parameters

- `new_value`: new value for the *LivelinessQosPolicy*

ReliabilityQosPolicy &**reliability**()
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* reference

const *ReliabilityQosPolicy* &**reliability**() **const**
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* const reference

void **reliability**(**const** *ReliabilityQosPolicy* &*new_value*)
Setter for *ReliabilityQosPolicy*

Parameters

- `new_value`: new value for the *ReliabilityQosPolicy*

DestinationOrderQosPolicy &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* reference

const *DestinationOrderQosPolicy* &**destination_order**() **const**
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* const reference

void **destination_order**(**const** *DestinationOrderQosPolicy* &*new_value*)
Setter for *DestinationOrderQosPolicy*

Parameters

- `new_value`: new value for the *DestinationOrderQosPolicy*

HistoryQosPolicy &**history**()
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* reference

const *HistoryQosPolicy* &**history**() **const**
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* const reference

void **history**(**const** *HistoryQosPolicy* &*new_value*)
Setter for *HistoryQosPolicy*

Parameters

- `new_value`: new value for the *HistoryQosPolicy*

ResourceLimitsQosPolicy &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* reference

const *ResourceLimitsQosPolicy* &**resource_limits**() **const**
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* const reference

void **resource_limits**(**const** *ResourceLimitsQosPolicy* &*new_value*)
Setter for *ResourceLimitsQosPolicy*

Parameters

- `new_value`: new value for the *ResourceLimitsQosPolicy*

UserDataQosPolicy &**user_data**()
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

const *UserDataQosPolicy* &**user_data**() **const**
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* const reference

void **user_data**(**const** *UserDataQosPolicy* &*new_value*)
Setter for *UserDataQosPolicy*

Parameters

- `new_value`: new value for the *UserDataQosPolicy*

OwnershipQosPolicy &**ownership**()
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* reference

const *OwnershipQosPolicy* &**ownership**() **const**
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* const reference

void **ownership**(**const** *OwnershipQosPolicy* &*new_value*)
Setter for *OwnershipQosPolicy*

Parameters

- `new_value`: new value for the *OwnershipQosPolicy*

TimeBasedFilterQosPolicy &**time_based_filter**()
Getter for *TimeBasedFilterQosPolicy*

Return *TimeBasedFilterQosPolicy* reference

const *TimeBasedFilterQosPolicy* &**time_based_filter**() **const**
Getter for *TimeBasedFilterQosPolicy*

Return *TimeBasedFilterQosPolicy* const reference

void **time_based_filter**(**const** *TimeBasedFilterQosPolicy* &*new_value*)
Setter for *TimeBasedFilterQosPolicy*

Parameters

- `new_value`: new value for the *TimeBasedFilterQosPolicy*

ReaderDataLifecycleQosPolicy &**reader_data_lifecycle**()
Getter for *ReaderDataLifecycleQosPolicy*

Return *ReaderDataLifecycleQosPolicy* reference

const *ReaderDataLifecycleQosPolicy* &**reader_data_lifecycle**() **const**
Getter for *ReaderDataLifecycleQosPolicy*

Return *ReaderDataLifecycleQosPolicy* const reference

void **reader_data_lifecycle**(**const** *ReaderDataLifecycleQosPolicy* &*new_value*)
Setter for *ReaderDataLifecycleQosPolicy*

Parameters

- `new_value`: new value for the *ReaderDataLifecycleQosPolicy*

LifespanQosPolicy &**lifespan**()
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* reference

const *LifespanQosPolicy* &**lifespan**() **const**
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* const reference

void **lifespan**(**const** *LifespanQosPolicy* &*new_value*)
Setter for *LifespanQosPolicy*

Parameters

- `new_value`: new value for the *LifespanQosPolicy*

DurabilityServiceQosPolicy &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* reference

const *DurabilityServiceQosPolicy* &**durability_service**() **const**
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* const reference

void **durability_service**(**const** *DurabilityServiceQosPolicy* &*new_value*)
Setter for *DurabilityServiceQosPolicy*

Parameters

- `new_value`: new value for the *DurabilityServiceQosPolicy*

RTPSReliableReaderQos &**reliable_reader_qos**()
Getter for *RTPSReliableReaderQos*

Return *RTPSReliableReaderQos* reference

const *RTPSReliableReaderQos* &**reliable_reader_qos**() **const**
Getter for *RTPSReliableReaderQos*

Return *RTPSReliableReaderQos* const reference

void **reliable_reader_qos**(**const** *RTPSReliableReaderQos* &*new_value*)
Setter for *RTPSReliableReaderQos*

Parameters

- `new_value`: new value for the *RTPSReliableReaderQos*

TypeConsistencyQos &**type_consistency**()
Getter for *TypeConsistencyQos*

Return *TypeConsistencyQos* reference

const *TypeConsistencyQos* &**type_consistency**() **const**
Getter for *TypeConsistencyQos*

Return *TypeConsistencyQos* const reference

void **type_consistency**(**const** *TypeConsistencyQos* &*new_value*)
Setter for *TypeConsistencyQos*

Parameters

- `new_value`: new value for the *TypeConsistencyQos*

bool **expects_inline_qos**() **const**
Getter for expectsInlineQos_

Return expectsInlineQos_

void **expects_inline_qos**(bool *new_value*)
Setter for expectsInlineQos_

Parameters

- `new_value`: new value for the expectsInlineQos_

PropertyPolicyQos &**properties**()
Getter for PropertyPolicyQos

Return PropertyPolicyQos reference

const *PropertyPolicyQos* &**properties**() **const**
Getter for PropertyPolicyQos

Return PropertyPolicyQos const reference

void **properties**(**const** *PropertyPolicyQos* &*new_value*)
Setter for PropertyPolicyQos

Parameters

- `new_value`: new value for the PropertyPolicyQos

RTPSEndpointQos &**endpoint**()
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* reference

const *RTPSEndpointQos* &**endpoint**() **const**
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* const reference

void **endpoint**(**const** *RTPSEndpointQos* &*new_value*)
Setter for *RTPSEndpointQos*

Parameters

- `new_value`: new value for the *RTPSEndpointQos*

ReaderResourceLimitsQos &**reader_resource_limits**()
Getter for *ReaderResourceLimitsQos*

Return *ReaderResourceLimitsQos* reference

const *ReaderResourceLimitsQos* &**reader_resource_limits**() **const**
Getter for *ReaderResourceLimitsQos*

Return *ReaderResourceLimitsQos* const reference

void **reader_resource_limits**(**const** *ReaderResourceLimitsQos* &*new_value*)
Setter for *ReaderResourceLimitsQos*

Parameters

- `new_value`: new value for the *ReaderResourceLimitsQos*

DataSharingQosPolicy &**data_sharing**()
Getter for *DataSharingQosPolicy*

Return *DataSharingQosPolicy* reference

const *DataSharingQosPolicy* &**data_sharing**() **const**
Getter for *DataSharingQosPolicy*

Return *DataSharingQosPolicy* reference

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 = 0x0001 << 0
Instance is currently in existence.

enumerator NOT_ALIVE_DISPOSED_INSTANCE_STATE = 0x0001 << 1
Not alive disposed instance. The instance has been disposed by a *DataWriter*.

enumerator NOT_ALIVE_NO_WRITERS_INSTANCE_STATE = 0x0001 << 2
Not alive no writers for instance. None of the *DataWriter* objects that are currently alive (according to the LIVELINESS QoS) are writing the instance.

ReaderResourceLimitsQos

class eprosima::fastdds::dds::**ReaderResourceLimitsQos**
Qos Policy to configure the limit of the reader resources.

Public Functions

ReaderResourceLimitsQos() = default
Constructor.

~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 eprosima::fastdds::dds::**RTPSReliableReaderQos**
Qos Policy to configure the DisablePositiveACKsQos and the reader attributes.

Public Functions

RTPSReliableReaderQos()
Constructor.

~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 eprosima::fastdds::dds::**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 **absoulte_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 = 0x0001 << 0
  : Sample has been read.

  enumerator NOT_READ_SAMPLE_STATE = 0x0001 << 1
  : Sample has not been read.

Subscriber

class eprosima::fastdds::dds::**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

~Subscriber()
Destructor.

ReturnCode_t **enable**() **override**
This operation enables the *Subscriber*.

Return 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.

Return *SubscriberQos* reference

ReturnCode_t **get_qos**(*SubscriberQos* &*qos*) **const**
Retrieves the *Subscriber* Qos.

Return RETCODE_OK

Parameters

- `qos`: *SubscriberQos* where the qos is returned

ReturnCode_t **set_qos**(**const** *SubscriberQos* &*qos*)
Allows modifying the *Subscriber* Qos. The given Qos must be supported by the *SubscriberQos*.

Return 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.

Parameters

- `qos`: new value for *SubscriberQos*

const *SubscriberListener* ***get_listener**() **const**
Retrieves the attached *SubscriberListener*.

Return Pointer to the *SubscriberListener*

ReturnCode_t **set_listener**(*SubscriberListener* **listener*)
Modifies the *SubscriberListener*, sets the mask to *StatusMask::all()*

Return RETCODE_OK

Parameters

- `listener`: new value for *SubscriberListener*

ReturnCode_t **set_listener**(*SubscriberListener* **listener*, **const** *StatusMask* &*mask*)
Modifies the *SubscriberListener*.

Return RETCODE_OK

Parameters

- `listener`: new value for the *SubscriberListener*
- `mask`: *StatusMask* that holds statuses the listener responds to.

DataReader ***create_datareader**(*TopicDescription* *topic, **const** *DataReaderQos* &reader_qos, *DataReaderListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())

This operation creates a *DataReader*. The returned *DataReader* will be attached and belong to the *Subscriber*.

Return Pointer to the created *DataReader*. nullptr if failed.

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).

DataReader ***create_datareader_with_profile**(*TopicDescription* *topic, **const** std::string &profile_name, *DataReaderListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())

This operation creates a *DataReader*. The returned *DataReader* will be attached and belongs to the *Subscriber*.

Return Pointer to the created *DataReader*. nullptr if failed.

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).

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.

Return RETCODE_PRECONDITION_NOT_MET if the datareader does not belong to this subscriber, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Parameters

- `reader`: *DataReader* to delete

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.

Return Pointer to a previously created *DataReader* created on a *Topic* with that topic_name

Parameters

- `topic_name`: Name of the topic associated to the *DataReader*

ReturnCode_t **get_datareaders**(std::vector<*DataReader**> &readers) **const**

This operation allows the application to access the *DataReader* objects.

Return RETCODE_OK

Parameters

- `readers`: Vector of *DataReader* where the list of existing readers is returned

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.

Return RETCODE_OK

Parameters

- `[out] readers`: 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

bool **has_datareaders**() **const**

This operation checks if the subscriber has DataReaders

Return 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*.

Return RETCODE_OK

ReturnCode_t **end_access**()

Indicates that the application has finished accessing the data samples in *DataReader* objects managed by the *Subscriber*.

Return 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.

Return 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.

Return 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.

Return RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: new value for *DataReaderQos* to set as default

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.

Return 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.

Return 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.

Return RETCODE_OK

Parameters

- `qos`: *DataReaderQos* where the default_qos is returned

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.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *DataReader* profile name.
- `qos`: *DataReaderQos* object where the qos is returned.

const *DomainParticipant* ***get_participant**() **const**

This operation returns the *DomainParticipant* to which the *Subscriber* belongs.

Return *DomainParticipant* Pointer

const InstanceHandle_t &**get_instance_handle**() **const**
Returns the *Subscriber*'s handle.

Return InstanceHandle of this *Subscriber*.

Public Static Functions

ReturnCode_t **copy_from_topic_qos**(*DataReaderQos* &*reader_qos*, **const** *TopicQos* &*topic_qos*)
Copies *TopicQos* into the corresponding *DataReaderQos*.

Return RETCODE_OK if successful, an error code otherwise

Parameters

- `[inout] reader_qos`:
- `[in] topic_qos`:

SubscriberListener

class `eprosima::fastdds::dds::`**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

SubscriberListener()
Constructor.

~SubscriberListener()
Destructor.

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 `eprosima::fastdds::dds::`**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

SubscriberQos()
Constructor.

~SubscriberQos()
Destructor.

const *PresentationQosPolicy* &**presentation**() **const**
Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

PresentationQosPolicy &**presentation**()
Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

void **presentation**(**const** *PresentationQosPolicy* &*presentation*)
Setter for *PresentationQosPolicy*

Parameters

- `presentation`: new value for the *PresentationQosPolicy*

const *PartitionQosPolicy* &**partition**() **const**
Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

PartitionQosPolicy &**partition**()
Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

void **partition**(**const** *PartitionQosPolicy* &*partition*)
Setter for *PartitionQosPolicy*

Parameters

- `partition`: new value for the *PartitionQosPolicy*

const *GroupDataQosPolicy* &**group_data**() **const**
Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

GroupDataQosPolicy &**group_data**()
Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

void **group_data**(**const** *GroupDataQosPolicy* &*group_data*)
Setter for *GroupDataQosPolicy*

Parameters

- `group_data`: new value for the *GroupDataQosPolicy*

const *EntityFactoryQosPolicy* &**entity_factory**() **const**
Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

EntityFactoryQosPolicy &**entity_factory**()
Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

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 `eprosima::fastdds::dds::`**TypeConsistencyQos** : **public** eprosima::fastdds::dds::*QosPolicy*
Qos Policy to configure the XTypes Qos associated to the *DataReader*.

Public Functions

TypeConsistencyQos()
Constructor.

~TypeConsistencyQos() = default
Destructor.

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 = 0x0001 << 0
New instance.This latest generation of the instance has not previously been accessed.

enumerator NOT_NEW_VIEW_STATE = 0x0001 << 1
Not a new instance. This latest generation of the instance has previously been accessed.

Topic

Topic

class eprosima::fastdds::dds::**Topic** : **public** eprosima::fastdds::dds::*DomainEntity*, **public** eprosima::fastdds::dds
Class *TopicDescription*, represents the fact that both publications and subscriptions are tied to a single data-type

Public Functions

~Topic()
Destructor.

DomainParticipant ***get_participant**() **const override**
Getter for the *DomainParticipant*.

Return *DomainParticipant* pointer

ReturnCode_t **get_inconsistent_topic_status**(*InconsistentTopicStatus* &*status*)
Allows the application to retrieve the INCONSISTENT_TOPIC_STATUS status of a *Topic*.

Return RETCODE_OK

Parameters

- `status`: [out] Status to be retrieved.

const *TopicQos* &**get_qos**() **const**
Allows accessing the *Topic* Qos.

Return reference to *TopicQos*

ReturnCode_t **get_qos**(*TopicQos* &*qos*) **const**
Retrieves the *Topic* Qos.

Return RETCODE_OK

Parameters

- `qos`: *TopicQos* where the qos is returned

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 Value

- `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*.

Return pointer to *TopicListener*

ReturnCode_t **set_listener**(*TopicListener* *listener, **const** *StatusMask* &mask = *StatusMask*::*all*())
Modifies the *TopicListener*.

Return RETCODE_OK

Parameters

- `listener`: new value for the *TopicListener*
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all).

TopicDescriptionImpl ***get_impl**() **const override**
Getter for the TopicDescriptionImpl.

Return pointer to TopicDescriptionImpl

TopicDataType

class `eprosima::fastdds::dds::`**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.

~TopicDataType()
Destructor.

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.

Return True if correct.

Parameters

- `[in] data`: Pointer to the data
- `[out] payload`: Pointer to the payload

bool **deserialize**(fastrtps::rtps::*SerializedPayload_t* *payload, void *data) = 0
Deserialize method, it should be implemented by the user, since it is abstract.

Return True if correct.

Parameters

- `[in] payload`: Pointer to the payload
- `[out] data`: Pointer to the data

std::function<uint32_t()> **getSerializedSizeProvider**
void *data = 0Gets the SerializedSizeProvider function.

Return function

Parameters

- `data`: Pointer

void ***createData**() = 0
Create a Data Type.

Return Void pointer to the created object.

void **deleteData**(void *data) = 0
Remove a previously created object.

Parameters

- `data`: Pointer to the created Data.

bool **getKey**(void *data, fastrtps::rtps::*InstanceHandle_t* *ihandle, bool *force_md5* = false) = 0
Get the key associated with the data.

Return True if correct.

Parameters

- `[in] data`: Pointer to the data.
- `[out] ihandle`: Pointer to the Handle.
- `[in] force_md5`: Force MD5 checking.

void **setName**(**const** char *nam)
Set topic data type name

Parameters

- `nam`: *Topic* data type name

const char ***getName**() **const**
Get topic data type name

Return *Topic* data type name

bool **auto_fill_type_object**() **const**
Get the type object auto-fill configuration

Return true if the type object should be auto-filled

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

bool **auto_fill_type_information**() **const**
Get the type information auto-fill configuration

Return true if the type information should be auto-filled

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

const std::shared_ptr<*TypeIdV1*> **type_identifier**() **const**
Get the type identifier

Return *TypeIdV1*

void **type_identifier**(**const** *TypeIdV1* &*id*)
Set type identifier

Parameters

- `id`: new value for *TypeIdV1*

void **type_identifier**(**const** std::shared_ptr<*TypeIdV1*> *id*)
Set type identifier

Parameters

- `id`: shared pointer to *TypeIdV1*

const std::shared_ptr<*TypeObjectV1*> **type_object**() **const**
Get the type object

Return *TypeObjectV1*

void **type_object**(**const** *TypeObjectV1* &*object*)
Set type object

Parameters

- `object`: new value for *TypeObjectV1*

void **type_object**(std::shared_ptr<*TypeObjectV1*> *object*)
Set type object

Parameters

- `object`: shared pointer to *TypeObjectV1*

const std::shared_ptr<xtypes::*TypeInformation*> **type_information**() **const**
Get the type information

Return TypeInformation

void **type_information**(**const** xtypes::*TypeInformation* &*info*)
Set type information

Parameters

- `info`: new value for TypeInformation

void **type_information**(std::shared_ptr<xtypes::*TypeInformation*> *info*)
Set type information

Parameters

- `info`: shared pointer to TypeInformation

bool **is_bounded**() **const**
Checks if the type is bounded.

bool **is_plain**() **const**
Checks if the type is plain.

bool **construct_sample**(void **memory*) **const**
Construct a sample on a memory location.

Return whether this type supports in-place construction or not.

Parameters

- `memory`: Pointer to the memory location where the sample should be constructed.

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.

TopicDescription

class `eprosima::fastdds::dds::`**TopicDescription**
Class *TopicDescription*, represents the fact that both publications and subscriptions are tied to a single data-type

Subclassed by *eprosima::fastdds::dds::Topic*

Public Functions

DomainParticipant ***get_participant**() **const** = 0
Get the *DomainParticipant* to which the *TopicDescription* belongs.

Return The *DomainParticipant* to which the *TopicDescription* belongs.

const std::string &**get_name**() **const**
Get the name used to create this *TopicDescription*.

Return the name used to create this *TopicDescription*.

const std::string &**get_type_name**() **const**
Get the associated type name.

Return the type name.

TopicDescriptionImpl ***get_impl**() **const** = 0
Get the TopicDescriptionImpl

Return pointer to TopicDescriptionImpl

TopicListener

class eprosima::fastdds::dds::**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

TopicListener()

Constructor.

~TopicListener()

Destructor.

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 eprosima::fastdds::dds::**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.

const *TopicDataQosPolicy* &**topic_data**() **const**

Getter for *TopicDataQosPolicy*

Return TopicDataQos reference

TopicDataQosPolicy &**topic_data**()

Getter for *TopicDataQosPolicy*

Return TopicDataQos reference

void **topic_data**(**const** *TopicDataQosPolicy* &value)

Setter for *TopicDataQosPolicy*

Parameters

- value: new value for the *TopicDataQosPolicy*

const *DurabilityQosPolicy* &**durability**() **const**

Getter for *DurabilityQosPolicy*

Return DurabilityQos reference

DurabilityQosPolicy &**durability**()
Getter for *DurabilityQosPolicy*

Return DurabilityQos reference

void **durability**(**const** *DurabilityQosPolicy* &*durability*)
Setter for *DurabilityQosPolicy*

Parameters

- `durability`: new value for the *DurabilityQosPolicy*

const *DurabilityServiceQosPolicy* &**durability_service**() **const**
Getter for *DurabilityServiceQosPolicy*

Return DurabilityServiceQos reference

DurabilityServiceQosPolicy &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Return DurabilityServiceQos reference

void **durability_service**(**const** *DurabilityServiceQosPolicy* &*durability_service*)
Setter for *DurabilityServiceQosPolicy*

Parameters

- `durability_service`: new value for the *DurabilityServiceQosPolicy*

const *DeadlineQosPolicy* &**deadline**() **const**
Getter for *DeadlineQosPolicy*

Return DeadlineQos reference

DeadlineQosPolicy &**deadline**()
Getter for *DeadlineQosPolicy*

Return DeadlineQos reference

void **deadline**(**const** *DeadlineQosPolicy* &*deadline*)
Setter for *DeadlineQosPolicy*

Parameters

- `deadline`: new value for the *DeadlineQosPolicy*

const *LatencyBudgetQosPolicy* &**latency_budget**() **const**
Getter for *LatencyBudgetQosPolicy*

Return LatencyBudgetQos reference

LatencyBudgetQosPolicy &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Return LatencyBudgetQos reference

void **latency_budget**(**const** *LatencyBudgetQosPolicy* &*latency_budget*)
Setter for *LatencyBudgetQosPolicy*

Parameters

- `latency_budget`: new value for the *LatencyBudgetQosPolicy*

const *LivelinessQosPolicy* &**liveliness**() **const**
Getter for *LivelinessQosPolicy*

Return LivelinessQos reference

LivelinessQosPolicy &**liveliness**()
Getter for *LivelinessQosPolicy*

Return LivelinessQos reference

void **liveliness**(**const** *LivelinessQosPolicy* &*liveliness*)
Setter for *LivelinessQosPolicy*

Parameters

- `liveliness`: new value for the *LivelinessQosPolicy*

const *ReliabilityQosPolicy* &**reliability**() **const**
Getter for *ReliabilityQosPolicy*

Return ReliabilityQos reference

ReliabilityQosPolicy &**reliability**()
Getter for *ReliabilityQosPolicy*

Return ReliabilityQos reference

void **reliability**(**const** *ReliabilityQosPolicy* &*reliability*)
Setter for *ReliabilityQosPolicy*

Parameters

- `reliability`: new value for the *ReliabilityQosPolicy*

const *DestinationOrderQosPolicy* &**destination_order**() **const**
Getter for *DestinationOrderQosPolicy*

Return DestinationOrderQos reference

DestinationOrderQosPolicy &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Return DestinationOrderQos reference

void **destination_order**(**const** *DestinationOrderQosPolicy* &*destination_order*)
Setter for *DestinationOrderQosPolicy*

Parameters

- `destination_order`: new value for the *DestinationOrderQosPolicy*

const *HistoryQosPolicy* &**history**() **const**
Getter for *HistoryQosPolicy*

Return HistoryQos reference

HistoryQosPolicy &**history**()
Getter for *HistoryQosPolicy*

Return HistoryQos reference

void **history**(**const** *HistoryQosPolicy* &*history*)
Setter for *HistoryQosPolicy*

Parameters

- `history`: new value for the *HistoryQosPolicy*

const *ResourceLimitsQosPolicy* &**resource_limits**() **const**
Getter for *ResourceLimitsQosPolicy*

Return ResourceLimitsQos reference

ResourceLimitsQosPolicy &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Return ResourceLimitsQos reference

void **resource_limits**(**const** *ResourceLimitsQosPolicy* &*resource_limits*)
Setter for *ResourceLimitsQosPolicy*

Parameters

- `resource_limits`: new value for the *ResourceLimitsQosPolicy*

const *TransportPriorityQosPolicy* &**transport_priority**() **const**
Getter for *TransportPriorityQosPolicy*

Return TransportPriorityQos reference

TransportPriorityQosPolicy &**transport_priority**()
Getter for *TransportPriorityQosPolicy*

Return TransportPriorityQos reference

void **transport_priority**(**const** *TransportPriorityQosPolicy* &*transport_priority*)
Setter for *TransportPriorityQosPolicy*

Parameters

- `transport_priority`: new value for the *TransportPriorityQosPolicy*

const *LifespanQosPolicy* &**lifespan**() **const**
Getter for *LifespanQosPolicy*

Return LifespanQos reference

LifespanQosPolicy &**lifespan**()
Getter for *LifespanQosPolicy*

Return LifespanQos reference

void **lifespan**(**const** *LifespanQosPolicy* &*lifespan*)
Setter for *LifespanQosPolicy*

Parameters

- `lifespan`: new value for the *LifespanQosPolicy*

const *OwnershipQosPolicy* &**ownership**() **const**
Getter for *OwnershipQosPolicy*

Return OwnershipQos reference

OwnershipQosPolicy &**ownership**()
Getter for *OwnershipQosPolicy*

Return OwnershipQos reference

void **ownership**(**const** *OwnershipQosPolicy* &*ownership*)
Setter for *OwnershipQosPolicy*

Parameters

- `ownership`: new value for the *OwnershipQosPolicy*

const *DataRepresentationQosPolicy* &**representation**() **const**
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

DataRepresentationQosPolicy &**representation**()
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

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 eprosima::fastdds::dds::**TypeIdV1** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosima::fastdds::d
Class *TypeIdV1*

Public Functions

TypeIdV1()
Constructor without parameters.

TypeIdV1(**const** *TypeIdV1* &*type*)
Copy constructor.

Parameters

- `type`: Another instance of *TypeIdV1*

TypeIdV1(**const** fastrtps::types::TypeIdentifier &*identifier*)
Constructor using a TypeIndentifier.

Parameters

- `identifier`: TypeIdentifier to be set

TypeIdV1(*TypeIdV1* &&*type*)
Move constructor.

Parameters

- `type`: Another instance of *TypeIdV1*

~TypeIdV1() **override** = default
Destructor.

void **clear**() **override**
Clears the *QosPolicy* object.

const fastrtps::types::TypeIdentifier &**get**() **const**
Getter for the TypeIndentifier.

Return TypeIdentifier reference

Public Members

fastrtps::types::TypeIdentifier **m_type_identifier**
Type Identifier.

TypeInformation

class eprosima::fastdds::dds::xtypes::**TypeInformation** : **public** eprosima::fastdds::dds::Parameter_t, **publi**
Class *xtypes::TypeInformation*

Public Functions

TypeInformation()
Constructor.

TypeInformation(**const** *TypeInformation* &*type*)
Copy constructor.

Parameters

- `type`: Another instance of *TypeInformation*

TypeInformation(**const** fastrtps::types::TypeInformation &*info*)
Constructor using a fastrtps::types::TypeInformation.

Parameters

- `info`: fastrtps::types::TypeInformation to be set

TypeInformation(*TypeInformation* &&*type*)
Move Constructor.

Parameters

- `type`: Another instance of *TypeInformation*

~TypeInformation() **override** = default
Destructor.

void **clear**() **override**
Clears the *QosPolicy* object.

bool **assigned**() **const**
Check if it is assigned.

Return true if assigned, false if not

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 eprosima::fastdds::dds::**TypeObjectV1** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosima::fast
Class *TypeObjectV1*

Public Functions

TypeObjectV1()
Constructor.

TypeObjectV1(**const** *TypeObjectV1* &*type*)
Copy constructor.

Parameters

- `type`: Another instance of *TypeObjectV1*

TypeObjectV1(**const** fastrtps::types::TypeObject &*type*)
Constructor using a TypeObject.

Parameters

- `type`: TypeObject to be set

TypeObjectV1(*TypeObjectV1* &&*type*)
Move constructor.

Parameters

- `type`: Another instance of *TypeObjectV1*

~TypeObjectV1() **override** = default
Destructor.

void **clear**() **override**
Clears the *QosPolicy* object.

const fastrtps::types::TypeObject &**get**() **const**
Getter for the TypeObject.

Return TypeObject reference

Public Members

fastrtps::types::TypeObject **m_type_object**
Type Object.

TypeSupport

class eprosima::fastdds::dds::**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*

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`:

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`:

ReturnCode_t **register_type** (*DomainParticipant* *participant) **const**
Registers the type on a participant.

Return 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

Parameters

- `participant`: *DomainParticipant* where the type is going to be registered

ReturnCode_t **register_type** (*DomainParticipant* *participant*, std::string *type_name*) **const**
Registers the type on a participant.

Return 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

Parameters

- `participant`: *DomainParticipant* where the type is going to be registered
- `type_name`: Name of the type to register

const std::string &**get_type_name** () **const**
Getter for the type name.

Return name of the data type

bool **serialize** (void *data, fastrtps::rtps::*SerializedPayload_t* *payload)
Serializes the data.

Return true if it is serialized correctly, false if not

Parameters

- `data`: Pointer to data
- `payload`: Pointer to payload

bool **deserialize** (fastrtps::rtps::*SerializedPayload_t* *payload, void *data)
Deserializes the data.

Return true if it is deserialized correctly, false if not

Parameters

- `payload`: Pointer to payload

- `data`: Pointer to data

std::function<uint32_t()> **get_serialized_size_provider**
void **data*Getter for the SerializedSizeProvider.

Return function

Parameters

- `data`: Pointer to data

void ***create_data**()
Creates new data.

Return Pointer to the data

void **delete_data**(void **data*)
Deletes data.

Parameters

- `data`: Pointer to the data to delete

bool **get_key**(void **data*, InstanceHandle_t **i_handle*, bool *force_md5* = false)
Getter for the data key.

Return true if the key is returned, false if not

Parameters

- `data`: Pointer to data
- `i_handle`: InstanceHandle pointer to store the key
- `force_md5`: boolean to force md5 (default: false)

bool **empty**() **const**
Check if the *TypeSupport* is empty.

Return true if empty, false if not

bool **is_bounded**() **const**
Checks if the type is bounded.

bool **is_plain**() **const**
Checks if the type is plain.

6.30.2 RTPS

eProsima Fast DDS Real-Time Publish-Subscribe (RTPS) layer API.

Attributes

BuiltinAttributes

class eprosima::fastrtps::rtps::**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.

LocatorList_t **metatrafficUnicastLocatorList**

Metatraffic Unicast Locator List.

LocatorList_t **metatrafficMulticastLocatorList**

Metatraffic Multicast Locator List.

LocatorList_t **initialPeersList**

Initial peers.

MemoryManagementPolicy_t **readerHistoryMemoryPolicy** = MemoryManagementPolicy_t::PREALLOCATED_WITH

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

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_RTPSParticipantAllocatio**

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 eprosima::fastrtps::rtps::**DiscoverySettings**

Class *DiscoverySettings*, to define the attributes of the several discovery protocols available

Public Functions

const char ***getStaticEndpointXMLFilename**() **const**

Get the static endpoint XML filename

Return Static endpoint XML filename

void **setStaticEndpointXMLFilename**(**const** char **str*)

Set the static endpoint XML filename

Parameters

- `str`: Static endpoint XML filename

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://).

const char ***static_edp_xml_config**() **const**
Get the static endpoint XML configuration.

Return 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 eprosima::fastrtps::rtps::**EndpointAttributes**

Structure *EndpointAttributes*, describing the attributes associated with an RTPS *Endpoint*.

Public Functions

int16_t **getUserDefinedID**() **const**

Get the user defined ID

Return User defined ID

int16_t **getEntityID**() **const**

Get the entity defined ID

Return Entity ID

void **setUserDefinedID**(uint8_t *id*)

Set the user defined ID

Parameters

- id: User defined ID to be set

void **setEntityID**(uint8_t *id*)

Set the entity ID

Parameters

- id: Entity ID to be set

void **set_data_sharing_configuration**(DataSharingQosPolicy *cfg*)

Set the DataSharing configuration

Parameters

- cfg: Configuration to be set

const DataSharingQosPolicy &**data_sharing_configuration**() **const**

Get the DataSharing configuration

Return Configuration of data sharing

Public Members

EndpointKind_t **endpointKind**

Endpoint kind, default value WRITER.

TopicKind_t **topicKind**

Topic kind, default value NO_KEY.

ReliabilityKind_t **reliabilityKind**

Reliability kind, default value BEST_EFFORT.

DurabilityKind_t **durabilityKind**

Durability kind, default value VOLATILE.

GUID_t **persistence_guid**

GUID used for persistence.

LocatorList_t **unicastLocatorList**

Unicast locator list.

LocatorList_t **multicastLocatorList**
Multicast locator list.

LocatorList_t **remoteLocatorList**
Remote locator list.

PropertyPolicy **properties**
Properties.

HistoryAttributes

class eprosima::fastrtps::rtps::**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

HistoryAttributes()
Default constructor.

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.

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 eprosima::fastrtps::rtps::**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 = 0

enumerator FILTER_DIFFERENT_HOST = 0x1

enumerator FILTER_DIFFERENT_PROCESS = 0x2

enumerator FILTER_SAME_PROCESS = 0x4

PropertyPolicy

class eprosima::fastrtps::rtps::**PropertyPolicy**

Public Functions

const *PropertySeq* &**properties**() **const**
Get properties.

PropertySeq &**properties**()
Set properties.

const *BinaryPropertySeq* &**binary_properties**() **const**
Get binary_properties.

BinaryPropertySeq &**binary_properties**()
Set binary_properties.

PropertyPolicyHelper

class eprosima::fastrtps::rtps::**PropertyPolicyHelper**

Public Static Functions

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.

Return A copy of properties whose name starts with the prefix.

Parameters

- property_policy: *PropertyPolicy* where properties will be searched.
- prefix: Prefix used to search properties.

size_t **length**(**const** *PropertyPolicy* &*property_policy*)
Get the length of the property_policy.

std::string ***find_property**(*PropertyPolicy* &*property_policy*, **const** std::string &*name*)
Look for a property_policy by name.

const std::string ***find_property**(**const** *PropertyPolicy* &*property_policy*, **const** std::string &*name*)
Retrieves a property_policy by name.

ReaderAttributes

class eprosima::fastrtps::rtps::**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.

ReaderTimes

class eprosima::fastrtps::rtps::**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 eprosima::fastrtps::rtps::**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 eprosima::fastdds::rtps::**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 eprosima::fastrtps::rtps::**RTPSParticipantAllocationAttributes**
Holds allocation limits affecting collections managed by a participant.

Public Functions

ResourceLimitedContainerConfig **total_readers**() **const**

Return the allocation config for the total of readers in the system (participants * readers)

ResourceLimitedContainerConfig **total_writers**() **const**

Return 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.

RTPSParticipantAttributes

class eprosima::fastrtps::rtps::**RTPSParticipantAttributes**
Class *RTPSParticipantAttributes* used to define different aspects of a *RTPSParticipant*.

Public Functions

void **setName**(**const** char *_nam_)
Set the name of the participant.

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 UnicastLocators. This is usually left empty.

uint32_t **sendSocketBufferSize**
Send socket buffer size for the send resource.

Zero value indicates to use default system buffer size. Default value: 0.

uint32_t **listenSocketBufferSize**
Listen socket buffer for all listen resources.

Zero value indicates to use default system buffer size. Default value: 0.

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**
Participant ID.

ThroughputControllerDescriptor **throughputController**
Throughput controller parameters. Leave default for uncontrolled flow.

std::vector<std::shared_ptr<fastdds::rtps::TransportDescriptorInterface>> **userTransports**
User defined transports to use alongside or in place of builtins.

bool **useBuiltinTransports**
Set as false to disable the default UDPv4 implementation.

RTPSParticipantAllocationAttributes **allocation**
Holds allocation limits affecting collections managed by a participant.

PropertyPolicy **properties**
Property policies.

RTPSWriterPublishMode

enum eprosima::fastrtps::rtps::**RTPSWriterPublishMode**
Values:

enumerator SYNCHRONOUS_WRITER

enumerator ASYNCHRONOUS_WRITER

SendBuffersAllocationAttributes

struct eprosima::fastrtps::rtps::**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 eprosima::fastrtps::rtps::**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 eprosima::fastrtps::rtps::**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 eprosima::fastrtps::rtps::**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 eprosima::fastrtps::rtps::**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.

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.

WriterTimes

struct eprosima::fastrtps::rtps::**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.

Common

BinaryProperty

BinaryProperty

class BinaryProperty

BinaryPropertyHelper

class BinaryPropertyHelper

BinaryPropertySeq

typedef std::vector<*BinaryProperty*> eprosima::fastrtps::rtps::**BinaryPropertySeq**

CacheChange

CacheChange_t

struct eprosima::fastrtps::rtps::**CacheChange_t**

Structure *CacheChange_t*, contains information on a specific CacheChange.

Public Functions

CacheChange_t() = default

Default constructor.

Creates an empty *CacheChange_t*.

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.

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.

Return True if correct.

Parameters

- [in] ch_ptr: Pointer to the change.

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

- `[in] ch_ptr`: Pointer to the change.

uint32_t **getFragmentCount**() **const**
Get the number of fragments this change is split into.

Return number of fragments.

uint16_t **getFragmentSize**() **const**
Get the size of each fragment this change is split into.

Return size of fragment (0 means change is not fragmented).

bool **is_fully_assembled**()
Checks if all fragments have been received.

Return true when change is fully assembled (i.e. no missing fragments).

void **get_missing_fragments**(*FragmentNumberSet_t* &*frag_sns*)
Fills a FragmentNumberSet_t with the list of missing fragments.

Parameters

- `[out] frag_sns`: FragmentNumberSet_t where result is stored.

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.

bool **isRead** = false
Indicates if the cache has been read (only used in READERS)

Time_t **sourceTimestamp**
Source TimeStamp (only used in Readers)

Time_t **receptionTimestamp**
Reception TimeStamp (only used in Readers)

ChangeForReader_t

class eprosima::fastrtps::rtps::**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

CacheChange_t ***getChange**() **const**
Get the cache change

Return Cache change

void **notValid**()
Set change as not valid.

bool **isValid**() **const**
Set change as valid.

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 = 0
UNSENT.

enumerator REQUESTED = 1
REQUESTED.

enumerator UNACKNOWLEDGED = 2
UNACKNOWLEDGED.

enumerator ACKNOWLEDGED = 3
ACKNOWLEDGED.

enumerator UNDERWAY = 4
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 eprosima::fastrtps::rtps::**CDRMessage_t**
Structure *CDRMessage_t*, contains a serialized message.

Public Functions

CDRMessage_t(uint32_t *size*)
Constructor with maximum size

Parameters

- `size`: Maximum size

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
Max size of RTPS message in bytes.

RTPSMESSAGE_COMMON_RTPS_PAYLOAD_SIZE

RTPSMESSAGE_COMMON_DATA_PAYLOAD_SIZE

RTPSMESSAGE_HEADER_SIZE

RTPSMESSAGE_SUBMESSAGEHEADER_SIZE

RTPSMESSAGE_DATA_EXTRA_INLINEQOS_SIZE

RTPSMESSAGE_INFOTS_SIZE

RTPSMESSAGE_OCTETSTOINLINEQOS_DATASUBMSG

RTPSMESSAGE_OCTETSTOINLINEQOS_DATAFRAGSUBMSG

RTPSMESSAGE_DATA_MIN_LENGTH

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_RTPSPartici

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_SPDPWriter** = ENTITYID_SPDP_BUILTIN_RTPSPartici

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_SEDPPubWriter** = ENTITYID_SEDP_BUILTIN_PUBLI

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_SEDPPubReader** = ENTITYID_SEDP_BUILTIN_PUBLI

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_SEDPSubWriter** = ENTITYID_SEDP_BUILTIN_SUBSC

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_SEDPSubReader** = ENTITYID_SEDP_BUILTIN_SUBSC

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_RTPSParticipant** = ENTITYID_RTPSParticipant

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_WriterLiveliness** = ENTITYID_P2P_BUILTIN_RTPS

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_ReaderLiveliness** = ENTITYID_P2P_BUILTIN_RTPS

const *EntityId_t* eprosima::fastrtps::rtps::**participant_stateless_message_writer_entity_id** = ENTI

const *EntityId_t* eprosima::fastrtps::rtps::**participant_stateless_message_reader_entity_id** = ENTI

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_request_writer** = ENTITYID_TL_SV

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_request_reader** = ENTITYID_TL_SV

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_reply_writer** = ENTITYID_TL_SVC_

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_TypeLookup_reply_reader** = ENTITYID_TL_SVC_

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_publications_secure_writer** = ENTITYID_SE

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_publications_secure_reader** = ENTITYID_SE

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_subscriptions_secure_writer** = ENTITYID_SE

const *EntityId_t* eprosima::fastrtps::rtps::**sedp_builtin_subscriptions_secure_reader** = ENTITYID_SE

const *EntityId_t* eprosima::fastrtps::rtps::**participant_volatile_message_secure_writer_entity_id**

const *EntityId_t* eprosima::fastrtps::rtps::**participant_volatile_message_secure_reader_entity_id**

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_WriterLivelinessSecure** = ENTITYID_P2P_BUILT

const *EntityId_t* eprosima::fastrtps::rtps::**c_EntityId_ReaderLivelinessSecure** = ENTITYID_P2P_BUILT

Macro definitions (#define)

ENTITYID_UNKNOWN

ENTITYID_RTPSParticipant

ENTITYID_SEDP_BUILTIN_TOPIC_WRITER

ENTITYID_SEDP_BUILTIN_TOPIC_READER

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_WRITER

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_READER

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_WRITER

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_READER

ENTITYID_SPDP_BUILTIN_RTPSParticipant_WRITER

ENTITYID_SPDP_BUILTIN_RTPSParticipant_READER

ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_WRITER

ENTITYID_P2P_BUILTIN_RTPSParticipant_MESSAGE_READER

ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_WRITER

ENTITYID_P2P_BUILTIN_PARTICIPANT_STATELESS_READER

ENTITYID_TL_SVC_REQ_WRITER

ENTITYID_TL_SVC_REQ_READER

ENTITYID_TL_SVC_REPLY_WRITER

ENTITYID_TL_SVC_REPLY_READER

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_WRITER

ENTITYID_SEDP_BUILTIN_PUBLICATIONS_SECURE_READER

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_WRITER

ENTITYID_SEDP_BUILTIN_SUBSCRIPTIONS_SECURE_READER

ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_WRITER

ENTITYID_P2P_BUILTIN_PARTICIPANT_MESSAGE_SECURE_READER

ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_WRITER

ENTITYID_P2P_BUILTIN_PARTICIPANT_VOLATILE_MESSAGE_SECURE_READER

ENTITYID_SPDP_RELIABLE_BUILTIN_PARTICIPANT_SECURE_WRITER

ENTITYID_SPDP_RELIABLE_BUILTIN_PARTICIPANT_SECURE_READER

EntityId_t

struct eprosima::fastrtps::rtps::**EntityId_t**

Structure *EntityId_t*, entity id part of *GUID_t*.

Public Functions

EntityId_t()

Default constructor. Unknown entity.

EntityId_t(uint32_t *id*)

Main constructor.

Parameters

- id: Entity id

EntityId_t(**const** *EntityId_t* &*id*)

Copy constructor.

EntityId_t(*EntityId_t* &&*id*)

Move constructor.

EntityId_t &**operator=**(uint32_t *id*)

Assignment operator.

Parameters

- id: Entity id to copy

EntityId_t Operators

bool eprosima::fastrtps::rtps::**operator==**(*EntityId_t* &*id1*, **const** uint32_t *id2*)

Guid prefix comparison operator

Return True if equal

Parameters

- id1: EntityId to compare
- id2: ID prefix to compare

bool eprosima::fastrtps::rtps::**operator==**(**const** *EntityId_t* &*id1*, **const** *EntityId_t* &*id2*)

Guid prefix comparison operator

Return True if equal

Parameters

- id1: First EntityId to compare
- id2: Second EntityId to compare

bool eprosima::fastrtps::rtps::**operator!=**(**const** *EntityId_t* &*id1*, **const** *EntityId_t* &*id2*)

Guid prefix comparison operator

Return True if not equal

Parameters

- id1: First EntityId to compare

- `id2`: Second EntityId to compare

std::ostream &`eprosima::fastrtps::rtps::`**`operator<<`**(std::ostream &*output*, **const** *EntityId_t &enI*)

std::istream &`eprosima::fastrtps::rtps::`**`operator>>`**(std::istream &*input*, *EntityId_t* &*enP*)

FragmentNumber

FragmentNumber_t

using `eprosima::fastrtps::rtps::`**`FragmentNumber_t`** = uint32_t

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 `eprosima::fastrtps::rtps::`**`GUID_t`**

Structure *GUID_t*, entity identifier, unique in DDS-RTPS Domain.

Public Functions

`GUID_t``()` **`noexcept`**

Default constructor.

Contructs an unknown GUID.

`GUID_t`(**const** *GuidPrefix_t* &*guid_prefix*, uint32_t *id*) **noexcept**

Construct

Parameters

- `guid_prefix`: Guid prefix
- `id`: Entity id

`GUID_t`(**const** *GuidPrefix_t* &*guid_prefix*, **const** *EntityId_t* &*entity_id*) **noexcept**

Parameters

- `guid_prefix`: Guid prefix
- `entity_id`: Entity id

bool **is_on_same_host_as**(**const** *GUID_t* &*other_guid*) **const**
Checks whether this guid is for an entity on the same host as another guid.

Return true when this guid is on the same host, false otherwise.

Parameters

- other_guid: *GUID_t* to compare to.

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.

Return true when this guid is on the same host and process, false otherwise.

Parameters

- other_guid: *GUID_t* to compare to.

bool **is_builtin**() **const**
Checks whether this guid corresponds to a builtin entity.

Return 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

bool eprosima::fastrtps::rtps::**operator==**(**const** *GUID_t* &*g1*, **const** *GUID_t* &*g2*)
GUID comparison operator

Return True if equal

Parameters

- g1: First GUID to compare
- g2: Second GUID to compare

bool eprosima::fastrtps::rtps::**operator!=**(**const** *GUID_t* &*g1*, **const** *GUID_t* &*g2*)
GUID comparison operator

Return True if not equal

Parameters

- g1: First GUID to compare
- g2: Second GUID to compare

bool eprosima::fastrtps::rtps::**operator<**(**const** *GUID_t* &*g1*, **const** *GUID_t* &*g2*)

std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &*output*, **const** *GUID_t* &*guid*)
Stream operator, prints a GUID.

Return Stream operator.

Parameters

- `output`: Output stream.
- `guid`: *GUID_t* to print.

std::istream &`eprosima::fastrtps::rtps::`**`operator>>`**(std::istream &*input*, *GUID_t* &*guid*)
Stream operator, retrieves a GUID.

Return Stream operator.

Parameters

- `input`: Input stream.
- `guid`: *GUID_t* to print.

GuidPrefix

c_GuidPrefix_Unknown

`const` *GuidPrefix_t* `eprosima::fastrtps::rtps::`**`c_GuidPrefix_Unknown`**

GuidPrefix_t

`struct` `eprosima::fastrtps::rtps::`**`GuidPrefix_t`**
Structure *GuidPrefix_t*, Guid Prefix of *GUID_t*.

Public Functions

`GuidPrefix_t``()`
Default constructor. Set the Guid prefix to 0.

bool **`operator==`**`(`**`const`** *GuidPrefix_t* &*prefix*`)` **`const`**
Guid prefix comparison operator

Return True if the guid prefixes are equal

Parameters

- `prefix`: guid prefix to compare

bool **`operator!=`**`(`**`const`** *GuidPrefix_t* &*prefix*`)` **`const`**
Guid prefix comparison operator

Return True if the guid prefixes are not equal

Parameters

- `prefix`: Second guid prefix to compare

bool **`operator<`**`(`**`const`** *GuidPrefix_t* &*prefix*`)` **`const`**
Guid prefix minor operator

Return True if prefix is higher

Parameters

- `prefix`: Second guid prefix to compare

GuidPrefix_t Operators

std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &*output*, **const** *GuidPrefix_t* &*guiP*)

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 eprosima::fastrtps::rtps::**InstanceHandle_t**

Struct *InstanceHandle_t*, used to contain the key for WITH_KEY topics.

Public Functions

InstanceHandle_t &**operator=**(**const** *InstanceHandle_t* &*ihandle*)

Assignment operator

Parameters

- ihandle: Instance handle to copy the data from

InstanceHandle_t &**operator=**(**const** *GUID_t* &*guid*)

Assignment operator

Parameters

- guid: GUID to copy the data from

bool **isDefined**() **const**

Know if the instance handle is defined

Return True if the values are not zero.

Public Members

octet **value**[16]

Value.

InstanceHandle_t Operators

bool eprosima::fastrtps::rtps::**operator==**(**const** *InstanceHandle_t* &*ihandle1*, **const** *InstanceHandle_t* &*ihandle2*)

Comparison operator

Return True if equal

Parameters

- ihandle1: First *InstanceHandle_t* to compare
- ihandle2: Second *InstanceHandle_t* to compare

bool eprosima::fastrtps::rtps::**operator!=**(**const** *InstanceHandle_t* &*ihandle1*, **const** *InstanceHandle_t* &*ihandle2*)

bool eprosima::fastrtps::rtps::**operator<**(**const** *InstanceHandle_t* &*h1*, **const** *InstanceHandle_t* &*h2*)

std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &*output*, **const** *InstanceHandle_t* &*iHandle*)

Parameters

- output:
- iHandle:

void eprosima::fastrtps::rtps::**iHandle2GUID**(*GUID_t* &*guid*, **const** *InstanceHandle_t* &*ihandle*)

Convert *InstanceHandle_t* to GUID

Parameters

- guid: GUID to store the results
- ihandle: *InstanceHandle_t* to copy

GUID_t eprosima::fastrtps::rtps::**iHandle2GUID**(**const** *InstanceHandle_t* &*ihandle*)

Convert GUID to *InstanceHandle_t*

Return *GUID_t*

Parameters

- ihandle: *InstanceHandle_t* to store the results

Locator

Macro definitions (#define)

LOCATOR_INVALID(*loc*)

LOCATOR_KIND_INVALID

LOCATOR_ADDRESS_INVALID(*a*)

LOCATOR_PORT_INVALID

LOCATOR_KIND_RESERVED

LOCATOR_KIND_UDPv4

LOCATOR_KIND_UDPv6

LOCATOR_KIND_TCPv4

LOCATOR_KIND_TCPv6

LOCATOR_KIND_SHM

IsAddressDefined

bool eprosima::fastrtps::rtps::**IsAddressDefined**(**const** *Locator_t* &*loc*)

IsLocatorValid

bool eprosima::fastrtps::rtps::**IsLocatorValid**(**const** *Locator_t* &*loc*)

Locator_t

class eprosima::fastrtps::rtps::**Locator_t**

Class *Locator_t*, uniquely identifies a communication channel for a particular transport.

Public Functions

Locator_t()

Default constructor.

Locator_t(*Locator_t* &&*loc*)

Move constructor.

Locator_t(**const** *Locator_t* &*loc*)

Copy constructor.

Locator_t(uint32_t *portin*)

Port constructor.

Locator_t(int32_t *kindin*, uint32_t *portin*)

Kind and port constructor.

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

LocatorList

class LocatorList
Class *LocatorList*, a Locator vector that doesn't avoid duplicates.

LocatorList_t

using eprosima::fastrtps::rtps::**LocatorList_t** = eprosima::fastdds::rtps::*LocatorList*

LocatorListConstIterator

typedef std::vector<*Locator_t*>::const_iterator eprosima::fastrtps::rtps::**LocatorListConstIterator**

LocatorListIterator

typedef std::vector<*Locator_t*>::iterator eprosima::fastrtps::rtps::**LocatorListIterator**

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*

Locator Operators

bool eprosima::fastrtps::rtps::**operator<**(**const** *Locator_t* &*loc1*, **const** *Locator_t* &*loc2*)

bool eprosima::fastrtps::rtps::**operator==**(**const** *Locator_t* &*loc1*, **const** *Locator_t* &*loc2*)

bool eprosima::fastrtps::rtps::**operator!=**(**const** *Locator_t* &*loc1*, **const** *Locator_t* &*loc2*)

std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &*output*, **const** *Locator_t* &*loc*)

std::ostream &eprosima::fastdds::rtps::**operator<<**(std::ostream &*output*, **const** *LocatorList* &*locList*)

bool eprosima::fastrtps::rtps::**operator==**(**const** ResourceLimitedVector<*Locator_t*> &*lhs*, **const** ResourceLimitedVector<*Locator_t*> &*rhs*)

LocatorSelectorEntry

struct eprosima::fastrtps::rtps::**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

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.

void **enable**(bool *should_enable*)

Set the enabled value.

Parameters

- should_enable: Whether this entry should be enabled.

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

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 `eprosima::fastrtps::rtps::`**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

LocatorSelector(**const** ResourceLimitedContainerConfig &*entries_allocation*)
Construct a *LocatorSelector*.

Parameters

- `entries_allocation`: Allocation configuration regarding the number of remote entities.

void **clear**()
Clears all internal data.

bool **add_entry**(*LocatorSelectorEntry* **entry*)
Add an entry to this selector.

Parameters

- `entry`: Pointer to the *LocatorSelectorEntry* to add.

bool **remove_entry**(**const** *GUID_t* &*guid*)
Remove an entry from this selector.

Parameters

- `guid`: Identifier of the entry to be removed.

void **reset**(bool *enable_all*)
Reset the enabling state of the selector.

Parameters

- `enable_all`: Indicates whether entries should be initially enabled.

void **enable**(**const** *GUID_t* &*guid*)
Enable an entry given its GUID.

Parameters

- `guid`: GUID of the entry to enable.

bool **state_has_changed**() **const**
Check if enabling state has changed.

Return true if the enabling state has changed, false otherwise.

void **selection_start**()
Reset the selection state of the selector.

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.

Return a reference to the entries collection.

void **select**(size_t *index*)
Marks an entry as selected.

Parameters

- `index`: The index of the entry to mark as selected.

size_t **`selected_size`**`()` **`const`**
Count the number of selected locators.

Return the number of selected locators.

bool **`is_selected`**`(`**`const`** *Locator_t locator*`)` **`const`**
Check if a locator is present in the selections of this object.

Return True if the locator has been selected, false otherwise.

Parameters

- `locator`: The locator to be checked.

template<class **`UnaryPredicate`**>
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*

`struct IteratorIndex`

MatchingInfo

MatchingInfo

`class` `eprosima::fastrtps::rtps::`**`MatchingInfo`**
Class *MatchingInfo* contains information about the matching between two endpoints.

Public Functions

`MatchingInfo``()`
Default constructor.

`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 eprosima::fastrtps::rtps::**PortParameters**
Class *PortParameters*, to define the port parameters and gains related with the RTPS protocol.

Public Functions

uint32_t **getMulticastPort**(uint32_t *domainId*) **const**
Get a multicast port based on the domain ID.

Return Multicast port

Parameters

- domainId: Domain ID.

uint32_t **getUnicastPort**(uint32_t *domainId*, uint32_t *RTPSParticipantID*) **const**
Get a unicast port based on the domain ID and the participant ID.

Return Unicast port

Parameters

- domainId: Domain ID.
- RTPSParticipantID: Participant ID.

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

std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream &*output*, **const** *RemoteLocatorList* &*remote_locators*)

RemoteLocatorList

struct eprosima::fastrtps::rtps::**RemoteLocatorList**

Holds information about the locators of a remote entity.

Public Functions

RemoteLocatorList()

Default constructor of *RemoteLocatorList* for deserialize.

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.

RemoteLocatorList(**const** *RemoteLocatorList* &*other*)

Copy-construct a *RemoteLocatorList*.

Parameters

- `other`: *RemoteLocatorList* to copy data from.

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.

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.

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 eprosima::fastrtps::rtps::**SampleIdentity**
This class is used to specify a sample.

Public Functions

SampleIdentity()
Default constructor.

Constructs an unknown *SampleIdentity*.

SampleIdentity(**const** *SampleIdentity* &*sample_id*)
Copy constructor.

SampleIdentity(*SampleIdentity* &&*sample_id*)
Move constructor.

SampleIdentity &**operator=**(**const** *SampleIdentity* &*sample_id*)
Assignment operator.

SampleIdentity &**operator=**(*SampleIdentity* &&*sample_id*)
Move constructor.

bool **operator<**(**const** *SampleIdentity* &*sample*) **const**
To allow using *SampleIdentity* as map key.

Return

Parameters

- sample:

SequenceNumber

c_SequenceNumber_Unknown

const SequenceNumber_t eprosima::fastrtps::rtps::c_SequenceNumber_Unknown (-1, 0)

SequenceNumber_t Operators

bool eprosima::fastrtps::rtps::**operator==**(**const** *SequenceNumber_t* &*sn1*, **const** *SequenceNumber_t* &*sn2*) **noexcept**

Compares two *SequenceNumber_t*.

Return True if equal

Parameters

- sn1: First *SequenceNumber_t* to compare
- sn2: Second *SequenceNumber_t* to compare

bool eprosima::fastrtps::rtps::**operator!=**(**const** *SequenceNumber_t* &*sn1*, **const** *SequenceNumber_t* &*sn2*) **noexcept**

Compares two *SequenceNumber_t*.

Return True if not equal

Parameters

- sn1: First *SequenceNumber_t* to compare
- sn2: Second *SequenceNumber_t* to compare

bool eprosima::fastrtps::rtps::**operator>**(**const** *SequenceNumber_t* &*seq1*, **const** *SequenceNumber_t* &*seq2*) **noexcept**

Checks if a *SequenceNumber_t* is greater than other.

Return True if the first *SequenceNumber_t* is greater than the second

Parameters

- seq1: First *SequenceNumber_t* to compare
- seq2: Second *SequenceNumber_t* to compare

bool eprosima::fastrtps::rtps::**operator<**(**const** *SequenceNumber_t* &*seq1*, **const** *SequenceNumber_t* &*seq2*) **noexcept**

Checks if a *SequenceNumber_t* is less than other.

Return True if the first *SequenceNumber_t* is less than the second

Parameters

- seq1: First *SequenceNumber_t* to compare
- seq2: Second *SequenceNumber_t* to compare

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.

Return True if the first *SequenceNumber_t* is greater or equal than the second

Parameters

- seq1: First *SequenceNumber_t* to compare
- seq2: Second *SequenceNumber_t* to compare

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.

Return True if the first *SequenceNumber_t* is less or equal than the second

Parameters

- `seq1`: First *SequenceNumber_t* to compare
- `seq2`: Second *SequenceNumber_t* to compare

SequenceNumber_t `eprosima::fastrtps::rtps::operator-`(**const** *SequenceNumber_t* &*seq*, **const** uint32_t *inc*) **noexcept**

Subtract one uint32_t from a *SequenceNumber_t*

Return Result of the subtraction

Parameters

- `seq`: Base *SequenceNumber_t*
- `inc`: uint32_t to subtract

SequenceNumber_t `eprosima::fastrtps::rtps::operator+`(**const** *SequenceNumber_t* &*seq*, **const** uint32_t *inc*) **noexcept**

Add one uint32_t to a *SequenceNumber_t*

Return Result of the addition

Parameters

- `[in] seq`: Base sequence number
- `inc`: value to add to the base

SequenceNumber_t `eprosima::fastrtps::rtps::operator-`(**const** *SequenceNumber_t* &*minuend*, **const** *SequenceNumber_t* &*subtrahend*) **noexcept**

Subtract one *SequenceNumber_t* to another

Return Result of the subtraction

Parameters

- `minuend`: Minuend. Has to be greater than or equal to subtrahend.
- `subtrahend`: Subtrahend.

std::ostream &`eprosima::fastrtps::rtps::operator<<`(std::ostream &*output*, **const** *SequenceNumber_t* &*seqNum*)

Return

Parameters

- `output`:
- `seqNum`:

std::ostream &`eprosima::fastrtps::rtps::operator<<`(std::ostream &*output*, **const** std::vector<*SequenceNumber_t*> &*seqNumSet*)

std::ostream &`eprosima::fastrtps::rtps::operator<<`(std::ostream &*output*, **const** *SequenceNumberSet_t* &*sns*)

Prints a sequence Number set

Return OStream.

Parameters

- `output`: Output Stream
- `sns`: SequenceNumber set

SequenceNumber_t

struct eprosima::fastrtps::rtps::**SequenceNumber_t**

Structure *SequenceNumber_t*, different for each change in the same writer.

Public Functions

SequenceNumber_t() **noexcept**

Default constructor.

SequenceNumber_t(int32_t *hi*, uint32_t *lo*) **noexcept**

Parameters

- hi:
- lo:

SequenceNumber_t(uint64_t *u*) **noexcept**

Parameters

- u:

uint64_t **to64long**() **const noexcept**

Convert the number to 64 bit.

Return 64 bit representation of the SequenceNumber

SequenceNumber_t &**operator++**() **noexcept**

Increase SequenceNumber in 1.

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*, *SequenceNumberDif*

Structure SequenceNumberSet_t, contains a group of sequencenumbers.

sort_seqNum

bool eprosima::fastrtps::rtps::**sort_seqNum**(**const** *SequenceNumber_t* &*s1*, **const** *SequenceNumber_t* &*s2*) **noexcept**

Sorts two instances of *SequenceNumber_t*

Return True if s1 is less than s2

Parameters

- `s1`: First *SequenceNumber_t* to compare
- `s2`: First *SequenceNumber_t* to compare

SerializedPayload

Macro definitions (#define)

CDR_BE

CDR_LE

PL_CDR_BE

PL_CDR_LE

SerializedPayload_t

struct eprosima::fastrtps::rtps::**SerializedPayload_t**

Structure *SerializedPayload_t*.

Public Functions

SerializedPayload_t()

Default constructor.

SerializedPayload_t(uint32_t *len*)

Parameters

- `len`: Maximum size of the payload

bool **copy**(**const** *SerializedPayload_t* *serData, bool *with_limit* = true)

Copy another structure (including allocating new space for the data.)

Return True if correct

Parameters

- `[in] serData`: Pointer to the structure to copy
- `with_limit`: if true, the function will fail when providing a payload too big

bool **reserve_fragmented**(*SerializedPayload_t* *serData)
Allocate new space for fragmented data.

Return True if correct

Parameters

- `[in] serData`: Pointer to the structure to copy

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

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

TIME_T_INFINITE_NANOSECONDS

eprosima::fastrtps::Duration_t

using eprosima::fastrtps::**Duration_t** = *Time_t*

eprosima::fastrtps::Time_t

struct eprosima::fastrtps::**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

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

bool eprosima::fastrtps::rtps::**operator==**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Comparison assignment

Return True if equal

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::rtps::**operator!=**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Comparison assignment

Return True if not equal

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::rtps::**operator<**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Checks if a *Time_t* is less than other.

Return True if the first *Time_t* is less than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::rtps::**operator>**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Checks if a *Time_t* is greater than other.

Return True if the first *Time_t* is greater than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::rtps::**operator<=**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Checks if a *Time_t* is less or equal than other.

Return True if the first *Time_t* is less or equal than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::rtps::**operator>=**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Checks if a *Time_t* is greater or equal than other.

Return True if the first *Time_t* is greater or equal than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

std::ostream &eprosima::fastrtps::rtps::**operator<<**(std::ostream *&output*, **const** *Time_t* *&t*)

Time_t eprosima::fastrtps::rtps::**operator+**(**const** *Time_t* *&ta*, **const** *Time_t* *&tb*)
Adds two *Time_t*.

Return A new *Time_t* with the result.

Parameters

- ta: First *Time_t* to add
- tb: Second *Time_t* to add

Time_t eprosima::fastrtps::rtps::**operator-**(**const** *Time_t* *&ta*, **const** *Time_t* *&tb*)
Subtracts two *Time_t*.

Return A new *Time_t* with the result.

Parameters

- ta: First *Time_t* to subtract
- tb: Second *Time_t* to subtract

bool eprosima::fastrtps::**operator==**(**const** *Time_t* *&t1*, **const** *Time_t* *&t2*)
Comparison assignment

Return True if equal

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::**operator!=**(**const** *Time_t* *&t1*, **const** *Time_t* *&t2*)
Comparison assignment

Return True if not equal

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::**operator<**(**const** *Time_t* *&t1*, **const** *Time_t* *&t2*)
Checks if a *Time_t* is less than other.

Return True if the first *Time_t* is less than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::**operator>**(**const** *Time_t* *&t1*, **const** *Time_t* *&t2*)
Checks if a *Time_t* is greater than other.

Return True if the first *Time_t* is greater than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::**operator<=**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Checks if a *Time_t* is less or equal than other.

Return True if the first *Time_t* is less or equal than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

bool eprosima::fastrtps::**operator>=**(**const** *Time_t* &*t1*, **const** *Time_t* &*t2*)
Checks if a *Time_t* is greater or equal than other.

Return True if the first *Time_t* is greater or equal than the second

Parameters

- t1: First *Time_t* to compare
- t2: Second *Time_t* to compare

std::ostream &eprosima::fastrtps::**operator<<**(std::ostream &*output*, **const** *Time_t* &*t*)

Time_t eprosima::fastrtps::**operator+**(**const** *Time_t* &*ta*, **const** *Time_t* &*tb*)
Adds two *Time_t*.

Return A new *Time_t* with the result.

Parameters

- ta: First *Time_t* to add
- tb: Second *Time_t* to add

Time_t eprosima::fastrtps::**operator-**(**const** *Time_t* &*ta*, **const** *Time_t* &*tb*)
Subtracts two *Time_t*.

Return A new *Time_t* with the result.

Parameters

- ta: First *Time_t* to subtract
- tb: Second *Time_t* to subtract

Time_t

class eprosima::fastrtps::rtps::**Time_t**
Structure *Time_t*, used to describe times at RTPS protocol.

Public Functions

Time_t()
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.

void **fraction** (uint32_t *frac*)
Sets fraction field and updates the nanoseconds.

Public Static Functions

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

BIT1

BIT2

BIT3

BIT4

BIT5

BIT6

BIT7

BIT(*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 = 0x1
Big endianness.

enumerator LITTLEEND = 0x0
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.

std::ostream &`eprosima::fastrtps::rtps::`**operator<<**(std::ostream *&output*, **const** *ProtocolVersion_t* *&pv*)
Prints a ProtocolVersion

Return OStream.

Parameters

- `output`: Output Stream
- `pv`: ProtocolVersion

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 eprosima::fastrtps::rtps::**WriteParams**

This class contains additional information of a CacheChange.

Public Functions

WriteParams() = default

Default constructor.

WriteParams(**const** *WriteParams* &*wparam*)

Copy constructor.

WriteParams(*WriteParams* &&*wparam*)

Move constructor.

WriteParams &**operator=**(**const** *WriteParams* &*wparam*)

Assignment operator.

WriteParams &**operator=**(*WriteParams* &&*wparam*)

Assignment operator.

Endpoint

class eprosima::fastrtps::rtps::**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

const *GUID_t* &**getGuid**() **const**

Get associated GUID

Return Associated GUID

RecursiveTimedMutex &**getMutex**()

Get mutex

Return Associated Mutex

EndpointAttributes &**getAttributes**()

Get associated attributes

Return *Endpoint* attributes

Exceptions

Exception

class eprosima::fastrtps::rtps::**Exception** : **public** exception
This abstract class is used to create exceptions.

Subclassed by eprosima::fastrtps::rtps::security::SecurityException

Public Functions

~Exception()
Default destructor.

const int32_t &**minor**() **const**
This function returns the number associated with the system exception.

Return 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.

void **raise**() **const** = 0
This function throws the object as exception.

const char ***what**() **const**
This function returns the error message.

Return The error message.

Flow control

ThroughputControllerDescriptor

struct eprosima::fastrtps::rtps::**ThroughputControllerDescriptor**
Descriptor for a Throughput Controller, containing all constructor information for it.

Public Members

uint32_t **bytesPerPeriod**
Packet size in bytes that this controller will allow in a given period.

uint32_t **periodMillisecs**
Window of time in which no more than 'bytesPerPeriod' bytes are allowed.

History

History

class eprosima::fastrtps::rtps::**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

bool **reserve_Cache**(*CacheChange_t* ***change*, **const** std::function<uint32_t) > &*calculateSizeFunc*Reserve a *CacheChange_t* from the CacheChange pool.

Return True if reserved

Warning This method has been deprecated and will be removed on v3.0.0

Parameters

- `[out] change`: Pointer to pointer to the *CacheChange_t* to reserve
- `[in] calculateSizeFunc`: Function to calculate the size of the payload.

bool **reserve_Cache**(*CacheChange_t* ***change*, uint32_t *dataSize*)

Reserve a *CacheChange_t* from the CacheChange pool.

Return True if reserved

Warning This method has been deprecated and will be removed on v3.0.0

Parameters

- `[out] change`: Pointer to pointer to the *CacheChange_t* to reserve
- `[in] dataSize`: Required size for the payload.

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*.

bool **isFull**()

Check if the history is full

Return true if the *History* is full.

size_t **getHistorySize**()

Get the *History* size.

Return 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

Return an iterator if a suitable change is found

Parameters

- `ch`: Pointer to the *CacheChange_t* to search for.

iterator **remove_change_nts**(const_iterator *removal*, bool *release* = true)
Remove a specific change from the history. No Thread Safe

Return iterator to the next *CacheChange_t* or end iterator.

Parameters

- `removal`: iterator to the *CacheChange_t* to remove.
- `release`: defaults to true and hints if the *CacheChange_t* should return to the pool

bool **remove_all_changes**()
Remove all changes from the *History*

Return True if everything was correctly removed.

bool **remove_change**(*CacheChange_t* **ch*)
Remove a specific change from the history.

Return True if removed.

Parameters

- `ch`: Pointer to the *CacheChange_t*.

const_iterator **find_change**(*CacheChange_t* **ch*)
Find a specific change in the history using the matches_change method criteria.

Return an iterator if a suitable change is found

Parameters

- `ch`: Pointer to the *CacheChange_t* to search for.

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.

Return true if the iterator identifies this change.

Parameters

- `ch_inner`: element of the collection to compare with the given change
- `ch_outer`: Pointer to the *CacheChange_t* to identify.

iterator **remove_change**(const_iterator *removal*, bool *release* = true)
Remove a specific change from the history.

Return iterator to the next *CacheChange_t* or end iterator.

Parameters

- `removal`: iterator to the *CacheChange_t* to remove.
- `release`: defaults to true and hints if the *CacheChange_t* should return to the pool

iterator **changesBegin**()
Get the beginning of the changes history iterator.

Return Iterator to the beginning of the vector.

iterator **changesEnd**()
Get the end of the changes history iterator.

Return Iterator to the end of the vector.

bool **get_min_change**(*CacheChange_t* ***min_change*)
Get the minimum *CacheChange_t*.

Return True if correct.

Parameters

- `min_change`: Pointer to pointer to the minimum change.

bool **get_max_change**(*CacheChange_t* ***max_change*)
Get the maximum *CacheChange_t*.

Return True if correct.

Parameters

- `max_change`: Pointer to pointer to the maximum change.

uint32_t **getTypeMaxSerialized**()
Get the maximum serialized payload size

Return Maximum serialized payload size

RecursiveTimedMutex ***getMutex**()
Get the mutex.

Return Mutex

bool **get_earliest_change**(*CacheChange_t* ***change*)
A method to get the change with the earliest timestamp.

Return True on success

Parameters

- `change`: Pointer to pointer to earliest change

Public Members

HistoryAttributes **m_att**
Attributes of the *History*.

IChangePool

class `eprosima::fastrtps::rtps::IChangePool`
An interface for classes responsible of cache changes allocation management.

Public Functions

bool **reserve_cache**(*CacheChange_t* *&*cache_change*) = 0
Get a new cache change from the pool.

Return 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`

Parameters

- `[out] cache_change`: Pointer to the new cache change.

bool **`release_cache`**(*CacheChange_t* **cache_change*) = 0
Return a cache change to the pool.

Return 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`

Parameters

- `[in] cache_change`: Pointer to the cache change to release.

IPayloadPool

`class` `eprosima::fastrtps::rtps::`**`IPayloadPool`**
An interface for classes responsible of serialized payload management.

Public Functions

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.

Return 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`

Parameters

- `[in] size`: Number of bytes required for the serialized payload. Should be greater than 0.
- `[inout] cache_change`: Cache change to assign the payload to

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.

Return whether the operation succeeded or not

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.

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`

Parameters

- `[inout] data`: Serialized payload received
- `[inout] data_owner`: Payload pool owning incoming data
- `[inout] cache_change`: Cache change to assign the payload to

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.

Return whether the operation succeeded or not

Pre

- Field `payload_owner` of `cache_change` equals this

Post

- Field `payload_owner` of `cache_change` is `nullptr`

Parameters

- `[inout] cache_change`: Cache change to assign the payload to

ReaderHistory

class `eprosima::fastrtps::rtps::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*.

bool **received_change**(*CacheChange_t* **change*, size_t)

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.

Return True if added.

Parameters

- `change`: Pointer to the change

bool **add_change**(*CacheChange_t* **a_change*)

Add a *CacheChange_t* to the *ReaderHistory*.

Return True if added.

Parameters

- `a_change`: Pointer to the CacheChange to add.

iterator **remove_change_nts**(const_iterator *removal*, bool *release* = true) **override**

Remove a specific change from the history. No Thread Safe

Return iterator to the next change if any

Parameters

- `removal`: iterator to the change for removal
- `release`: specifies if the change must be returned to the pool

bool **matches_change**(**const** *CacheChange_t* **inner*, *CacheChange_t* **outer*) **override**

Criteria to search a specific *CacheChange_t* on history

Return true if inner matches outer criteria

Parameters

- `inner`: change to compare
- `outer`: change for comparison

bool **remove_changes_with_guid**(**const** *GUID_t* &*a_guid*)

Remove all changes from the *History* that have a certain guid.

Return True if successful, even if no changes have been removed.

Parameters

- `a_guid`: Pointer to the target guid to search for.

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.

Return True if successful, even if no changes have been removed.

Parameters

- `seq_num`: First *SequenceNumber_t* not to be removed.
- `writer_guid`: GUID of the writer for which changes should be looked for.

WriterHistory

class `eprosima::fastrtps::rtps::`**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*.

Return True if added.

Parameters

- `a_change`: Pointer to the *CacheChange_t* to be added.

bool **add_change**(*CacheChange_t* **a_change*, *WriteParams* &*wparams*)
Add a *CacheChange_t* to the *WriterHistory*.

Return True if added.

Parameters

- `a_change`: Pointer to the *CacheChange_t* to be added.
- `wparams`: Extra write parameters.

iterator **remove_change_nts**(const_iterator *removal*, bool *release* = true) **override**
Remove a specific change from the history. No Thread Safe

Return iterator to the next change if any

Parameters

- `removal`: iterator to the change for removal
- `release`: specifies if the change should be return to the pool

bool **matches_change**(**const** *CacheChange_t* **inner*, *CacheChange_t* **outer*) **override**
Criteria to search a specific *CacheChange_t* on history

Return true if inner matches outer criteria

Parameters

- `inner`: change to compare
- `outer`: change for comparison

bool **remove_min_change**()
Remove the *CacheChange_t* with the minimum sequenceNumber.

Return True if correctly removed.

RTPSParticipant

ParticipantDiscoveryInfo

ParticipantAuthenticationInfo

struct eprosima::fastrtps::rtps::**ParticipantAuthenticationInfo**

Public Members

AUTHENTICATION_STATUS **status**
Status.

GUID_t **guid**
Associated GUID.

bool eprosima::fastrtps::rtps::**operator==**(**const** *ParticipantAuthenticationInfo* &*l*, **const** *ParticipantAuthenticationInfo* &*r*)

ParticipantDiscoveryInfo

struct eprosima::fastrtps::rtps::**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

Public Members

DISCOVERY_STATUS **status**
Status.

const *ParticipantProxyData* &**info**
Participant discovery info.

ParticipantProxyData

class eprosima::fastrtps::rtps::**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.

Return True on success

Parameters

- pdata: Object to copy the data from

uint32_t **get_serialized_size**(bool *include_encapsulation*) **const**
Get the size in bytes of the CDR serialization of this object.

Return size in bytes of the CDR serialization.

Parameters

- include_encapsulation: Whether to include the size of the encapsulation info.

bool **writeToCDRMessage**(*CDRMessage_t* *msg, bool *write_encapsulation*)
Write as a parameter list on a *CDRMessage_t*

Return True on success

bool **readFromCDRMessage**(*CDRMessage_t* *msg, bool *use_encapsulation*, **const** NetworkFactory &*network*, bool *is_shm_transport_available*)
Read the parameter list from a received *CDRMessage_t*

Return 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*

Return 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*

Return *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

Return GUID

Public Members

ProtocolVersion_t **m_protocolVersion**
Protocol version.

GUID_t **m_guid**
GUID.

VendorId_t **m_VendorId**
Vendor ID.

bool **m_expectsInlineQos**
Expects Inline QOS.

BuiltinEndpointSet_t **m_availableBuiltinEndpoints**
Available builtin endpoints.

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

TYPELOOKUP_DATA_MAX_SIZE

DISC_BUILTIN_ENDPOINT_PARTICIPANT_ANNOUNCER

DISC_BUILTIN_ENDPOINT_PARTICIPANT_DETECTOR

DISC_BUILTIN_ENDPOINT_PUBLICATION_ANNOUNCER

DISC_BUILTIN_ENDPOINT_PUBLICATION_DETECTOR

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_ANNOUNCER

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_DETECTOR

DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_ANNOUNCER

DISC_BUILTIN_ENDPOINT_PARTICIPANT_PROXY_DETECTOR

DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_ANNOUNCER

DISC_BUILTIN_ENDPOINT_PARTICIPANT_STATE_DETECTOR

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_WRITER

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_DATA_READER

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REQUEST_DATA_WRITER

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REQUEST_DATA_READER

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REPLY_DATA_WRITER

BUILTIN_ENDPOINT_TYPELOOKUP_SERVICE_REPLY_DATA_READER

DISC_BUILTIN_ENDPOINT_PUBLICATION_SECURE_ANNOUNCER

DISC_BUILTIN_ENDPOINT_PUBLICATION_SECURE_DETECTOR

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_SECURE_ANNOUNCER

DISC_BUILTIN_ENDPOINT_SUBSCRIPTION_SECURE_DETECTOR

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_SECURE_DATA_WRITER

BUILTIN_ENDPOINT_PARTICIPANT_MESSAGE_SECURE_DATA_READER

DISC_BUILTIN_ENDPOINT_PARTICIPANT_SECURE_ANNOUNCER

DISC_BUILTIN_ENDPOINT_PARTICIPANT_SECURE_DETECTOR

ReaderDiscoveryInfo

struct eprosima::fastrtps::rtps::**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

Public Members

DISCOVERY_STATUS **status**
Status.

const *ReaderProxyData* &**info**
Participant discovery info.

ReaderProxyData

class eprosima::fastrtps::rtps::**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

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*

Return *SampleIdentity*

uint32_t **get_serialized_size**(bool *include_encapsulation*) **const**
Get the size in bytes of the CDR serialization of this object.

Return size in bytes of the CDR serialization.

Parameters

- include_encapsulation: Whether to include the size of the encapsulation info.

bool **writeToCDRMessage**(*CDRMessage_t* *msg, bool *write_encapsulation*) **const**
Write as a parameter list on a *CDRMessage_t*

Return True on success

bool **readFromCDRMessage**(*CDRMessage_t* *msg, **const** NetworkFactory &*network*, bool *is_shm_transport_available*)
Read the information from a *CDRMessage_t*. The position of the message must be in the beginning on the parameter list.

Return true on success

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.

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.

Return true if this object can be updated with the information on rdata.

Parameters

- `rdata`: *ReaderProxyData* object to be checked.

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 `eprosima::fastrtps::rtps::`**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

Public Members

DISCOVERY_STATUS **status**
Status.

const *WriterProxyData* &**info**
Participant discovery info.

WriterProxyData

class eprosima::fastrtps::rtps::**WriterProxyData**

Public Functions

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*

Return *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.

Return true if this object can be updated with the information on wdata.

Parameters

- `wdata`: *WriterProxyData* object to be checked.

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.

Return size in bytes of the CDR serialization.

Parameters

- `include_encapsulation`: Whether to include the size of the encapsulation info.

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*)
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 eprosima::fastrtps::rtps::**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

Return True if correctly added.

Parameters

- `pguid`: *GUID_t* of the discovered Writer.
- `userDefinedId`: ID of the discovered Writer.

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

Return True if correctly added.

Parameters

- `pguid`: *GUID_t* of the discovered Reader.
- `userDefinedId`: ID of the discovered Reader.

uint32_t **getRTPSParticipantID**() **const**
Get the Participant ID.

Return Participant ID.

bool **registerWriter**(*RTPSWriter* *Writer, **const** TopicAttributes &topicAtt, **const** WriterQos &wqos)
Register a *RTPSWriter* in the builtin Protocols.

Return True if correctly registered.

Parameters

- `Writer`: Pointer to the *RTPSWriter*.
- `topicAtt`: Topic Attributes where you want to register it.
- `wqos`: WriterQos.

bool **registerReader**(*RTPSReader* *Reader, **const** TopicAttributes &topicAtt, **const** ReaderQos &rqos)
Register a *RTPSReader* in the builtin Protocols.

Return True if correctly registered.

Parameters

- `Reader`: Pointer to the *RTPSReader*.
- `topicAtt`: Topic Attributes where you want to register it.
- `rqos`: ReaderQos.

bool **updateWriter**(*RTPSWriter* *Writer, **const** TopicAttributes &topicAtt, **const** WriterQos &wqos)
Update writer QOS

Return true on success

Parameters

- `Writer`: to update
- `topicAtt`: Topic Attributes where you want to register it.
- `wqos`: New writer QoS

bool **updateReader**(*RTPSReader* *Reader, **const** TopicAttributes &topicAtt, **const** ReaderQos &rqos)
Update reader QOS

Return true on success

Parameters

- `Reader`: to update
- `topicAtt`: Topic Attributes where you want to register it.
- `rqos`: New reader QoS

std::vector<std::string> **getParticipantNames**() **const**
Returns a list with the participant names.

Return list of participant names.

const *RTPSParticipantAttributes* &**getRTPSParticipantAttributes**() **const**
Get a copy of the actual state of the RTPSParticipantParameters

Return *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.

Return 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.

Return True if filled or the entityId is available.

Parameters

- `entityId`: to check of fill. If filled, EntityKind will be "vendor-specific" (0x01)

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.

Return

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 **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 eprosima::fastrtps::rtps::**RTPSParticipantListener**

Class *RTPSParticipantListener* with virtual method that the user can overload to respond to certain events.

Public Functions

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`: Pointer to the Participant which discovered the remote participant.
- `info`: Remote participant information. User can take ownership of the object.

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`: Pointer to the Participant which discovered the remote reader.
- `info`: Remote reader information. User can take ownership of the object.

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`: Pointer to the Participant which discovered the remote writer.
- `info`: Remote writer information. User can take ownership of the object.

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;

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.

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 eprosima::fastrtps::rtps::**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

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

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

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).

void **on_liveliness_changed**(*RTPSReader* *reader, **const** LivelinessChangedStatus &status)

Method called when the liveliness of a reader changes.

Parameters

- reader: The reader
- status: The liveliness changed status

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.

RTPSReader

class `eprosima::fastrtps::rtps::RTPSReader` : **public** eprosima::fastrtps::rtps::*Endpoint*, **public** eprosima::fastdd

Class *RTPSReader*, manages the reception of data from its matched writers.

Subclassed by eprosima::fastrtps::rtps::StatefulReader, eprosima::fastrtps::rtps::StatelessReader

Public Functions

bool **matched_writer_add**(**const** *WriterProxyData* *&wdata*) = 0

Add a matched writer represented by its attributes.

Return True if correctly added.

Parameters

- `wdata`: Attributes of the writer to add.

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.

Return True if correctly removed.

Parameters

- `writer_guid`: GUID of the writer to remove.
- `removed_by_lease`: Whether the writer is being unmatched due to a participant drop.

bool **matched_writer_is_matched**(**const** *GUID_t* *&writer_guid*) = 0

Tells us if a specific Writer is matched against this reader.

Return True if it is matched.

Parameters

- `writer_guid`: GUID of the writer to check.

bool **processDataMsg**(*CacheChange_t* **change*) = 0

Processes a new DATA message. Previously the message must have been accepted by function acceptMsgDirectedTo.

Return true if the reader accepts messages from the.

Parameters

- `change`: Pointer to the *CacheChange_t*.

bool **processDataFragMsg**(*CacheChange_t* *change, uint32_t sampleSize, uint32_t fragmentStartingNum, uint16_t fragmentsInSubmessage) = 0

Processes a new DATA FRAG message.

Return true if the reader accepts 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.

bool **processHeartbeatMsg**(**const** *GUID_t* &writerGUID, uint32_t hbCount, **const** *SequenceNumber_t* &firstSN, **const** *SequenceNumber_t* &lastSN, bool finalFlag, bool livelinessFlag) = 0

Processes a new HEARTBEAT message.

Return true if the reader accepts messages from the.

Parameters

- `writerGUID`:
- `hbCount`:
- `firstSN`:
- `lastSN`:
- `finalFlag`:
- `livelinessFlag`:

bool **processGapMsg**(**const** *GUID_t* &writerGUID, **const** *SequenceNumber_t* &gapStart, **const** *SequenceNumberSet_t* &gapList) = 0

Processes a new GAP message.

Return true if the reader accepts messages from the.

Parameters

- `writerGUID`:
- `gapStart`:
- `gapList`:

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.

Return True if correctly removed.

Parameters

- `change`: Pointer to the *CacheChange_t*.
- `prox`: Pointer to the WriterProxy.

ReaderListener ***getListener**() **const**

Get the associated listener, secondary attached Listener in case it is of compound type

Return 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.

Return True is correctly set.

Parameters

- `target`: Pointed to ReaderLister to attach

bool **reserveCache**(*CacheChange_t* **change*, uint32_t *dataCdrSerializedSize*)
Reserve a *CacheChange_t*.

Return True if correctly reserved.

Parameters

- `change`: Pointer to pointer to the Cache.
- `dataCdrSerializedSize`: Size of the Cache.

void **releaseCache**(*CacheChange_t* *change)
Release a cacheChange.

bool **nextUnreadCache**(*CacheChange_t* **change*, WriterProxy **wp*) = 0
Read the next unread *CacheChange_t* from the history

Return True if read.

Parameters

- `change`: Pointer to pointer of *CacheChange_t*
- `wp`: Pointer to pointer to the WriterProxy

bool **nextUntakenCache**(*CacheChange_t* **change*, WriterProxy **wp*) = 0
Get the next *CacheChange_t* from the history to take.

Return True if read.

Parameters

- `change`: Pointer to pointer of *CacheChange_t*.
- `wp`: Pointer to pointer to the WriterProxy.

bool **expectsInlineQos**()

Return True if the reader expects Inline QOS.

ReaderHistory ***getHistory**()
Returns a pointer to the associated *History*.

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.

Return There is a clean state with all Writers.

void **assert_writer_liveliness**(**const** *GUID_t* &*writer*) = 0
Assert the liveliness of a matched writer.

Parameters

- `writer`: GUID of the writer to assert.

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.

Return Whether the change is still valid or not.

Parameters

- `[in] change`: Pointer to the change being accessed.
- `[out] wp`: Writer proxy the `change` belongs to.
- `[out] is_future_change`: Whether the change is in the future (i.e. there are earlier unreceived changes from the same writer).

void **end_sample_access_nts**(*CacheChange_t* *change, WriterProxy *&wp, bool mark_as_read) = 0

Called after the change has been deserialized.

Parameters

- `[in] change`: Pointer to the change being accessed.
- `[in] wp`: Writer proxy the `change` belongs to.
- `[in] mark_as_read`: Whether the `change` should be marked as read or not.

void **change_read_by_user**(*CacheChange_t* *change, **const** 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

Return true if the sample is valid

Parameters

- `data`: Pointer to the sample data to check
- `writer`: GUID of the writer that sent `data`
- `sn`: Sequence number related to `data`

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 eprosima::fastrtps::rtps::**RTPSDomain**

Class *RTPSDomain*,it manages the creation and destruction of *RTPSParticipant RTPSWriter* and *RTPSReader*. It stores a list of all created *RTPSParticipant*. Is has only static methods.

Public Static Functions

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.

RTPSParticipant ***createParticipant**(uint32_t *domain_id*, **const** *RTPSParticipantAttributes* *&attrs*, *RTPSParticipantListener* **plisten* = nullptr)

Create a *RTPSParticipant*.

Return Pointer to the *RTPSParticipant*.

Parameters

- `domain_id`: DomainId to be used by the *RTPSParticipant* (80 by default).
- `attrs`: *RTPSParticipant* Attributes.
- `plisten`: Pointer to the ParticipantListener.

RTPSParticipant ***createParticipant**(uint32_t *domain_id*, bool *enabled*, **const** *RTPSParticipantAttributes* *&attrs*, *RTPSParticipantListener* **plisten* = nullptr)

Create a *RTPSParticipant*.

Return Pointer to the *RTPSParticipant*.

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.

RTPSWriter ***createRTPSWriter**(*RTPSParticipant* *p, *WriterAttributes* &watt, *WriterHistory* *hist, *WriterListener* *listen = nullptr)

Create a *RTPSWriter* in a participant.

Return Pointer to the created *RTPSWriter*.

Parameters

- `p`: Pointer to the *RTPSParticipant*.
- `watt`: Writer Attributes.
- `hist`: Pointer to the *WriterHistory*.
- `listen`: Pointer to the *WriterListener*.

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.

Return Pointer to the created *RTPSWriter*.

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*.

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.

Return Pointer to the created *RTPSWriter*.

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*.

bool **removeRTPSWriter**(*RTPSWriter* *writer)

Remove a *RTPSWriter*.

Return True if correctly removed.

Parameters

- `writer`: Pointer to the writer you want to remove.

RTPSReader ***createRTPSReader**(*RTPSParticipant* *p, *ReaderAttributes* &ratt, *ReaderHistory* *hist, *ReaderListener* *listen = nullptr)

Create a *RTPSReader* in a participant.

Return Pointer to the created *RTPSReader*.

Parameters

- `p`: Pointer to the *RTPSParticipant*.
- `ratt`: Reader Attributes.
- `hist`: Pointer to the *ReaderHistory*.
- `listen`: Pointer to the *ReaderListener*.

RTPSReader ***createRTPSReader**(*RTPSParticipant* *p, *ReaderAttributes* &ratt, **const** std::shared_ptr<*IPayloadPool*> &*payload_pool*, *ReaderHistory* *hist, *ReaderListener* *listen = nullptr)

Create a *RTPSWriter* in a participant using a custom payload pool.

Return Pointer to the created *RTPSReader*.

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*.

bool **removeRTPSReader**(*RTPSReader* *reader)

Remove a *RTPSReader*.

Return True if correctly removed.

Parameters

- `reader`: Pointer to the reader you want to remove.

bool **removeRTPSParticipant**(*RTPSParticipant* *p)

Remove a *RTPSParticipant* and delete all its associated Writers, Readers, resources, etc.

Return True if correct.

Parameters

- `[in] p`: Pointer to the *RTPSParticipant*;

void **setMaxRTPSParticipantId**(uint32_t *maxRTPSParticipantId*)

Set the maximum RTPSParticipantID.

Parameters

- `maxRTPSParticipantId`: ID.

RTPSParticipant ***clientServerEnvironmentCreationOverride**(uint32_t *domain_id*, bool *enabled*, **const** *RTPSParticipantAttributes* &*attrs*, *RTPSParticipantListener* **listen*)

Creates a *RTPSParticipant* as default server or client if ROS_MASTER_URI environment variable is set.

Return Pointer to the *RTPSParticipant*.

Parameters

- `domain_id`: DDS domain associated
- `enabled`: True if the *RTPSParticipant* should be enabled on creation. False if it will be enabled later with *RTPSParticipant::enable()*
- `attrs`: *RTPSParticipant* Attributes.
- `listen`: Pointer to the ParticipantListener.

RTPSWriter

LivelinessData

struct `eprosima::fastrtps::rtps::`**LivelinessData**

A struct keeping relevant liveliness information of a writer.

Public Functions

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

bool **operator==**(**const** *LivelinessData* &*other*) **const**

Equality operator.

Return True if equal

Parameters

- `other`: Liveliness data to compare to

bool **operator!=**(**const** *LivelinessData* &*other*) **const**

Inequality operator.

Return True if different

Parameters

- `other`: Liveliness data to compare to

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 `eprosima::fastrtps::rtps::RTPSWriter` : **public** eprosima::fastrtps::rtps::*Endpoint*, **public** eprosima::fastrtp
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**>
CacheChange_t ***new_change**(*T &data*, *ChangeKind_t changeKind*, *InstanceHandle_t handle* = *c_InstanceHandle_Unknown*)
Create a new change based with the provided changeKind.

Return Pointer to the CacheChange or nullptr if incorrect.

Parameters

- `data`: Data of the change.
- `changeKind`: The type of change.
- `handle`: InstanceHandle to assign.

bool **release_change**(*CacheChange_t *change*)
Release a change when it is not being used anymore.

Return 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

Parameters

- `change`: Pointer to the cache change to be released.

bool **matched_reader_add**(**const** *ReaderProxyData* &*data*) = 0
Add a matched reader.

Return True if added.

Parameters

- `data`: Pointer to the *ReaderProxyData* object added.

bool **matched_reader_remove**(**const** *GUID_t* &*reader_guid*) = 0
Remove a matched reader.

Return True if removed.

Parameters

- `reader_guid`: GUID of the reader to remove.

bool **matched_reader_is_matched**(**const** *GUID_t* &*reader_guid*) = 0
Tells us if a specific Reader is matched against this writer.

Return True if it was matched.

Parameters

- `reader_guid`: GUID of the reader to check.

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.

Return True if acknowledged by all.

bool **wait_for_all_acked**(**const** *Duration_t*&)
Waits until all changes were acknowledged or max_wait.

Return True if all were acknowledged.

void **updateAttributes**(**const** *WriterAttributes* &*att*) = 0
Update the Attributes of the Writer.

Parameters

- `att`: New attributes

void **send_any_unsent_changes**() = 0
This method triggers the send operation for unsent changes.

Return number of messages sent

SequenceNumber_t **get_seq_num_min**()
Get Min Seq Num in *History*.

Return Minimum sequence number in history

SequenceNumber_t **get_seq_num_max**()
Get Max Seq Num in *History*.

Return Maximum sequence number in history

uint32_t **getTypeMaxSerialized**()
Get maximum size of the serialized type

Return 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.

WriterListener ***getListener**()
Get listener

Return Listener

bool **isAsync**() **const**
Get the publication mode

Return publication mode

bool **remove_older_changes**(unsigned int *max* = 0)
Remove an specified max number of changes

Return at least one change has been removed

Parameters

- `max`: Maximum number of changes to remove.

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.

Return at least one change has been removed

Parameters

- `max_blocking_time_point`: Maximum time to wait for.
- `lock`: Lock of the Change list.

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.

Return true when change was acknowledged, false when timeout is reached.

Parameters

- `seq`: Sequence number to wait for acknowledgement.
- `max_blocking_time_point`: Maximum time to wait for.
- `lock`: Lock of the Change list.

RTPSParticipantImpl ***getRTPSParticipant**() **const**
Get RTPS participant

Return RTPS participant

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

bool **get_separate_sending**() **const**
Inform if data is sent to readers separately

Return true if separate sending is enabled

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*)

Process an incoming ACKNACK submessage.

Return true when the submessage was destinated to this writer, false otherwise.

Parameters

- `[in] writer_guid`: GUID of the writer the submessage is directed to.
- `[in] reader_guid`: GUID of the reader originating the submessage.
- `[in] ack_count`: Count field of the submessage.
- `[in] sn_set`: Sequence number bitmap field of the submessage.
- `[in] final_flag`: Final flag field of the submessage.
- `[out] result`: true if the writer could process the submessage. Only valid when returned value is true.

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*)

Process an incoming NACKFRAG submessage.

Return true when the submessage was destinated to this writer, false otherwise.

Parameters

- `[in] writer_guid`: GUID of the writer the submessage is directed to.
- `[in] reader_guid`: GUID of the reader originating the submessage.
- `[in] ack_count`: Count field of the submessage.
- `[in] seq_num`: Sequence number field of the submessage.
- `[in] fragments_state`: Fragment number bitmap field of the submessage.
- `[out] result`: true if the writer could process the submessage. Only valid when returned value is true.

const LivelinessQosPolicyKind &**get_liveliness_kind**() **const**

A method to retrieve the liveliness kind.

Return Liveliness kind

const *Duration_t* &**get_liveliness_lease_duration**() **const**

A method to retrieve the liveliness lease duration.

Return Lease duration

const *Duration_t* &**get_liveliness_announcement_period**() **const**

A method to return the liveliness announcement period.

Return The announcement period

bool **destinations_have_changed**() **const override**

Check if the destinations managed by this sender interface have changed.

Return true if destinations have changed, false otherwise.

GuidPrefix_t **destination_guid_prefix**() **const override**
Get a GUID prefix representing all destinations.

Return When all the destinations share the same prefix (i.e. belong to the same participant) that prefix is returned. When there are no destinations, or they belong to different participants, c_GuidPrefix_Unknown is returned.

const std::vector<*GuidPrefix_t*> &**remote_participants**() **const override**
Get the GUID prefix of all the destination participants.

Return a const reference to a vector with the GUID prefix of all destination participants.

const std::vector<*GUID_t*> &**remote_guids**() **const override**
Get the GUID of all destinations.

Return a const reference to a vector with the GUID of all destinations.

bool **send**(*CDRMessage_t* *message, std::chrono::steady_clock::time_point &max_blocking_time_point) **const override**
Send a message through this interface.

Parameters

- message: Pointer to the buffer with the message already serialized.
- max_blocking_time_point: Future timepoint where blocking send should end.

bool **is_datasharing_compatible**() **const**

Return Whether the writer is data sharing compatible or not

Public Members

LivelinessLostStatus **liveliness_lost_status_**
Liveliness lost status of this writer.

WriterListener

class eprosima::fastrtps::rtps::**WriterListener**
Class *WriterListener* with virtual method so the user can implement callbacks to certain events.

Public Functions

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.

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.

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.

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*.

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

6.30.3 LOG

Data Distribution Service (DDS) Data-Centric Publish-Subscribe (DCPS) Platform Independent Model (PIM) API

Colors

A collection of macros for ease the stream coloring.

Color Blue

C_BLUE

Color Bright

C_BRIGHT

Color Bright Blue

C_B_BLUE

Color Bright Cyan

C_B_CYAN

Color Bright Green

C_B_GREEN

Color Bright Magenta

C_B_MAGENTA

Color Bright Red

C_B_RED

Color Bright White

C_B_WHITE

Color Bright Yellow

C_B_YELLOW

Color Cyan

C_CYAN

Color Def

C_DEF

Color Green

C_GREEN

Color Magenta

C_MAGENTA

Color Red

C_RED

Color White

C_WHITE

Color Yellow

C_YELLOW

FileConsumer

class eprosima::fastdds::dds::**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 eprosima::fastdds::dds::**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

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.

void **ClearConsumers**()

Removes all registered consumers, including the default stdout.

void **ReportFilenames**(bool)

Enables the reporting of filenames in log entries. Disabled by default.

void **ReportFunctions**(bool)

Enables the reporting of function names in log entries. Enabled by default when supported.

void **SetVerbosity**(*Log*::*Kind*)

Sets the verbosity level, allowing for messages equal or under that priority to be logged.

Log::*Kind* **GetVerbosity**()

Returns the current verbosity level.

void **SetCategoryFilter**(**const** std::regex&)

Sets a filter that will pattern-match against log categories, dropping any unmatched categories.

void **SetFilenameFilter**(**const** std::regex&)

Sets a filter that will pattern-match against filenames, dropping any unmatched categories.

void **SetErrorStringFilter**(**const** std::regex&)

Sets a filter that will pattern-match against the provided error string, dropping any unmatched categories.

void **Reset**()
Returns the logging engine to configuration defaults.

void **Flush**()
Waits until no more log info is available.

void **KillThread**()
Stops the logging thread. It will re-launch on the next call to a successful log macro.

void **QueueLog**(**const** std::string &*message*, **const** *Log*::*Context*&, *Log*::*Kind*)
Not recommended to call this method directly! Use the following macros:

- *logInfo(cat, msg)*;
- *logWarning(cat, msg)*;
- *logError(cat, msg)*;

struct Context

struct Entry

LogConsumer

class LogConsumer
Consumes a log entry to output it somewhere.

Subclassed by *eprosima::fastdds::dds::OStreamConsumer*

logError

logError(*cat*, *msg*)
Logs an error. Disable reporting through define LOG_NO_ERROR.

logInfo

logInfo(*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.

logWarning

logWarning(*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 eprosima::fastdds::dds::**StdoutErrConsumer** : **public** eprosima::fastdds::dds::*OStreamConsumer*

Public Functions

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.

Log::*Kind* **stderr_threshold**() **const**
Retrieve the stderr_threshold.

Return The *Log::Kind* to which stderr_threshold is set.

Public Static Attributes

constexpr *Log*::*Kind* **STDERR_THRESHOLD_DEFAULT** = *Log*::*Kind*::*Warning*
Default value of stderr_threshold.

6.30.4 Statistics

ePrisima Fast DDS Statistics Module extension API.

DomainParticipant

class eprosima::fastdds::statistics::dds::**DomainParticipant** : **public** eprosima::fastdds::dds::*DomainPar*

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.

Return 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 is inconsistent, RETCODE_OK if the DataWriter has been created or if it has been created previously, and RETCODE_ERROR otherwise

Parameters

- `topic_name`: Name of the topic associated to the Statistics DataWriter
- `dwqos`: *DataWriterQos* to be set

ReturnCode_t **disable_statistics_datawriter**(**const** std::string &*topic_name*)

This operation disables a Statistics DataWriter.

Return 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

Parameters

- `topic_name`: Name of the topic associated to the Statistics DataWriter

Public Static Functions

DomainParticipant ***narrow**(eprosima::fastdds::dds::*DomainParticipant* **domain_participant*)

This operation narrows the DDS *DomainParticipant* to the Statistics *DomainParticipant*.

Return Reference to the Statistics *DomainParticipant* if successful. nullptr otherwise.

Parameters

- `domain_participant`: Reference to the DDS *DomainParticipant*

const *DomainParticipant* ***narrow**(**const** eprosima::fastdds::dds::*DomainParticipant* **domain_participant*)

This operation narrows the DDS *DomainParticipant* to the Statistics *DomainParticipant*.

Return Constant reference to the Statistics *DomainParticipant* if successful. nullptr otherwise.

Parameters

- `domain_participant`: Constant reference to the DDS *DomainParticipant*

DataWriterQos

class `eprosima::fastdds::statistics::dds::`**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_DATAW**
Constant to access default Statistics DataWriter Qos.

DataReaderQos

class `eprosima::fastdds::statistics::dds::`**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_DATA**
Constant to access default Statistics DataReader Qos.

Topic names

constexpr const char *`eprosima::fastdds::statistics::`**HISTORY_LATENCY_TOPIC** = "_fastdds_statistics_histor
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_netwo
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_statis
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_stati
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_dat
Statistics topic that reports the number of DATA/DATAFRAG sub-messages resent.

constexpr const char *eprosima::fastdds::statistics::**HEARTBEAT_COUNT_TOPIC** = "_fastdds_statistics_heartb
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_
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_nackfra
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_ent
Statistics topic that reports when new entities are discovered.

constexpr const char *eprosima::fastdds::statistics::**SAMPLE_DATAS_TOPIC** = "_fastdds_statistics_sample_da
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_
Statistics topic that reports the host, user and process where the module is running.

6.31 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 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*).

For installing *Fast DDS-Gen*, please refer to *Linux installation of Fast DDS-Gen* or to *Window installation of Fast DDS-Gen*.

6.32 Usage

This section explains the usage of *Fast DDS-Gen* tool and briefly describes the generated files.

6.32.1 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:

```
$ fastrtpsgen
```

- Windows:

```
> fastrtpsgen.bat
```

Note: In case the PATH has not been modified, these scripts can be found in the `<fastrtpsgen_directory>/scripts` directory.

6.32.2 Supported options

The expected argument list of the application is:

```
fastrtpsgen [<options>] <IDL file> [<IDL file> ...]
```

Where the option choices are:

Option	Description
-help	Shows the help information.
-version	Shows the current version of eProsima *Fast DDS-Gen*.
-d <directory>	Sets the output directory where the generated files are created.
-I <directory>	Add directory to preprocessor include paths.
-t <directory>	Sets a specific directory as a temporary directory.
-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.
-replace	Replaces the generated source code files even if they exist.
-ppDisable	Disables the preprocessor.
-ppPath	Specifies the preprocessor path.
-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

Please refer to *Dynamic Topic Types* for more information on TypeObject representation.

6.33 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.

6.33.1 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.

6.33.2 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.

6.33.3 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.

```
.
└── workspace_DDSHelloWorld
    ├── build
    │   ├── CMakeCache.txt
    │   ├── CMakeFiles
    │   ├── cmake_install.cmake
    │   ├── HelloWorld
    │   ├── libHelloWorld_lib.a
    │   └── Makefile
    ├── CMakeLists.txt
    ├── HelloWorld.cxx
    ├── HelloWorld.h
    ├── HelloWorld.idl
    ├── 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
```

6.33.4 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`.

6.33.5 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 publisher and subscriber application*.

6.33.6 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/fastrtpsgen/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/fastrtpsgen/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:

```
fastrtpsgen.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/fastrtpsgen
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 receiver 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.

6.33.7 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.

6.34 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.

6.34.1 Supported IDL types

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>

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();
    ~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;
};

class ChildStruct : public ParentStruct
{
    long child_member;
};
```

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);

    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();
```

(continues on next page)

(continued from previous page)

```
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, int> 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 `int`, 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;
```

(continues on next page)

(continued from previous page)

```
};

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.

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.

6.34.2 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.

```
#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.

6.34.3 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.
@autoid	Unimplemented.
@optional	Unimplemented.
@extensibility	Unimplemented.
@final	Unimplemented.
@appendable	Unimplemented.
@mutable	Unimplemented.
@position	Used by *bitmasks*.
@value	Allows to set a constant value to any element.
@key	Alias for eProsima's @Key annotation.
@must_understand	Unimplemented.
@default_literal	Allows selecting one member as the default within a collection.
@default	Allows specifying the default value of the annotated element.
@range	Unimplemented.
@min	Unimplemented.
@max	Unimplemented.
@unit	Unimplemented.
@bit_bound	Allows setting a size to a *bitmasks*.
@external	Unimplemented.
@nested	Unimplemented.
@verbatim	Unimplemented.
@service	Unimplemented.
@oneway	Unimplemented.
@ami	Unimplemented.
@non_serialized	The annotated member will be omitted from serialization.

Most unimplemented annotations are related to Extended Types.

6.34.4 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;
};

union ForwardUnion switch (long)
{
    case 0:
        ForwardStruct fw_struct;
    default:
        string empty;
};
```

6.34.5 IDL 4.2 aliases

IDL 4.2 allows using the following names for primitive types:

int8
uint8
int16
uint16
int32
uint32
int64
uint64

6.34.6 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.
};
```

6.35 CLI

The *Fast DDS* command line interface provides a set commands and sub-commands to perform, *Fast DDS* related, maintenance and configuration tasks.

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 two 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*

6.35.1 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*).

How to run

On a shell, execute:

```
fastdds discovery -i {0-255} [optional parameters]
```

Where the parameters are:

Option	Description
-i --server-id	**Mandatory** unique server identifier. Specifies zero based server position in ROS_DISCOVERY_SERVER environment variable. Must be an integer in range [0, 255]
-h -help	Produce help message.
-l --ip-address	IP address chosen to listen the clients. Defaults to any (0.0.0.0).
-p --port	UDP port chosen to listen the clients. Defaults to '11811'.
-b --backup	Creates a BACKUP *server* (see *Discovery Protocol*)

The output is:

```
### Server is running ###
  Participant Type:   <SERVER|BACKUP>
  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>
                      UDPv4:[<ip-address>]:<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*)

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
     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
   ```

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
     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
   ```

3. 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
  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
```

4. 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
  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
```

6.35.2 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.

6.36 Version 2.3.1

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*.

6.37 Previous versions

6.37.1 Version 2.3.0

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.
- 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*.

6.37.2 Version 2.2.0

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.
- 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*.

6.37.3 Version 2.1.0

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.
- 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 *fastrtpsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.37.4 Version 2.0.2

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 *fastrtpsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.37.5 Version 2.0.1

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 *fastrtpsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.37.6 Version 2.0.0

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

6.37.7 Version 1.10.0

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 *fastrtpsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.37.8 Version 1.9.4

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 *fastrtpsgen*.

6.37.9 Version 1.9.3

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 *fastrtpsgen*.

6.37.10 Version 1.9.2

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 *fastrtpsgen*.

6.37.11 Version 1.9.1

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 *fastrtpsgen*.

6.37.12 Version 1.9.0

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 *fastrtpsgen*.

6.37.13 Version 1.8.4

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 *fastrtpsgen*

6.37.14 Version 1.8.3

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 *fastrtpsgen*

6.37.15 Version 1.8.2

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 *fastrtpsgen* 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 *fastrtpsgen*

6.37.16 Version 1.8.1

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 fastrtpsgen.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 *fastrtpsgen*.

6.37.17 Version 1.8.0

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 *fastrtpsgen*.

6.37.18 Version 1.7.2

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 *fastrtpsgen*.

6.37.19 Version 1.7.1

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 *fastrtpsgen*.

6.37.20 Version 1.7.0

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 *fastrtpsgen*.

6.37.21 Version 1.6.0

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 *fastrtpsgen*.

6.37.22 Version 1.5.0

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 *fastrtpsgen*.

6.37.23 Version 1.4.0

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 *fastrtpsgen*.

6.37.24 Version 1.3.1

This release included the following:

- New examples that illustrate how to tweak Fast RTPS towards different applications.
- Improved support for embedded Linux.
- Bug fixing.

6.37.25 Version 1.3.0

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.

6.37.26 Version 1.2.0

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

B

C

D

E

F

L

P

R

T