
Fast DDS Documentation

Release 3.0.0

eProsima

May 14, 2024

INTRODUCTION

1	Fast DDS-Gen	3
2	RTPS Wire Protocol	5
3	Main Features	7
4	Dependencies and compatibilities	9
5	Contacts and Commercial support	11
6	Contributing to the documentation	13
7	Structure of the documentation	15
7.1	DDS API	16
7.2	Fast DDS-Gen	16
7.3	RTPS Wire Protocol	16
7.4	Main Features	17
7.5	Dependencies and compatibilities	18
7.6	Contacts and Commercial support	18
7.7	Contributing to the documentation	18
7.8	Structure of the documentation	18
7.9	Linux installation from binaries	19
7.10	Windows installation from binaries	20
7.11	Linux installation from sources	22
7.12	Windows installation from sources	32
7.13	Mac OS installation from sources	43
7.14	QNX 7.1 installation from sources	49
7.15	CMake options	54
7.16	Getting Started	58
7.17	Library Overview	94
7.18	DDS Layer	99
7.19	RTPS Layer	267
7.20	Discovery	275
7.21	Transport Layer	306
7.22	Persistence Service	358
7.23	Security	363
7.24	Logging	389
7.25	XML profiles	398
7.26	Environment variables	480
7.27	PropertyPolicyQos Options	486
7.28	Statistics Module	498

7.29	XTypes	514
7.30	Typical Use-Cases	551
7.31	ROS 2 using Fast DDS middleware	619
7.32	Troubleshooting	633
7.33	C++ API Reference	634
7.34	Python API Reference	698
7.35	Introduction	703
7.36	Usage	704
7.37	Building a publish/subscribe application	705
7.38	Building Python auxiliary libraries	710
7.39	Defining a data type via IDL	711
7.40	CLI	722
7.41	Docker Images	730
7.42	Dependencies and compatibilities	736
7.43	Version 2.14.1 (latest)	741
7.44	Supported versions	742
7.45	Previous end-of-life versions	763

Warning: In preparation for v3.0.0, Fast DDS’ master branch is undergoing major changes entailing **API breaks**. Until Fast DDS v3.0.0 is released, it is strongly advisable to use the latest stable version, 2.14.



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

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

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

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

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

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

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

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

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.

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.

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

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

DEPENDENCIES AND COMPATIBILITIES

eProsima Fast DDS is continuously evolving and improving. This means that the different software products that are part of the Fast DDS ecosystem are evolving and improving together with Fast DDS.

Fast DDS has some *library dependencies*, e.g. the previously mentioned Fast CDR for data serialization, or OpenSSL for secure communications. Depending on different *platform support levels*, it has also different *build dependencies*.

Finally, there are some other *eProsima* products that use Fast DDS as a middleware, such as *Micro XRCE-DDS*, *DDS Router* and *Fast DDS python* wrapper. Those that are strongly attached to each Fast DDS supported version are described in *this product compatibility table*.

CONTACTS AND COMMERCIAL SUPPORT

Find more about us at [eProsimas webpage](#).

Support available at:

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

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.

STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the sections below.

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

Warning: In preparation for v3.0.0, Fast DDS' master branch is undergoing major changes entailing **API breaks**. Until Fast DDS v3.0.0 is released, it is strongly advisable to use the latest stable version, 2.14.



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

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

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

7.1 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.

7.2 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.

7.3 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.

7.4 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 constraints.
- **Built-in Discovery Server.** *eProsima Fast DDS* is based on the dynamical discovery of existing publishers and subscribers, and performs this task continuously without the need to contacting or setting any servers. However, a Client-Server discovery as well as other discovery paradigms can also be configured.
- **Sync and Async publication modes.** *eProsima Fast DDS* supports both synchronous and asynchronous data publication.
- **Best effort and reliable communication.** *eProsima Fast DDS* supports an optional reliable communication paradigm over *Best Effort* communications protocols such as UDP. Furthermore, another way of setting a reliable communication is to use our TCP transport.
- **Transport layers.** *eProsima Fast DDS* implements an architecture of pluggable transports. The current version implements five transports: UDPv4, UDPv6, TCPv4, TCPv6 and SHM (shared memory).
- **Security.** *eProsima Fast DDS* can be configured to provide secure communications. For this purpose, it implements pluggable security at three levels: authentication of remote participants, access control of entities and encryption of data.
- **Statistics Module.** *eProsima Fast DDS* can be configured to gather and provide information about the data being exchanged by the user application.
- **Flow controllers.** We support user-configurable flow controllers, that can be used to limit the amount of data to be sent under certain conditions.
- **Plug-and-play Connectivity.** New applications and services are automatically discovered, and can join and leave the network at any time without the need for reconfiguration.
- **Scalability and Flexibility.** DDS builds on the concept of a global data space. The middleware is in charge of propagating the information between publishers and subscribers. This guarantees that the distributed network is adaptable to reconfigurations and scalable to a large number of entities.
- **Application Portability.** The DDS specification includes a platform specific mapping to IDL, allowing an application using DDS to switch among DDS implementations with only a re-compile.
- **Extensibility.** *eProsima Fast DDS* allows the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- **Configurability and Modularity.** *eProsima Fast DDS* provides an intuitive way to be configured, either through code or XML profiles. Modularity allows simple devices to implement a subset of the protocol and still participate in the network.
- **High performance.** *eProsima Fast DDS* uses a static low-level serialization library, [Fast CDR](#), a C++ library that serializes according to the standard CDR serialization mechanism defined in the [RTPS Specification](#) (see the Data Encapsulation chapter as a reference).
- **Easy to use.** The project comes with an out-of-the-box example, the *DDSHelloWorld* (see [Getting Started](#)) that puts into communication a publisher and a subscriber, showcasing how *eProsima Fast DDS* is deployed. Additionally, the interactive demo *ShapesDemo* is available for the user to dive into the DDS world. The DDS and the RTPS layers are thoroughly explained in the [DDS Layer](#) and [RTPS Layer](#) sections.
- **Low resources consumption.** *eProsima Fast DDS*:
 - Allows to preallocate resources, to minimize dynamic resource allocation.

- Avoids the use of unbounded resources.
- Minimizes the need to copy data.
- **Multi-platform.** The OS dependencies are treated as pluggable modules. Users may easily implement platform modules using the *eProsima Fast DDS* library on their target platforms. By default, the project can run over Linux, Windows and MacOS.
- **Free and Open Source.** The Fast DDS library, the underneath RTPS library, the generator tool, the internal dependencies (such as *eProsima Fast CDR*) and the external ones (such as the *foonathan* library) are free and open source.

7.5 Dependencies and compatibilities

eProsima Fast DDS is continuously evolving and improving. This means that the different software products that are part of the Fast DDS ecosystem are evolving and improving together with Fast DDS.

Fast DDS has some *library dependencies*, e.g. the previously mentioned Fast CDR for data serialization, or OpenSSL for secure communications. Depending on different *platform support levels*, it has also different *build dependencies*.

Finally, there are some other *eProsima* products that use Fast DDS as a middleware, such as *Micro XRCE-DDS*, *DDS Router* and *Fast DDS python* wrapper. Those that are strongly attached to each Fast DDS supported version are described in *this product compatibility table*.

7.6 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

7.7 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.

7.8 Structure of the documentation

This documentation is organized into the sections below.

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

7.9 Linux installation from binaries

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

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

7.9.1 Install

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

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

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

To use the *xml* validation tool, please refer to the [Linux installation from sources](#) page.

Contents

The `src` folder contains the following packages:

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

See also:

For further information about Fast DDS dependencies, as well as for the corresponding versions of other related products, please refer to the Fast DDS [Dependencies and compatibilities](#) section.

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

Run an application

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

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

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

- Add it permanently to the PATH by executing:

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

Including Fast-DDS in a CMake project

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

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

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

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

7.9.2 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.

7.10 Windows installation from binaries

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

- *Requirements*
 - *Visual Studio*
- *Install*
 - *Contents*
 - *Environment variables*

– *Including Fast-DDS in a CMake project*

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

7.10.1 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.

7.10.2 Install

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

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

To use the *xml* validation tool, please refer to the [Windows installation from sources](#) page.

Contents

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

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

See also:

For further information about Fast DDS dependencies, as well as for the corresponding versions of other related products, please refer to the Fast DDS [Dependencies and compatibilities](#) section.

Environment variables

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

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

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

Including Fast-DDS in a CMake project

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

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

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

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

2. Through the custom CMake variable *fastdds_SHARED_LIBS*.

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

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

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

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

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

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

7.11 Linux installation from sources

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

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

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

7.11.1 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.
- `fastdds`, the core library of *eProsima Fast DDS* library.

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

Requirements

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

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

CMake, g++, pip3, wget and git

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

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

Dependencies

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

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

See also:

For further information about this Fast DDS version dependencies, as well as for the corresponding versions of other related products, please refer to the Fast DDS [Library dependencies](#) section.

Asio and TinyXML2 libraries

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

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

OpenSSL

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

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

Libp11 and SoftHSM libraries

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

Install [libp11](#) using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

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

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

Install [SoftHSM](#) using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install softhsm2
```

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

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

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

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

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

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

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

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

```
openssl engine pkcs11 -t
```

Gtest

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

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

XML validation tool

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

For more details, please refer to the [xml](#) section.

Install the [xmllschema](#) dependency to be able to use this optional tool.

Colcon installation

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

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

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

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

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

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

3. Build the packages:

```
colcon build
```

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

Run an application

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

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

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

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

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

CMake installation

This section explains how to compile *eProsima Fast DDS* with [CMake](#), either *locally* or *globally*.

Local installation

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

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

2. Clone the following dependencies and compile them using [CMake](#).

- [Foonathan memory](#)

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

- [Fast CDR](#)

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

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

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

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

Global installation

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

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

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

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

Run an application

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

- Prepare the environment locally by typing the command:

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

- Add it permanently to the PATH, by typing:

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

7.11.2 Fast DDS Python bindings installation

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

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

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

Requirements

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

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

SWIG

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

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

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

```
sudo apt install swig
```

Header files and static library for Python

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

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

Dependencies

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

- *Fast DDS dependencies*

Colcon installation

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

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

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

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

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


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

3. Build the packages:

```
colcon build
```

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

Run an application

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

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

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

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

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

CMake installation

This section explains how to compile *Fast DDS Python bindings* with [CMake](#), either *locally* or *globally*.

Local installation

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

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

2. Clone the following dependencies and compile them using [CMake](#).

- [Foonathan memory](#)

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

- [Fast CDR](#)

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

- Fast DDS

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

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

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

Global installation

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

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

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

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

Run an application

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

- Prepare the environment locally by typing the command:

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

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

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

7.11.3 Fast DDS-Gen installation

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

Requirements

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

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

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

See also:

For further information about Fast DDS-Gen product related versions, please refer to the [Library dependencies](#) section.

Java JDK

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

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

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

Compiling Fast DDS-Gen

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

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

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

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

```
gradle assemble
```

Contents

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

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

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

7.12 Windows installation from sources

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

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

7.12.1 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.
- `fastdds`, the core library of *eProsima Fast DDS* library.

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

Requirements

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

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

Visual Studio

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

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

Chocolatey

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

CMake, pip3, wget and git

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

Gtest

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

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

and add next argument to the *colcon* call

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

XML validation tool

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

For more details, please refer to the [xml](#) section.

Install the [xmlschema](#) dependency to be able to use this optional tool.

Dependencies

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

- [Asio and TinyXML2 libraries](#)
- [OpenSSL](#)

See also:

For further information about this Fast DDS version dependencies, as well as for the corresponding versions of other related products, please refer to the Fast DDS [Library dependencies](#) section.

Asio and TinyXML2 libraries

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

- [Asio](#)
- [TinyXML2](#)

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

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

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

OpenSSL

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

```
choco install -y openssl
```

Libp11 and SoftHSM libraries

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

Download the latest [libp11](#) version for Windows from this [repository](#) and follow the [installation instructions](#)

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

Download the [SoftHSM](#) for Windows installer from this [repository](#). Execute the installer and follow the installation instructions.

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

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

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

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

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

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

```
openssl_conf = openssl_init
```

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

```
[openssl_init]
    engines = engine_section

[engine_section]
    pkcs11 = pkcs11_section

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

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

```
openssl engine pkcs11 -t
```

Colcon installation

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

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

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

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

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

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

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

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

Finally, use `colcon` to compile all software:

```
colcon build
```

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

Run an application

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

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

```
setup.bat
```

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

CMake installation

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

Local installation

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

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

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

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

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

- **Fast CDR**

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

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

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

Global installation

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

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

Run an application

When running an instance of an application using *eProxima 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\fastdds
- *Fast CDR*: C:\Program Files\fastcdr

7.12.2 Fast DDS Python bindings installation

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

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

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

Requirements

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

- *Fast DDS requirements*
- *SWIG*

SWIG

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

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

Dependencies

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

- *Fast DDS dependencies*

Colcon installation

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

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

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

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

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

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

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

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

3. Build the packages:

```
colcon build
```

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

Run an application

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

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

```
setup.bat
```

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

CMake installation

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

Local installation

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

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

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

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

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

- **Fast CDR**

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

- **Fast DDS**

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

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

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

Global installation

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

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

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

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

Run an application

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

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

7.12.3 Fast DDS-Gen installation

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

Requirements

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

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

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

See also:

For further information about Fast DDS-Gen product related versions, please refer to the [Library dependencies](#) section.

Java JDK

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

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

Compiling Fast DDS-Gen

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

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

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

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

```
gradle assemble
```

Contents

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

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

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

7.13 Mac OS installation from sources

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

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

7.13.1 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.
- `fastdds`, the core library of *eProsima Fast DDS* library.

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

Requirements

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

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

Homebrew

Homebrew is a macOS package manager, it is needed to install some of *eProxima 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 *eProxima 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, *eProxima Fast DDS* does not compile tests. It is possible to activate them with the opportune [CMake configuration options](#) when calling [colcon](#) or [CMake](#). For more details, please refer to the [CMake options](#) section. Also add the [Gtest repository](#) into the workspace directory.

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

XML validation tool

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

For more details, please refer to the [xml](#) section.

Install the [xmlschema](#) dependency to be able to use this optional tool.

Dependencies

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

- *Asio and TinyXML2 libraries*
- *OpenSSL*

See also:

For further information about this Fast DDS version dependencies, as well as for the corresponding versions of other related products, please refer to the Fast DDS *Library dependencies* section.

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/fastdds.repos
mkdir src
vcs import src < fastdds.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
```

7.13.2 Fast DDS-Gen installation

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

Requirements

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

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

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

See also:

For further information about Fast DDS-Gen product related versions, please refer to the [Library dependencies](#) section.

Java JDK

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

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

Compiling Fast DDS-Gen

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

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

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

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

```
gradle assemble
```

Contents

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

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

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

7.14 QNX 7.1 installation from sources

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

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

7.14.1 Fast DDS library installation

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

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

See also:

For further information about Fast DDS library dependencies, as well as for the corresponding versions of other related products, please refer to the Fast DDS [Library dependencies](#) section.

The [Requirements](#) detailed below needs to be met first.

Requirements

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

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

QNX SDP 7.1

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

QNX SDP Installation Guide

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

See also:

For further information about Fast DDS build system dependencies regarding QNX 7.1, please refer to the Fast DDS *Build system dependencies* section.

CMake, pip3, git, dos2unix, and automake

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

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

Build and Installation

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

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

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

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

```
cd $WORKSPACE

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

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

(continues on next page)

(continued from previous page)

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

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

3. Get foonathan_memory_vendor:

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

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

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

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

5. Source the QNX environment script:

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

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

Note:

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

Note:

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

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

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

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

7.14.2 Run Examples and Tests on a QNX 7.1 Target

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

Move Libraries, Examples, and Tests to the QNX Target

1. Move the built libraries to the QNX target:

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

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

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

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

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

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

2. Move examples and tests to the QNX target:

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

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


Run DDSHelloWorldExample

1. Open a terminal and run a subscriber:

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

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

2. Open another terminal and run a publisher:

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

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

The following output will be shown in the subscriber terminal:

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

The following output will be shown for the publisher:

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

Run a Test

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

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

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

The following test output for EntityTests will be shown:

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

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

7.15 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.

7.15.1 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: <ul style="list-style-type: none"> 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 <i>Fast DDS</i> builds. It is set to ON if EPROSIMA_INSTALLER is ON. Setting EPROSIMA_BUILD to ON has the following effects on other options: <ul style="list-style-type: none"> INTERNAL_DEBUG is set to ON. COMPILE_EXAMPLES is set to ON if EPROSIMA_INSTALLER is OFF. THIRDPARTY_fastcd 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_TinyXML is set to ON if it was not set to FORCE. THIRDPARTY_android is set to ON if it was not set to FORCE. EPROSIMA_BUILD_TINYXML is set to ON if EPROSIMA_INSTALLER is OFF. 	ON OFF	OFF
7.15. CMake options			55
BUILD_SHARED_LIBS	Builds internal libraries	ON OFF	ON

7.15.2 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	De-fault
LOG_CONSUMER	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 <i>Fast DDS</i> is built in debug mode for Single-Config generators, the default value will be OFF.	ON OFF	ON
FASTDDS_ENABLE_INFO	Enables Info Log level even on non Debug configurations. This option only takes action if LOG_NO_INFO is set to OFF (see Disable Logging Module). Mind that this may entail a significant performance hit.	ON OFF	OFF
LOG_NO_WARNING	Deactivates Warning Log level.	ON OFF	OFF
LOG_NO_ERROR	Deactivates Error Log level.	ON OFF	OFF
INTERNAL_DEBUG	Activates compilation of log messages (See Disable Logging Module). Moreover, INTERNAL_DEBUG is set to ON if EPROSIMA_BUILD is ON.	ON OFF	OFF
ENABLE_OLD_LOG_MACROS	Enable old log macros (See Old Log macros disable).	ON OFF	ON

7.15.3 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	Pos- si- ble val- ues	De- fault
THIRDP	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 FORC	OFF
THIRDP	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 FORC	OFF
THIRDP	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 FORC	OFF
THIRDP	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 FORC	OFF
THIRDP	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 FORC	OFF
THIRDP	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.

7.15.4 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
FASTDDS_PI	Enables the building of black-box tests for the verification of DDS communications using the <i>Fast DDS</i> DDS-layer API.	ON OFF	OFF
FASTDDS_EX	Enables the building of example tests for the verification of the Fast DDS examples	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	Activates the building of profiling tests using Valgrind.	ON OFF	OFF
EPROSIMA_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
INSTALL_ANDROID	Android cross-compilation only. Marks the tests for installation on the connected device/emulator.	ON OFF	OFF
ANDROID_TEST_PATH	Android cross-compilation only. Path on the Android device/emulator to use for installing and running the tests.	Valid Unix filesystem path string	""

7.16 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.

7.16.1 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.

7.16.2 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.

7.16.3 Writing a simple C++ publisher and subscriber application

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

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

Background

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

Prerequisites

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

Create the application workspace

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

```

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

```

Let's create the directory tree first.

```

mkdir workspace_DDShelloWorld && cd workspace_DDShelloWorld
mkdir src build

```

Import linked libraries and its dependencies

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

Installation from binaries and manual installation

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

Colcon installation

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

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

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

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

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

Configure the CMake project

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

```
cmake_minimum_required(VERSION 3.20)

project(DDSHelloWorld)

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

if(NOT fastdds_FOUND)
    find_package(fastdds 3 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()

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

```

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

Build the topic data type

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

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

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

In the workspace directory, execute the following commands:

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

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

```

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

```

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

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

This must have generated the following files:

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

```
target_link_libraries(DDSHelloWorldPublisher fastdds fastcdr)
```

Write the Fast DDS publisher

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

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

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

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

(continues on next page)

(continued from previous page)

```

35 {
36 private:
37
38     HelloWorld hello_;
39
40     DomainParticipant* participant_;
41
42     Publisher* publisher_;
43
44     Topic* topic_;
45
46     DataWriter* writer_;
47
48     TypeSupport type_;
49
50     class PubListener : public DataWriterListener
51     {
52     public:
53
54         PubListener()
55             : matched_(0)
56         {
57         }
58
59         ~PubListener() override
60         {
61         }
62
63         void on_publication_matched(
64             DataWriter*,
65             const PublicationMatchStatus& info) override
66         {
67             if (info.current_count_change == 1)
68             {
69                 matched_ = info.total_count;
70                 std::cout << "Publisher matched." << std::endl;
71             }
72             else if (info.current_count_change == -1)
73             {
74                 matched_ = info.total_count;
75                 std::cout << "Publisher unmatched." << std::endl;
76             }
77             else
78             {
79                 std::cout << info.current_count_change
80                 << " is not a valid value for PublicationMatchStatus current_
81 ↪count change." << std::endl;
82             }
83
84             std::atomic_int matched_;
85

```

(continues on next page)

(continued from previous page)

```

86     } listener_;
87
88 public:
89
90     HelloWorldPublisher()
91         : participant_(nullptr)
92         , publisher_(nullptr)
93         , topic_(nullptr)
94         , writer_(nullptr)
95         , type_(new HelloWorldPubSubType())
96     {
97     }
98
99     virtual ~HelloWorldPublisher()
100    {
101        if (writer_ != nullptr)
102        {
103            publisher_>delete_datawriter(writer_);
104        }
105        if (publisher_ != nullptr)
106        {
107            participant_>delete_publisher(publisher_);
108        }
109        if (topic_ != nullptr)
110        {
111            participant_>delete_topic(topic_);
112        }
113        DomainParticipantFactory::get_instance()->delete_participant(participant_);
114    }
115
116    ///Initialize the publisher
117    bool init()
118    {
119        hello_.index(0);
120        hello_.message("HelloWorld");
121
122        DomainParticipantQos participantQos;
123        participantQos.name("Participant_publisher");
124        participant_ = DomainParticipantFactory::get_instance()->create_participant(0,
125        ↪ participantQos);
126
127        if (participant_ == nullptr)
128        {
129            return false;
130        }
131
132        // Register the Type
133        type_.register_type(participant_);
134
135        // Create the publications Topic
136        topic_ = participant_>create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_
137        ↪ DEFAULT);

```

(continues on next page)

(continued from previous page)

```

136     if (topic_ == nullptr)
137     {
138         return false;
139     }
140
141     // Create the Publisher
142     publisher_ = participant_>create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);
143
144     if (publisher_ == nullptr)
145     {
146         return false;
147     }
148
149     // Create the DataWriter
150     writer_ = publisher_>create_datawriter(topic_, DATAWRITER_QOS_DEFAULT, &
151 ↪ listener_);
152
153     if (writer_ == nullptr)
154     {
155         return false;
156     }
157     return true;
158 }
159
160 //!Send a publication
161 bool publish()
162 {
163     if (listener_.matched_ > 0)
164     {
165         hello_.index(hello_.index() + 1);
166         writer_>write(&hello_);
167         return true;
168     }
169     return false;
170 }
171
172 //!Run the Publisher
173 void run(
174     uint32_t samples)
175 {
176     uint32_t samples_sent = 0;
177     while (samples_sent < samples)
178     {
179         if (publish())
180         {
181             samples_sent++;
182             std::cout << "Message: " << hello_.message() << " with index: " << hello_
183 ↪ .index()
184                 << " SENT" << std::endl;
185         }
186         std::this_thread::sleep_for(std::chrono::milliseconds(1000));

```

(continues on next page)

(continued from previous page)

```

186     }
187 }
188 };
189
190 int main(
191     int argc,
192     char** argv)
193 {
194     std::cout << "Starting publisher." << std::endl;
195     uint32_t samples = 10;
196
197     HelloWorldPublisher* mypub = new HelloWorldPublisher();
198     if(mypub->init())
199     {
200         mypub->run(samples);
201     }
202
203     delete mypub;
204     return 0;
205 }

```

Examining the code

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

```

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

```

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

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

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

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

```
#include <chrono>
#include <thread>
```

(continues on next page)

(continued from previous page)

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

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

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

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

```
class HelloWorldPublisher
```

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

```
private:

    HelloWorld hello_;

    DomainParticipant* participant_;

    Publisher* publisher_;

    Topic* topic_;

    DataWriter* writer_;

    TypeSupport type_;
```

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

```
class PubListener : public DataWriterListener
{
public:

    PubListener()
        : matched_(0)
    {
    }
}
```

(continues on next page)

(continued from previous page)

```

~PubListener() override
{
}

void on_publication_matched(
    DataWriter*,
    const PublicationMatchedException& 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 PublicationMatchedException 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)
    {

```

(continues on next page)

(continued from previous page)

```

        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);

    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;
    }
}

```

(continues on next page)

(continued from previous page)

```

}

// 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()

```

(continues on next page)

(continued from previous page)

```

        << " SENT" << std::endl;
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
}
}

```

Finally, the HelloWorldPublisher is initialized and run in main.

```

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

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

    delete mypub;
    return 0;
}

```

CMakeLists.txt

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

```

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

```

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

```

cmake ..
cmake --build .
./DDSHelloWorldPublisher

```

Write the Fast DDS subscriber

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

```

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

```

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

```
1  // Copyright 2016 Proyectos y Sistemas de Mantenimiento SL (eProsima).
2  //
3  // Licensed under the Apache License, Version 2.0 (the "License");
4  // you may not use this file except in compliance with the License.
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 <chrono>
23 #include <thread>
24
25 #include <fastdds/dds/domain/DomainParticipant.hpp>
26 #include <fastdds/dds/domain/DomainParticipantFactory.hpp>
27 #include <fastdds/dds/subscriber/DataReader.hpp>
28 #include <fastdds/dds/subscriber/DataReaderListener.hpp>
29 #include <fastdds/dds/subscriber/qos/DataReaderQos.hpp>
30 #include <fastdds/dds/subscriber/SampleInfo.hpp>
31 #include <fastdds/dds/subscriber/Subscriber.hpp>
32 #include <fastdds/dds/topic/TypeSupport.hpp>
33
34 using namespace eprosima::fastdds::dds;
35
36 class HelloWorldSubscriber
37 {
38 private:
39
40     DomainParticipant* participant_;
41
42     Subscriber* subscriber_;
43
44     DataReader* reader_;
45
46     Topic* topic_;
47
48     TypeSupport type_;
49
50     class SubListener : public DataReaderListener
```

(continues on next page)

(continued from previous page)

```

51 {
52 public:
53
54     SubListener()
55         : samples_(0)
56     {
57     }
58
59     ~SubListener() override
60     {
61     }
62
63     void on_subscription_matched(
64         DataReader*,
65         const SubscriptionMatchedStatus& info) override
66     {
67         if (info.current_count_change == 1)
68         {
69             std::cout << "Subscriber matched." << std::endl;
70         }
71         else if (info.current_count_change == -1)
72         {
73             std::cout << "Subscriber unmatched." << std::endl;
74         }
75         else
76         {
77             std::cout << info.current_count_change
78                 << " is not a valid value for SubscriptionMatchedStatus_
↳current count change" << std::endl;
79         }
80     }
81
82     void on_data_available(
83         DataReader* reader) override
84     {
85         SampleInfo info;
86         if (reader->take_next_sample(&hello_, &info) ==
↳eprosima::fastdds::dds::RETCODE_OK)
87         {
88             if (info.valid_data)
89             {
90                 samples_++;
91                 std::cout << "Message: " << hello_.message() << " with index: " <<
↳hello_.index()
92                     << " RECEIVED." << std::endl;
93             }
94         }
95     }
96
97     HelloWorld hello_;
98
99     std::atomic_int samples_;

```

(continues on next page)

(continued from previous page)

```

100     }
101     listener_;
102
103
104 public:
105
106     HelloWorldSubscriber()
107         : participant_(nullptr)
108         , subscriber_(nullptr)
109         , topic_(nullptr)
110         , reader_(nullptr)
111         , type_(new HelloWorldPubSubType())
112     {
113     }
114
115     virtual ~HelloWorldSubscriber()
116     {
117         if (reader_ != nullptr)
118         {
119             subscriber_>delete_datareader(reader_);
120         }
121         if (topic_ != nullptr)
122         {
123             participant_>delete_topic(topic_);
124         }
125         if (subscriber_ != nullptr)
126         {
127             participant_>delete_subscriber(subscriber_);
128         }
129         DomainParticipantFactory::get_instance()->delete_participant(participant_);
130     }
131
132     ///Initialize the subscriber
133     bool init()
134     {
135         DomainParticipantQos participantQos;
136         participantQos.name("Participant_subscriber");
137         participant_ = DomainParticipantFactory::get_instance()->create_participant(0,
138 ↪ participantQos);
139
140         if (participant_ == nullptr)
141         {
142             return false;
143         }
144
145         // Register the Type
146         type_.register_type(participant_);
147
148         // Create the subscriptions Topic
149         topic_ = participant_>create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_
150 ↪ DEFAULT);

```

(continues on next page)

(continued from previous page)

```

150     if (topic_ == nullptr)
151     {
152         return false;
153     }
154
155     // Create the Subscriber
156     subscriber_ = participant_>create_subscriber(SUBSCRIBER_QOS_DEFAULT, nullptr);
157
158     if (subscriber_ == nullptr)
159     {
160         return false;
161     }
162
163     // Create the DataReader
164     reader_ = subscriber_>create_datareader(topic_, DATAREADER_QOS_DEFAULT, &
↳ listener_);
165
166     if (reader_ == nullptr)
167     {
168         return false;
169     }
170
171     return true;
172 }
173
174 //!Run the Subscriber
175 void run(
176     uint32_t samples)
177 {
178     while (listener_.samples_ < samples)
179     {
180         std::this_thread::sleep_for(std::chrono::milliseconds(100));
181     }
182 }
183
184 };
185
186 int main(
187     int argc,
188     char** argv)
189 {
190     std::cout << "Starting subscriber." << std::endl;
191     uint32_t samples = 10;
192
193     HelloWorldSubscriber* mysub = new HelloWorldSubscriber();
194     if (mysub->init())
195     {
196         mysub->run(samples);
197     }
198
199     delete mysub;
200     return 0;

```

(continues on next page)

}

Examining the code

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

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

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

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

(continues on next page)

(continued from previous page)

```

{
    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) == eprosima::fastdds::dds::RETCODE_OK)
    {
        if (info.valid_data)
        {
            samples_++;
            std::cout << "Message: " << hello_.message() << " with index: " << hello_.
↪index()
                        << " RECEIVED." << std::endl;
        }
    }
}

```

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

```

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

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

```

(continues on next page)

(continued from previous page)

```

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

```

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

```

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

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

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

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

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

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

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

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

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

    return true;
}

```

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

```

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

```

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

```

};

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

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

    delete mysub;
}

```

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 fastdds fastcdr)

```

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

```

cmake ..
cmake --build .
./DDSHelloWorldSubscriber

```

Putting all together

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

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

Summary

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

Next steps

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

7.16.4 Writing a simple Python publisher and subscriber application

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

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

Background

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

Prerequisites

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

Create the application workspace

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

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

Let's create the directory tree first.

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

Import linked libraries and its dependencies

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

Colcon installation

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

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

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

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

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

Build the topic data type

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

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

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

In the workspace directory, execute the following commands:

```
touch HelloWorld.idl
```

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

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

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

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

This must have generated the following files:

- HelloWorld.hpp: HelloWorld C++ type definition.
- HelloWorld.i: **SWIG** interface file for HelloWorld C++ type definition.
- HelloWorldPubSubTypes.cxx: C++ interface used by Fast DDS to support HelloWorld type.

- HelloWorldPubSubTypes.h: C++ header file for HelloWorldPubSubTypes.cxx.
- HelloWorldPubSubTypes.i: [SWIG](#) interface file for C++ Serialization and Deserialization code.
- HelloWorldCdrAux.ipp: C++ serialization and deserialization code for the HelloWorld type.
- HelloWorldCdrAux.hpp: C++ header file for HelloWorldCdrAux.ipp.
- HelloWorldTypeObjectSupport.cxx: *TypeObject representation* registration code.
- HelloWorldTypeObjectSupport.hpp: Header file for HelloWorldTypeObjectSupport.cxx.
- CMakeLists.txt: CMake file to generate C++ source code and Python module from the [SWIG](#) interface files, compile and generate C++ libraries.

After that, the python bindings can be generated by running the following command.

```
cmake .
make
```

This must have generated the python binding:

- HelloWorld.py: Python module to be imported by your Python example.

CMakeLists.txt

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

```
cmake .
make
```

Write the Fast DDS publisher

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

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

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

```
1  # Copyright 2022 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.
```

(continues on next page)

(continued from previous page)

```

14 """
15 HelloWorld Publisher
16 """
17 from threading import Condition
18 import time
19
20 import fastdds
21 import HelloWorld
22
23 DESCRIPTION = """HelloWorld Publisher example for Fast DDS python bindings"""
24 USAGE = ('python3 HelloWorldPublisher.py')
25
26 class WriterListener (fastdds.DataWriterListener) :
27     def __init__(self, writer) :
28         self._writer = writer
29         super().__init__()
30
31
32     def on_publication_matched(self, datawriter, info) :
33         if (0 < info.current_count_change) :
34             print ("Publisher matched subscriber {}".format(info.last_subscription_
↪handle))
35             self._writer._cvDiscovery.acquire()
36             self._writer._matched_reader += 1
37             self._writer._cvDiscovery.notify()
38             self._writer._cvDiscovery.release()
39         else :
40             print ("Publisher unmatched subscriber {}".format(info.last_subscription_
↪handle))
41             self._writer._cvDiscovery.acquire()
42             self._writer._matched_reader -= 1
43             self._writer._cvDiscovery.notify()
44             self._writer._cvDiscovery.release()
45
46
47 class Writer:
48
49
50     def __init__(self):
51         self._matched_reader = 0
52         self._cvDiscovery = Condition()
53         self.index = 0
54
55         factory = fastdds.DomainParticipantFactory.get_instance()
56         self.participant_qos = fastdds.DomainParticipantQos()
57         factory.get_default_participant_qos(self.participant_qos)
58         self.participant = factory.create_participant(0, self.participant_qos)
59
60         self.topic_data_type = HelloWorld.HelloWorldPubSubType()
61         self.topic_data_type.setName("HelloWorld")
62         self.type_support = fastdds.TypeSupport(self.topic_data_type)
63         self.participant.register_type(self.type_support)

```

(continues on next page)

(continued from previous page)

```

64
65     self.topic_qos = fastdds.TopicQos()
66     self.participant.get_default_topic_qos(self.topic_qos)
67     self.topic = self.participant.create_topic("HelloWorldTopic", self.topic_data_
↪type.getName(), self.topic_qos)
68
69     self.publisher_qos = fastdds.PublisherQos()
70     self.participant.get_default_publisher_qos(self.publisher_qos)
71     self.publisher = self.participant.create_publisher(self.publisher_qos)
72
73     self.listener = WriterListener(self)
74     self.writer_qos = fastdds.DataWriterQos()
75     self.publisher.get_default_datawriter_qos(self.writer_qos)
76     self.writer = self.publisher.create_datawriter(self.topic, self.writer_qos, self.
↪listener)
77
78
79     def write(self):
80         data = HelloWorld.HelloWorld()
81         data.message("Hello World")
82         data.index(self.index)
83         self.writer.write(data)
84         print("Sending {message} : {index}".format(message=data.message(), index=data.
↪index()))
85         self.index = self.index + 1
86
87
88     def wait_discovery(self) :
89         self._cvDiscovery.acquire()
90         print ("Writer is waiting discovery...")
91         self._cvDiscovery.wait_for(lambda : self._matched_reader != 0)
92         self._cvDiscovery.release()
93         print("Writer discovery finished...")
94
95
96     def run(self):
97         self.wait_discovery()
98         for x in range(10) :
99             time.sleep(1)
100             self.write()
101         self.delete()
102
103
104     def delete(self):
105         factory = fastdds.DomainParticipantFactory.get_instance()
106         self.participant.delete_contained_entities()
107         factory.delete_participant(self.participant)
108
109
110 if __name__ == '__main__':
111     print('Starting publisher.')
112     writer = Writer()

```

(continues on next page)

(continued from previous page)

```

113     writer.run()
114     exit()

```

Examining the code

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

```
import fastdds
```

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

```
import HelloWorld
```

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

```

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

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

```

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

```
class Writer:
```

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

1. Uses the `DomainParticipantFactory` to create the participant.
2. Registers the data type defined in the IDL.
3. Creates the topic for the publications.

4. Creates the publisher.
5. Creates the DataWriter with the listener previously created.

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

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

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

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

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

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

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

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

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

```
def wait_discovery(self) :
    self._cvDiscovery.acquire()
    print ("Writer is waiting discovery...")
```

(continues on next page)

(continued from previous page)

```
self._cvDiscovery.wait_for(lambda : self._matched_reader != 0)
self._cvDiscovery.release()
print("Writer discovery finished...")
```

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

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

Finally, the `Writer` is initialized and run in main.

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

Write the Fast DDS subscriber

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

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

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

```
1  # Copyright 2022 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 HelloWorld Subscriber
16 """
17 import signal
18
19 import fastdds
20 import HelloWorld
```

(continues on next page)

(continued from previous page)

```

21 DESCRIPTION = """HelloWorld Subscriber example for Fast DDS python bindings"""
22 USAGE = ('python3 HelloWorldSubscriber.py')
23
24 # To capture ctrl+C
25 def signal_handler(sig, frame):
26     print('Interrupted!')
27
28 class ReaderListener(fastdds.DataReaderListener):
29
30
31     def __init__(self):
32         super().__init__()
33
34
35     def on_subscription_matched(self, datareader, info) :
36         if (0 < info.current_count_change) :
37             print ("Subscriber matched publisher {}".format(info.last_publication_
38 ↪handle))
39         else :
40             print ("Subscriber unmatched publisher {}".format(info.last_publication_
41 ↪handle))
42
43     def on_data_available(self, reader):
44         info = fastdds.SampleInfo()
45         data = HelloWorld.HelloWorld()
46         reader.take_next_sample(data, info)
47
48         print("Received {message} : {index}".format(message=data.message(), index=data.
49 ↪index()))
50
51 class Reader:
52
53     def __init__(self):
54         factory = fastdds.DomainParticipantFactory.get_instance()
55         self.participant_qos = fastdds.DomainParticipantQos()
56         factory.get_default_participant_qos(self.participant_qos)
57         self.participant = factory.create_participant(0, self.participant_qos)
58
59         self.topic_data_type = HelloWorld.HelloWorldPubSubType()
60         self.topic_data_type.setName("HelloWorld")
61         self.type_support = fastdds.TypeSupport(self.topic_data_type)
62         self.participant.register_type(self.type_support)
63
64         self.topic_qos = fastdds.TopicQos()
65         self.participant.get_default_topic_qos(self.topic_qos)
66         self.topic = self.participant.create_topic("HelloWorldTopic", self.topic_data_
67 ↪type.getName(), self.topic_qos)
68

```

(continues on next page)

(continued from previous page)

```

69     self.subscriber_qos = fastdds.SubscriberQos()
70     self.participant.get_default_subscriber_qos(self.subscriber_qos)
71     self.subscriber = self.participant.create_subscriber(self.subscriber_qos)
72
73     self.listener = ReaderListener()
74     self.reader_qos = fastdds.DataReaderQos()
75     self.subscriber.get_default_datareader_qos(self.reader_qos)
76     self.reader = self.subscriber.create_datareader(self.topic, self.reader_qos,
↪ self.listener)
77
78
79     def delete(self):
80         factory = fastdds.DomainParticipantFactory.get_instance()
81         self.participant.delete_contained_entities()
82         factory.delete_participant(self.participant)
83
84
85     def run(self):
86         signal.signal(signal.SIGINT, signal_handler)
87         print('Press Ctrl+C to stop')
88         signal.pause()
89         self.delete()
90
91
92 if __name__ == '__main__':
93     print('Creating subscriber.')
94     reader = Reader()
95     reader.run()
96     exit()

```

Examining the code

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

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

```

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

```

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

```

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

```

(continues on next page)

(continued from previous page)

```
data = HelloWorld.HelloWorld()
reader.take_next_sample(data, info)
```

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

```
class Reader:
```

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

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

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

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

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

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

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

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

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

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

Putting all together

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

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

Summary

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

Next steps

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

See also:

Check Fast DDS dependencies, libraries, and related packages in the *Dependencies and compatibilities* section.

7.17 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.

7.17.1 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 as an abstraction layer of the transport layer.
- **Transport Layer.** *Fast DDS* can be used over various transport protocols such as unreliable transport protocols (UDP), reliable transport protocols (TCP), or shared memory transport protocols (SHM).

Fig. 4: *Fast DDS* layer model architecture

DDS Layer

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

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

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

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

RTPS layer

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

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

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

For a more detailed explanation of each entity, their attributes, and their listeners, please refer to [RTPS Layer](#) section.

Transport layer

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

7.17.2 Programming and execution model

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

Concurrency and multithreading

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

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

Name	Type	Cardinality	OS thread name	Description
Event	General	One per DomainParticipant	dds.ev.<participant_id>	Processes periodic and triggered time events. See DomainParticipantQos .
Discovery Server Event	General	One per DomainParticipant	dds.ds_ev.<participant_id>	Synchronizes access to the Discovery Server Database. See DomainParticipantQos .
Asynchronous Writer	General	One per enabled asynchronous flow controller. Minimum 1.	dds.asyn.<participant_id>.<async_flow_control	Manages asynchronous writes. Even for synchronous writers, some forms of communication must be initiated in the background. See DomainParticipantQos and FlowControllersQos .
Datasharing Listener	General	One per DataReader	dds.dsha.<reader_id>	Listener thread that processes messages received via Datasharing. See DataReaderQos .
Reception	UDP	One per port	dds.udp.<port>	Listener thread that processes incoming UDP messages. See TransportConfigQos and UDPTransportDescriptor .
Reception	TCP	One per TCP connection	dds.tcp.<port>	Listener thread that processes incoming TCP messages. See TCPTransportDescriptor .
Accept	TCP	One per TCP transport	dds.tcp_accept	Thread that processes incoming TCP connection requests. See TCPTransportDescriptor .
Keep Alive	TCP	One per TCP transport	dds.tcp_keep	Keep alive thread for TCP connections. See TCPTransportDescriptor .
Reception	SHM	One per port	dds.shm.<port>	Listener thread that processes incoming messages via SHM segments. See TransportConfigQos and SharedMemTransportDescriptor .
Logging	SHM	One per port	dds.shmd.<port>	Stores and dumps transferred packets to a file. See TransportConfigQos and SharedMemTransportDescriptor .
Watchdog	SHM	One	dds.shm.wdog	Monitors health of open shared memory segments. See TransportConfigQos and SharedMemTransportDescriptor .
General Logging	Log	One	dds.log	Accumulates and writes to the appropriate consumer log entries. See Logging Thread .
Security Logging	Log	One per DomainParticipant	dds.slog.<participant_id>	Accumulates and writes security log entries. See DomainParticipantQos .
Watchdog	File-watch	One	dds.fwatch	Tracks the status of the watched file for modifications. See DomainParticipantFactoryQos .
Callback	File-watch	One	dds.fwatch.cb	Runs the registered callback when the watched file changes. See DomainParticipantFactoryQos .
Reception	Type-Lookup Service	Two per DomainParticipant	dds.tls.replies.<participant_id> dds.tls.requests.<participant_id>	Runs when remote endpoint discovery information has been received with unknown data type.

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

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

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

Fast DDS offers the possibility of configuring certain attributes of the threads it creates by means of the [ThreadSettings](#).

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.

7.17.3 Functionalities

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

Discovery Protocols

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

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

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

Security

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

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

More information about security configuration in *Fast DDS* is available in the [Security](#) section.

Logging

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

XML profiles configuration

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

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

For a better understanding of how to write and use these XML profiles configuration files you can continue reading the [XML profiles](#) section.

Environment variables

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

7.18 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.

7.18.1 Core

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

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

Entity

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

Types of Entities

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

The following figure shows the hierarchy between all DDS entities:

Common Entity Characteristics

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

Entity Identifier

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

QoS policy

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

Listener

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

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

Fig. 5: Listeners inheritance diagram.

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

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

- `on_sample_rejected()`
- `on_liveliness_changed()`

- `on_requested_deadline_missed()`
- `on_requested_incompatible_qos()`
- `on_data_available()`
- `on_subscription_matched()`
- `on_sample_lost()`
- `on_offered_incompatible_qos()`
- `on_offered_deadline_missed()`
- `on_liveliness_lost()`
- `on_publication_matched()`
- `on_inconsistent_topic()`
- `on_data_on_readers()`

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

- `on_participant_discovery()`
- `onParticipantAuthentication()`
- `on_data_reader_discovery()`
- `on_data_writer_discovery()`
- `on_unacknowledged_sample_removed()`

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

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

Status

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

StatusCondition

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

Enabling Entities

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

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

Any other function called in this state will return NOT_ENABLED.

Policy

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

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

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

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

Standard QoS Policies

This section explains each of the DDS standard QoS Policies:

- *DeadlineQosPolicy*
- *DestinationOrderQosPolicy*
- *DurabilityQosPolicy*
- *DurabilityServiceQosPolicy*
- *EntityFactoryQosPolicy*
- *GroupDataQosPolicy*
- *HistoryQosPolicy*
- *LatencyBudgetQosPolicy*

- *LifespanQosPolicy*
- *LivelinessQosPolicy*
- *OwnershipQosPolicy*
- *OwnershipStrengthQosPolicy*
- *PartitionQosPolicy*
- *PresentationQosPolicy*
- *ReaderDataLifecycleQosPolicy*
- *ReliabilityQosPolicy*
- *ResourceLimitsQosPolicy*
- *TimeBasedFilterQosPolicy*
- *TopicDataQosPolicy*
- *TransportPriorityQosPolicy*
- *UserDataQosPolicy*
- *WriterDataLifecycleQosPolicy*

DeadlineQosPolicy

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

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

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
period	Duration_t	c_TimeInfinite

Note: This QoS Policy concerns to [Topic](#), [DataReader](#) and [DataWriter](#) entities.

It can be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See [Compatibility Rule](#) for further details.

Compatibility Rule

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

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

Example

C++

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

XML

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

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

DestinationOrderQosPolicy

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

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

List of QoS Policy data members:

Data Name	Member	Type	Default Value
kind		<i>DestinationOrderQosPolicyKind</i>	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
<code>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</code>	<code>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</code>	Yes
<code>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</code>	<code>BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</code>	No
<code>BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</code>	<code>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</code>	Yes
<code>BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</code>	<code>BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS</code>	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	<i>DurabilityQosPolicyKind</i>	VOLATILE_DURABILITY_QOS for DataReaders TRANSIENT_LOCAL_DURABILITY_QOS for DataWriters

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

It cannot be changed on enabled entities.

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

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

DurabilityQosPolicyKind

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

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

Compatibility Rule

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

VOLATILE_DURABILITY_QOS < TRANSIENT_LOCAL_DURABILITY_QOS < TRANSIENT_DURABILITY_QOS < PERSISTENT_DURABILITY_QOS

Table with the possible combinations:

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

Example

C++

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

XML

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


DurabilityServiceQosPolicy

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

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

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
service_cleanup_delay	Duration_t	c_TimeZero
history_kind	<i>HistoryQosPolicyKind</i>	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
<code>autoenable_created_entities</code>	<code>bool</code>	<code>true</code>

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
↪ 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
<code>collection</code>	<code>std::vector<octet></code>	Empty vector

Note: This QoS Policy concerns to *Publisher* and *Subscriber* entities.

It can be changed on enabled entities.

Example

C++

```
GroupDataQosPolicy group_data;
//The GroupDataQosPolicy is default constructed with an empty collection
//Collection is a private member so you need to use getters and setters to access
//Add data to the collection
std::vector<eprosima::fastrtps::rtps::octet> vec;
vec = group_data.data_vec(); // Getter function

//Add two new octets to group data vector
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
val = 10;
vec.push_back(val);
group_data.data_vec(vec); //Setter function
```

XML

```
<data_writer profile_name="writer_xml_conf_groupdata_profile">
  <qos>
    <groupData>
      <value>3.a</value>
    </groupData>
  </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_groupdata_profile">
  <qos>
    <groupData>
      <value>3.a</value>
    </groupData>
  </qos>
</data_reader>
```

HistoryQosPolicy

This QoS Policy controls the behavior of the system when the value of an instance changes one or more times before it can be successfully communicated to the existing DataReader entities.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	<i>HistoryQosPolicyKind</i>	KEEP_LAST_HISTORY_QOS
depth	int32_t	1

- **kind**: Controls if the service should deliver only the most recent values, all the intermediate values or do something in between. See *HistoryQosPolicyKind* for further details.
- **depth**: Establishes the maximum number of samples that must be kept on the history. It only has effect if the kind is set to KEEP_LAST_HISTORY_QOS and it needs to be consistent with the *ResourceLimitsQosPolicy*, which means that its value must be lower or equal to max_samples_per_instance.

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

It cannot be changed on enabled entities.

HistoryQosPolicyKind

There are two possible values (see `HistoryQosPolicyKind`):

- `KEEP_LAST_HISTORY_QOS`: The service will only attempt to keep the most recent values of the instance and discard the older ones. The maximum number of samples to keep and deliver is defined by the *depth* of the `HistoryQosPolicy`, which needs to be consistent with the *ResourceLimitsQosPolicy* settings. If the limit defined by *depth* is reached, the system will discard the oldest sample to make room for a new one.
- `KEEP_ALL_HISTORY_QOS`: The service will attempt to keep all the values of the instance until it can be delivered to all the existing Subscribers. If this option is selected, the depth will not have any effect, so the history is only limited by the values set in *ResourceLimitsQosPolicy*.

Consistency rule

The `HistoryQos` must be set consistently with the *ResourceLimitsQosPolicy*, but also other QoS as *DurabilityQosPolicy* and *ReliabilityQosPolicy*, so there are several cases to take into account:

- The *depth* is only considered if the *kind* is set to `KEEP_LAST_HISTORY_QOS`.
- The *depth* must be consistent with the *ResourceLimitsQosPolicy* settings, which means that the *depth* must be lower or equal than the *ResourceLimitsQosPolicy*'s *max_samples_per_instance*. Also, *max_samples* must be equal or higher than the product of *max_samples_per_instance* times *max_instances*.
- The *depth* cannot be lower or equal than zero. If an unlimited depth is required, please consider using *kind* as `KEEP_ALL_HISTORY_QOS`.
- Setting the *kind* as `KEEP_ALL_HISTORY_QOS` entails that limits are set by the *ResourceLimitsQosPolicy* limits (*max_samples_per_instance* prior than *max_samples*).
- In the case of the *ReliabilityQosPolicy* `ReliabilityQosPolicyKind` being set to `RELIABLE_RELIABILITY_QOS` and the *HistoryQosPolicy* *kind* being set to `KEEP_ALL_HISTORY_QOS`, when the resource limits are reached, the behavior of the service is depends on the *DurabilityQosPolicy*:
 - If the *DurabilityQosPolicy* *kind* is configured as `VOLATILE_DURABILITY_QOS`, the `DataWriter write()` call will discard the oldest sample in the history. Note that the removed sample may belong to different *instances* than the newly written one.
 - If the *DurabilityQosPolicy* *kind* is configured as `TRANSIENT_LOCAL_DURABILITY_QOS` or `TRANSIENT_DURABILITY_QOS`, the `DataWriter write()` call will be blocked until the history has space for the new sample.

Example

C++

```
HistoryQosPolicy history;
//The HistoryQosPolicy is default constructed with kind = KEEP_LAST and depth = 1.
//Change the depth to 20
history.depth = 20;
//You can also change the kind to KEEP_ALL but after that the depth will not have effect.
history.kind = KEEP_ALL_HISTORY_QOS;
```

XML

```
<topic>
  <historyQos>
    <kind>KEEP_LAST</kind> <!-- string -->
    <depth>20</depth> <!-- uint32 -->
  </historyQos>
</topic>
```

LatencyBudgetQosPolicy

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

This QoS Policy specifies the maximum acceptable delay from the time the data is written until the data is inserted on the DataReader History and notified of the fact. That delay by default is set to 0 in order to optimize the internal operations (see LatencyBudgetQosPolicy).

List of QoS Policy data members:

Data Member Name	Type	Default Value
duration	Duration_t	c_TimeZero

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

It can be changed on enabled entities.

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

Compatibility Rule

To maintain the compatibility between LatencyBudgetQosPolicy in DataReaders and DataWriters, the DataWriter duration must be lower or equal to the DataReader duration.

LifespanQosPolicy

Each data sample written by a *DataWriter* has an associated expiration time beyond which the data is removed from the DataWriter and DataReader history as well as from the transient and persistent information caches (see LifespanQosPolicy).

By default, the *duration* is infinite, which means that there is not a maximum duration for the validity of the samples written by the DataWriter.

The expiration time is computed by adding the *duration* to the source timestamp, which can be calculated automatically if `write()` member function is called or supplied by the application by means of `write_w_timestamp()` member function. The DataReader is allowed to use the reception timestamp instead of the source timestamp.

List of QoS Policy data members:

Data Member Name	Type	Default Value
duration	Duration_t	c_TimeInfinite

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

It can be changed on enabled entities.

Example

C++

```
LifespanQosPolicy lifespan;  
//The LifespanQosPolicy is default constructed with duration set to infinite.  
//Change the duration to 5 s  
lifespan.duration = {5, 0};
```

XML

```
<data_writer profile_name="writer_xml_conf_lifespan_profile">  
  <qos>  
    <lifespan>  
      <duration>  
        <sec>5</sec>  
      </duration>  
    </lifespan>  
  </qos>  
</data_writer>  
  
<data_reader profile_name="reader_xml_conf_lifespan_profile">  
  <qos>
```

(continues on next page)

(continued from previous page)

```

    <lifespan>
      <duration>
        <sec>5</sec>
      </duration>
    </lifespan>
  </qos>
</data_reader>

```

LivelinessQosPolicy

This QoS Policy controls the mechanism used by the service to ensure that a particular entity on the network is still alive. There are different settings that allow distinguishing between applications where data is updated periodically and applications where data is changed sporadically. It also allows customizing the application regarding the kind of failures that should be detected by the liveliness mechanism (see [LivelinessQosPolicy](#)).

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	LivelinessQosPolicyKind	AUTOMATIC_LIVELINESS_QOS
lease_duration	Duration_t	c_TimeInfinite
announcement_period	Duration_t	c_TimeInfinite

- **kind:** This data member establishes if the service needs to assert the liveliness automatically or if it needs to wait until the liveliness is asserted by the publishing side. See [LivelinessQosPolicyKind](#) for further details.
- **lease_duration:** Amount of time to wait since the last time the DataWriter asserts its liveliness to consider that it is no longer alive.
- **announcement_period:** Amount of time between consecutive liveliness messages sent by the DataWriter. This data member only takes effect if the kind is AUTOMATIC_LIVELINESS_QOS or MANUAL_BY_PARTICIPANT_LIVELINESS_QOS and needs to be lower than the lease_duration.

Note: This QoS Policy concerns to [Topic](#), [DataReader](#) and [DataWriter](#) entities.

It cannot be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See [Compatibility Rule](#) for further details.

LivelinessQosPolicyKind

There are three possible values (see [LivelinessQosPolicyKind](#)):

- **AUTOMATIC_LIVELINESS_QOS:** The service takes the responsibility for renewing the leases at the required rates, as long as the local process where the participant is running and the link connecting it to remote participants exists, the entities within the remote participant will be considered alive. This kind is suitable for applications that only need to detect whether a remote application is still running.

- The two *Manual* modes require that the application on the publishing side asserts the liveliness periodically before the `lease_duration` timer expires. Publishing any new data value implicitly asserts the DataWriter's liveliness, but it can be done explicitly by calling the `assert_liveliness` member function.
 - `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS`: If one of the entities in the publishing side asserts its liveliness, the service deduces that all other entities within the same `DomainParticipant` are also alive.
 - `MANUAL_BY_TOPIC_LIVELINESS_QOS`: This mode is more restrictive and requires that at least one instance within the DataWriter is asserted to consider that the DataWriter is alive.

Compatibility Rule

To maintain the compatibility between `LivelinessQosPolicy` in DataReaders and DataWriters, the DataWriter kind must be higher or equal to the DataReader kind. And the order between the different kinds is:

|`AUTOMATIC_LIVELINESS_QOS-api`| < |`MANUAL_BY_PARTICIPANT_LIVELINESS_QOS-api`| < |`MANUAL_BY_TOPIC_LIVELINESS_QOS-api`|

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
<code>AUTOMATIC_LIVELINESS_QOS</code>	<code>AUTOMATIC_LIVELINESS_QOS</code>	Yes
<code>AUTOMATIC_LIVELINESS_QOS</code>	<code>MANUAL_BY_PARTICIPANT_LIVELINESS_QOS</code>	No
<code>AUTOMATIC_LIVELINESS_QOS</code>	<code>MANUAL_BY_TOPIC_LIVELINESS_QOS</code>	No
<code>MANUAL_BY_PARTICIPANT_LIVELINESS_QOS</code>	<code>AUTOMATIC_LIVELINESS_QOS</code>	Yes
<code>MANUAL_BY_PARTICIPANT_LIVELINESS_QOS</code>	<code>MANUAL_BY_PARTICIPANT_LIVELINESS_QOS</code>	Yes
<code>MANUAL_BY_PARTICIPANT_LIVELINESS_QOS</code>	<code>MANUAL_BY_TOPIC_LIVELINESS_QOS</code>	No
<code>MANUAL_BY_TOPIC_LIVELINESS_QOS</code>	<code>AUTOMATIC_LIVELINESS_QOS</code>	Yes
<code>MANUAL_BY_TOPIC_LIVELINESS_QOS</code>	<code>MANUAL_BY_PARTICIPANT_LIVELINESS_QOS</code>	Yes
<code>MANUAL_BY_TOPIC_LIVELINESS_QOS</code>	<code>MANUAL_BY_TOPIC_LIVELINESS_QOS</code>	Yes

Additionally, the `lease_duration` of the DataWriter must not be greater than the `lease_duration` of the DataReader.

Example

C++

```
LivelinessQosPolicy liveliness;
//The LivelinessQosPolicy is default constructed with kind = AUTOMATIC
//Change the kind to MANUAL_BY_PARTICIPANT
liveliness.kind = MANUAL_BY_PARTICIPANT_LIVELINESS_QOS;
//The LivelinessQosPolicy is default constructed with lease_duration set to infinite
//Change the lease_duration to 1 second
liveliness.lease_duration = {1, 0};
//The LivelinessQosPolicy is default constructed with announcement_period set to infinite
//Change the announcement_period to 1 ms
liveliness.announcement_period = {0, 1000000};
```

XML


```

<data_writer profile_name="writer_xml_conf_liveliness_profile">
  <qos>
    <liveliness>
      <announcement_period>
        <nanosec>1000000</nanosec>
      </announcement_period>
      <lease_duration>
        <sec>1</sec>
      </lease_duration>
      <kind>AUTOMATIC</kind>
    </liveliness>
  </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_liveliness_profile">
  <qos>
    <liveliness>
      <lease_duration>
        <sec>1</sec>
      </lease_duration>
      <kind>AUTOMATIC</kind>
    </liveliness>
  </qos>
</data_reader>

```

OwnershipQosPolicy

This QoS Policy specifies whether it is allowed for multiple DataWriters to update the same instance of data, and if so, how these modifications should be arbitrated (see [OwnershipQosPolicy](#)).

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	<i>OwnershipQosPolicyKind</i>	SHARED_OWNERSHIP_QOS

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

It cannot be changed on enabled entities.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See [Compatibility Rule](#) for further details.

OwnershipQosPolicyKind

There are two possible values (see `OwnershipQosPolicyKind`):

- `SHARED_OWNERSHIP_QOS`: This option indicates that the service does not enforce unique ownership for each instance. In this case, multiple DataWriters are allowed to update the same data instance and all the updates are made available to the existing DataReaders. Those updates are also subject to the *TimeBasedFilterQosPolicy* or *HistoryQosPolicy* settings, so they can be filtered.
- `EXCLUSIVE_OWNERSHIP_QOS`: This option indicates that each instance can only be updated by one DataWriter, meaning that at any point in time a single DataWriter owns each instance and is the only one whose modifications will be visible for the existing DataReaders. The owner can be changed dynamically according to the highest *strength* between the alive DataWriters, which has not violated the deadline contract concerning the data instances. That *strength* can be changed using the *OwnershipStrengthQosPolicy*. In case two DataWriters have the same *strength* value, the DataWriter with a lower *GUID* value would be the owner of the topic.

Compatibility Rule

To maintain the compatibility between `OwnershipQosPolicy` in *DataReaders* and *DataWriters*, the DataWriter kind must be equal to the DataReader kind.

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
<code>SHARED_OWNERSHIP_QOS</code>	<code>SHARED_OWNERSHIP_QOS</code>	Yes
<code>SHARED_OWNERSHIP_QOS</code>	<code>EXCLUSIVE_OWNERSHIP_QOS</code>	No
<code>EXCLUSIVE_OWNERSHIP_QOS</code>	<code>SHARED_OWNERSHIP_QOS</code>	No
<code>EXCLUSIVE_OWNERSHIP_QOS</code>	<code>EXCLUSIVE_OWNERSHIP_QOS</code>	Yes

Example

C++

```
OwnershipQosPolicy ownership;
//The OwnershipQosPolicy is default constructed with kind = SHARED.
//Change the kind to EXCLUSIVE
ownership.kind = EXCLUSIVE_OWNERSHIP_QOS;
```

XML

```
<data_writer profile_name="writer_xml_conf_ownership_profile">
  <qos>
    <ownership>
      <kind>EXCLUSIVE</kind>
    </ownership>
  </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_ownership_profile">
  <qos>
    <ownership>
```

(continues on next page)

(continued from previous page)

```

        <kind>EXCLUSIVE</kind>
    </ownership>
</qos>
</data_reader>

```

OwnershipStrengthQosPolicy

This QoS Policy specifies the value of the *strength* used to arbitrate among multiple DataWriters that attempt to modify the same data instance. It is only applicable if the *OwnershipQosPolicy* kind is set to EXCLUSIVE_OWNERSHIP_QOS. See OwnershipStrengthQosPolicy.

List of QoS Policy data members:

Data Member Name	Type	Default Value
value	uint32_t	0

Note: This QoS Policy concerns to DataWriter entities.

It can be changed on enabled entities.

Example

C++

```

OwnershipStrengthQosPolicy ownership_strength;
//The OwnershipStrengthQosPolicy is default constructed with value 0
//Change the strength to 10
ownership_strength.value = 10;

```

XML

```

<data_writer profile_name="writer_xml_conf_ownership_strength_profile">
  <qos>
    <ownershipStrength>
      <value>10</value>
    </ownershipStrength>
  </qos>
</data_writer>

```

PartitionQosPolicy

This Qos Policy allows the introduction of a logical partition inside the physical partition introduced by a domain. For a DataReader to see the changes made by a DataWriter, not only the Topic must match, but also they have to share at least one logical partition (see PartitionQosPolicy).

The empty string is also considered as a valid partition and it matches with other partition names using the same rules of string matching and regular-expression matching used for any other partition name.

List of QoS Policy data members:

Data Member Name	Type	Default Value
max_size	uint32_t	0 (Length Unlimited)
names	SerializedPayload_t	Empty List

- max_size: Maximum size for the list of partition names.
- names: List of partition names.

Note: This QoS Policy concerns to Publisher and Subscriber entities.

Partitions can also be explicitly defined at the endpoint level to override this configuration. Information to do so can be found [here](#).

It can be changed on enabled Publishers and Subscribers.

Example

C++

```
PartitionQosPolicy partitions;
//The PartitionsQosPolicy is default constructed with max_size = 0.
//Max_size is a private member so you need to use getters and setters to access
//Change the max_size to 20
partitions.set_max_size(20); //Setter function
//The PartitionsQosPolicy is default constructed with an empty list of partitions
//Partitions is a private member so you need to use getters and setters to access
//Add new partitions
std::vector<std::string> part = partitions.names(); //Getter function
part.push_back("part1");
part.push_back("part2");
partitions.names(part); //Setter function
```

XML

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

(continues on next page)

(continued from previous page)

```

        </partition>
    </qos>
</data_writer>

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

```

PresentationQosPolicy

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

This QoS Policy specifies how the samples representing changes to data instances are presented to the subscribing application. It controls the extent to which changes to data instances can be made dependent on each other, as well as the kind of dependencies that can be propagated and maintained. See `PresentationQosPolicy`.

List of QoS Policy data members:

Data Member Name	Type	Default Value
<code>access_scope</code>	<i>PresentationQosPolicyAccessScopeKind</i>	INSTANCE_PRESENTATION_QOS
<code>coherent_access</code>	bool	false
<code>ordered_access</code>	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.

ReaderDataLifecyleQosPolicy

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 *ReaderDataLifecyleQosPolicy*.

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
<code>autopurge_no_writer_samples_delay</code>	<code>Duration_t</code>	<code>c_TimeInfinite</code>
<code>autopurge_disposed_samples_delay</code>	<code>Duration_t</code>	<code>c_TimeInfinite</code>

- `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_t	Duration_t	100 ms

- **kind:** Specifies the behavior of the service regarding delivery of the samples. See [ReliabilityQosPolicyKind](#) for further details.
- **max_blocking_time:** Configures the maximum duration that the write operation can be blocked.

Note: This QoS Policy concerns to [Topic](#), [DataWriter](#) and [DataReader](#) entities.

It cannot be changed on enabled entities.

Important: Setting this QoS Policy to BEST_EFFORT_RELIABILITY_QOS affects to the [DurabilityQosPolicy](#), making the endpoints behave as VOLATILE_DURABILITY_QOS.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See [Compatibility Rule](#) for further details.

ReliabilityQosPolicyKind

There are two possible values ():

- **BEST_EFFORT_RELIABILITY_QOS:** It indicates that it is acceptable not to retransmit the missing samples, so the messages are sent without waiting for an arrival confirmation. Presumably new values for the samples are generated often enough that it is not necessary to re-send any sample. However, the data samples sent by the same DataWriter will be stored in the DataReader history in the same order they occur. In other words, even if the DataReader misses some data samples, an older value will never overwrite a newer value.
- **RELIABLE_RELIABILITY_QOS:** It indicates that the service will attempt to deliver all samples of the DataWriter's history expecting an arrival confirmation from the DataReader. The data samples sent by the same DataWriter cannot be made available to the DataReader if there are previous samples that have not been received yet. The service will retransmit the lost data samples in order to reconstruct a correct snapshot of the DataWriter history before it is accessible by the DataReader.

This option may block the write operation, hence the `max_blocking_time` is set that will unblock it once the time expires. But if the `max_blocking_time` expires before the data is sent, the write operation will return an error.

Compatibility Rule

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

|BEST_EFFORT_RELIABILITY_QOS-api| < |RELIABLE_RELIABILITY_QOS-api|

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
BEST_EFFORT_RELIABILITY_QOS	BEST_EFFORT_RELIABILITY_QOS	Yes
BEST_EFFORT_RELIABILITY_QOS	RELIABLE_RELIABILITY_QOS	No
RELIABLE_RELIABILITY_QOS	BEST_EFFORT_RELIABILITY_QOS	Yes
RELIABLE_RELIABILITY_QOS	RELIABLE_RELIABILITY_QOS	Yes

Example

C++

```
ReliabilityQosPolicy reliability;
//The ReliabilityQosPolicy is default constructed with kind = BEST_EFFORT
//Change the kind to RELIABLE
reliability.kind = RELIABLE_RELIABILITY_QOS;
//The ReliabilityQosPolicy is default constructed with max_blocking_time = 100ms
//Change the max_blocking_time to 1s
reliability.max_blocking_time = {1, 0};
```

XML

```
<data_writer profile_name="writer_xml_conf_reliability_profile">
  <qos>
    <reliability>
      <kind>RELIABLE</kind>
      <max_blocking_time>
        <sec>1</sec>
      </max_blocking_time>
    </reliability>
  </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_reliability_profile">
  <qos>
    <reliability>
      <kind>BEST_EFFORT</kind>
    </reliability>
  </qos>
</data_reader>
```

ResourceLimitsQosPolicy

This QoS Policy controls the resources that the service can use in order to meet the requirements imposed by the application and other QoS Policies. See [ResourceLimitsQosPolicy](#).

List of QoS Policy data members:

Data Member Name	Type	Default Value
max_samples	int32_t	5000
max_instances	int32_t	10
max_samples_per_instance	int32_t	400
allocated_samples	int32_t	100
extra_samples	int32_t	1

- **max_samples**: Controls the maximum number of samples that the DataWriter or DataReader can manage across all the instances associated with it. In other words, it represents the maximum samples that the middleware can store for a DataReader or DataWriter. **Value less or equal to 0 means infinite resources.**
- **max_instances**: Controls the maximum number of instances that a DataWriter or DataReader can manage. **Value less or equal to 0 means infinite resources.**
- **max_samples_per_instance**: Controls the maximum number of samples within an instance that the DataWriter or DataReader can manage. **Value less or equal to 0 means infinite resources.**
- **allocated_samples**: States the number of samples that will be allocated on initialization.
- **extra_samples**: States the number of extra samples that will be allocated on the pool, so the maximum number of samples on the pool will be **max_samples** plus **extra_samples**. These extra samples act as a reservoir of samples even when the history is full.

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

It cannot be changed on enabled entities.

Consistency Rule

To maintain the consistency within the ResourceLimitsQosPolicy, the values of the data members must follow the next conditions:

- The value of **max_samples** must be higher or equal to the value of **max_samples_per_instance**.
- The value established for the [HistoryQosPolicy](#) depth must be lower or equal to the value stated for **max_samples_per_instance**.

Example

C++

```
ResourceLimitsQosPolicy resource_limits;
//The ResourceLimitsQosPolicy is default constructed with max_samples = 5000
//Change max_samples to 200
resource_limits.max_samples = 200;
//The ResourceLimitsQosPolicy is default constructed with max_instances = 10
//Change max_instances to 20
resource_limits.max_instances = 20;
//The ResourceLimitsQosPolicy is default constructed with max_samples_per_instance = 400
//Change max_samples_per_instance to 100 as it must be lower than max_samples
resource_limits.max_samples_per_instance = 100;
//The ResourceLimitsQosPolicy is default constructed with allocated_samples = 100
//Change allocated_samples to 50
resource_limits.allocated_samples = 50;
```

XML

```
<data_writer profile_name="writer_xml_conf_resource_limits_profile">
  <topic>
    <resourceLimitsQos>
      <max_samples>200</max_samples>
      <max_instances>20</max_instances>
      <max_samples_per_instance>100</max_samples_per_instance>
      <allocated_samples>50</allocated_samples>
    </resourceLimitsQos>
  </topic>
</data_writer>

<data_reader profile_name="reader_xml_conf_resource_limits_profile">
  <topic>
    <resourceLimitsQos>
      <max_samples>200</max_samples>
      <max_instances>20</max_instances>
      <max_samples_per_instance>100</max_samples_per_instance>
      <allocated_samples>50</allocated_samples>
    </resourceLimitsQos>
  </topic>
</data_reader>
```

TimeBasedFilterQosPolicy

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

Filter that allows a *DataReader* to specify that it is interested only in a subset of the values of the data. This filter states that the DataReader does not want to receive more than one value each `minimum_separation`, regardless of how fast the changes occur. See *TimeBasedFilterQosPolicy*.

The `minimum_separation` must be lower than the *DeadlineQosPolicy* period. By default, the

`minimum_separation` is zero, which means that the `DataReader` is potentially interested in all the values.

List of QoS Policy data members:

Data Member Name	Type	Default Value
<code>minimum_separation</code>	<code>Duration_t</code>	<code>c_TimeZero</code>

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
<code>collection</code>	<code>std::vector<octet></code>	Empty vector

Note: This QoS Policy concerns to `Topic` entities.

It can be changed even if it is already created.

Example

C++

```
//The TopicDataQosPolicy is default constructed with an empty vector.
TopicDataQosPolicy topic_data;
std::vector<eprosima::fastrtps::rtps::octet> vec;
vec = topic_data.data_vec(); // Getter Function

//Add two new octets to topic data vector
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
val = 10;
vec.push_back(val);
topic_data.data_vec(vec); //Setter Function
```

XML

```
<data_writer profile_name="writer_xml_conf_topicdata_profile">
  <qos>
    <topicData>
      <value>3.a</value>
```

(continues on next page)

(continued from previous page)

```

        </topicData>
    </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_topicdata_profile">
    <qos>
        <topicData>
            <value>3.a</value>
        </topicData>
    </qos>
</data_reader>

```

TransportPriorityQosPolicy

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

The purpose of this QoS Policy is to allow the service to take advantage of those transports capable of sending messages with different priorities. It establishes the priority of the underlying transport used to send the data. See `TransportPriorityQosPolicy`

You can choose any value within the 32-bit range for the priority. The higher the value, the higher the priority.

List of QoS Policy data members:

Data Member Name	Type	Default Value
value	uint32_t	0

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

It can be changed on enabled entities.

UserDataQosPolicy

Allows the application to attach additional information to the *Entity* object so that when the entity is discovered the remote application can access the data and use it. For example, it can be used to attach the security credentials to authenticate the source from the remote application. See `UserDataQosPolicy`.

List of QoS Policy data members:

Data Member Name	Type	Default Value
collection	std::vector<octet>	Empty vector

Note: This QoS Policy concerns to all DDS entities.

It can be changed on enabled entities.

Example

C++

```
//The TopicDataQosPolicy is default constructed with an empty vector.
UserDataQosPolicy user_data;
std::vector<eprosima::fastrtps::rtps::octet> vec;
vec = user_data.data_vec(); // Getter Function

//Add two new octets to user data vector
eprosima::fastrtps::rtps::octet val = 3;
vec.push_back(val);
val = 10;
vec.push_back(val);
user_data.data_vec(vec); //Setter Function
```

XML

```
<participant profile_name="participant_xml_conf_userdata_profile">
  <rtps>
    <userData>
      <value>3.a</value>
    </userData>
  </rtps>
</participant>

<data_writer profile_name="writer_xml_conf_userdata_profile">
  <qos>
    <userData>
      <value>3.a</value>
    </userData>
  </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_userdata_profile">
  <qos>
    <userData>
      <value>3.a</value>
    </userData>
  </qos>
</data_reader>
```

WriterDataLifecycleQosPolicy

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

This QoS Policy specifies the behavior of the DataWriter with respect to the lifecycle of the data instances it manages, that is, the instance that has been either explicitly registered with the DataWriter using the register operations or implicitly by directly writing data.

The `autodispose_unregistered_instances` controls whether a DataWriter will automatically dispose an instance each time it is unregistered. Even if it is disabled, the application can still get the same result if it uses the dispose operation before unregistering the instance.

List of QoS Policy data members:

Data Member Name	Type	Default Value
<code>autodispose_unregistered_instances</code>	<code>bool</code>	<code>true</code>

Note: This QoS Policy concerns to DataWriter entities.

It can be changed on enabled entities.

eProsima Extensions

The eProsima QoS Policies extensions are those that allow changing the values of the RTPS layer configurable settings.

- *DataSharingQosPolicy*
- *DisablePositiveACKsQosPolicy*
- *FlowControllersQos*
- *ParticipantResourceLimitsQos*
- *PropertyPolicyQos*
- *PublishModeQosPolicy*
- *ReaderResourceLimitsQos*
- *RTPSEndpointQos*
- *RTPSReliableReaderQos*
- *RTPSReliableWriterQos*
- *ThreadSettings*
- *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	<code>kind()</code>	AUTO
Shared memory directory	<code>string</code>	<code>shm_directory()</code>	Empty string
Maximum domain number	<code>uint32_t</code>	<code>max_domains()</code>	0 (unlimited)
Data-sharing domain IDs	<code>vector<uint64_t></code>	<code>domain_ids()</code>	Empty
Data-sharing listener thread settings	ThreadSettings	<code>data_sharing_listener_thread()</code>	

- 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 than 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.
- Data-sharing listener thread settings: The [ThreadSettings](#) for the data-sharing thread dedicated to listening for incoming traffic.

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
<code>automatic()</code>	AUTO	Optional	Optional
<code>on()</code>	ON	Mandatory	Optional
<code>off()</code>	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();

// [OPTIONAL] ThreadSettings for listening thread
datasharing.data_sharing_listener_thread(eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com">
    <data_writer profile_name="writer_profile_qos_datasharing">
      <qos>
        <data_sharing>
          <kind>AUTOMATIC</kind>
          <domain_ids>
            <domainId>123</domainId>
            <domainId>098</domainId>
```

(continues on next page)

(continued from previous page)

```

        </domain_ids>
      </data_sharing>
    </qos>
  </data_writer>

  <data_reader profile_name="reader_profile_qos_datasharing">
    <qos>
      <data_sharing>
        <kind>AUTOMATIC</kind>
        <domain_ids>
          <domainId>123</domainId>
          <domainId>098</domainId>
        </domain_ids>
        <data_sharing_listener_thread>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </data_sharing_listener_thread>
      </data_sharing>
    </qos>
  </data_reader>
</profiles>
</dds>

```

DisablePositiveACKsQosPolicy

This additional QoS allows reducing network traffic when strict reliable communication is not required and bandwidth is limited. It consists in changing the default behavior by which positive acks are sent from readers to writers. Instead, only negative acks will be sent when a reader is missing a sample, but writers will keep data for an adjustable time before considering it as acknowledged. See `DisablePositiveACKsQosPolicy`.

List of QoS Policy data members:

Data Member Name	Type	Default Value
enabled	bool	false
duration	Duration_t	c_TimeInfinite

- **enabled:** Specifies if the QoS is enabled or not. If it is true means that the positive acks are disabled and the `DataReader` only sends negative acks. Otherwise, both positive and negative acks are sent.
- **duration:** State the duration that the `DataWriters` keep the data before considering it as acknowledged. This value does not apply to `DataReaders`.

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

The enabled Data Member cannot be modified on enabled entities. Thus, this feature must be set up during initialization. Only the duration Data Member can be modified at runtime.

Warning: For DataWriters and DataReaders to match, they must follow the compatibility rule. See [Compatibility Rule](#) for further details.

Compatibility Rule

To maintain the compatibility between `DisablePositiveACKsQoSPolicy` in DataReaders and DataWriters, the DataReader cannot have this QoS enabled if the DataWriter have it disabled.

Table with the possible combinations:

DataWriter <i>enabled</i> value	DataReader <i>enabled</i> 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

```
<data_writer profile_name="writer_xml_conf_disable_positive_acks_profile">
  <qos>
    <disablePositiveAcks>
      <enabled>true</enabled>
      <duration>
        <sec>1</sec>
      </duration>
    </disablePositiveAcks>
  </qos>
</data_writer>

<data_reader profile_name="reader_xml_conf_disable_positive_acks_profile">
  <qos>
    <disablePositiveAcks>
      <enabled>true</enabled>
    </disablePositiveAcks>
  </qos>
</data_reader>
```

(continues on next page)

(continued from previous page)

```
</qos>
</data_reader>
```

FlowControllersQos

This QoS configures the list of flow controllers of a participant, so they can later be used on its DataWriters. It is a vector of shared pointers to `FlowControllerDescriptor`, which has the following fields:

Data Member Name	Type	Default Value
name	const char *	
scheduler	FlowControllerSchedulerPolicy	FIFO_SCHED_POLICY-api
max_bytes_per_period	int32_t	0 (i.e. infinite)
period_ms	uint64_t	100
sender_thread	<i>ThreadSettings</i>	

Please refer to *Flow Controllers* section for more information.

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

It cannot be changed on enabled entities.

ParticipantResourceLimitsQos

This QoS configures allocation limits and the use of physical memory for internal resources. See `ParticipantResourceLimitsQos`.

List of QoS Policy data members:

Data Member Name	Type
locators	<i>RemoteLocatorsAllocationAttributes</i>
participants	<i>ResourceLimitedContainerConfig</i>
readers	<i>ResourceLimitedContainerConfig</i>
writers	<i>ResourceLimitedContainerConfig</i>
send_buffers	<i>SendBuffersAllocationAttributes</i>
data_limits	<i>VariableLengthDataLimits</i>
content_filter	<i>ContentFilterProperty::AllocationConfiguration</i>

- **locators:** Defines the limits for collections of remote locators.
- **participants:** Specifies the allocation behavior and limits for collections dependent on the total number of participants.
- **readers:** Specifies the allocation behavior and limits for collections dependent on the total number of readers per participant.
- **writers:** Specifies the allocation behavior and limits for collections dependent on the total number of writers per participant.
- **send_buffers:** Defines the allocation behavior and limits for the send buffer manager.

- `data_limits`: States the limits for variable-length data.
- `content_filter`: States the limits for content-filter discovery information.

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

It cannot be changed on enabled entities.

RemoteLocatorsAllocationAttributes

This structure holds the limits for the remote locators' collections. See `RemoteLocatorsAllocationAttributes`.

List of structure members:

Member Name	Type	Default Value
<code>max_unicast_locators</code>	<code>size_t</code>	4
<code>max_multicast_locators</code>	<code>size_t</code>	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
<code>initial</code>	<code>size_t</code>	0
<code>maximum</code>	<code>size_t</code>	<code>std::numeric_limits<size_t>::max()</code>
<code>increment</code>	<code>size_t</code>	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
<code>preallocated_number</code>	<code>size_t</code>	0
<code>dynamic</code>	<code>bool</code>	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
<code>max_properties</code>	<code>size_t</code>	0
<code>max_user_data</code>	<code>size_t</code>	0
<code>max_partitions</code>	<code>size_t</code>	0

- `max_properties`: Defines the maximum size, in octets, of the properties data in the local or remote participant.
- `max_user_data`: Establishes the maximum size, in octets, of the user data in the local or remote participant.
- `max_partitions`: States the maximum size, in octets, of the partitions data in the local or remote participant.

ContentFilterProperty::AllocationConfiguration

This structure holds the limits for content-filter related discovery information. See `ContentFilterProperty::AllocationConfiguration`.

List of structure members:

Member Name	Type	Default Value
<code>expression_initial_size</code>	<code>size_t</code>	0
<code>expression_parameters</code>	<i><code>ResourceLimitedContainerConfig</code></i>	{0, 100, 1}

- `expression_initial_size`: Preallocated size of the filter expression.
- `expression_parameters`: Allocation configuration for the list of expression parameters.

Example

C++

```
ParticipantResourceLimitsQos participant_limits;
//Set the maximum size of participant resource limits collection to 3 and its allocation
↳configuration to fixed size
participant_limits.participants =
↳eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(
    3u);
//Set the maximum size of reader's resource limits collection to 2 and its allocation
↳configuration to fixed size
participant_limits.readers = eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_
↳size_configuration(2u);
//Set the maximum size of writer's resource limits collection to 1 and its allocation
↳configuration to fixed size
participant_limits.writers = eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_
↳size_configuration(1u);
//Set the maximum size of the partition data to 256
participant_limits.data_limits.max_partitions = 256u;
//Set the maximum size of the user data to 256
participant_limits.data_limits.max_user_data = 256u;
//Set the maximum size of the properties data to 512
participant_limits.data_limits.max_properties = 512u;
//Set the preallocated filter expression size to 512
participant_limits.content_filter.expression_initial_size = 512u;
//Set the maximum number of expression parameters to 4 and its allocation configuration
↳to fixed size
participant_limits.content_filter.expression_parameters =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(4u);
```

XML

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

(continues on next page)

(continued from previous page)

```

        <increment>0</increment>
    </total_readers>
    <!-- We know we have at most 1 writer on each participant -->
    <total_writers>
        <initial>1</initial>
        <maximum>1</maximum>
        <increment>0</increment>
    </total_writers>
    <max_partitions>256</max_partitions>
    <max_user_data>256</max_user_data>
    <max_properties>512</max_properties>

    <!-- content_filter cannot be configured using XML (yet) -->
</allocation>
</rtps>
</participant>

```

PropertyPolicyQos

This additional QoS Policy (PropertyPolicyQos) stores name/value pairs that can be used to configure certain DDS settings that cannot be configured directly using a standard QoS Policy. For the complete list of settings that can be configured with this QoS Policy, please refer to *PropertyPolicyQos Options*.

This QoS also allows to add custom user properties that could be sent to the external entities. This could be done by setting as true the propagate value of the Property.

Example

C++

```

PropertyPolicyQos property_policy;
//Add new property for the Auth:PKI-DH plugin
property_policy.properties().emplace_back("dds.sec.auth.plugin", "builtin.PKI-DH");
//Add new property for the Access:Permissions plugin
property_policy.properties().emplace_back(eprosima::fastrtps::rtps::Property("dds.sec.
↪access.plugin",
    "builtin.Access-Permissions"));

//Add new user custom property to send to external Participants
property_policy.properties().emplace_back("Custom Property Name", "Custom value", true);

```


XML

```

<participant profile_name="secure_participant_conf_all_plugin_xml_profile">
  <rtps>
    <propertiesPolicy>
      <properties>
        <!-- Activate Auth:PKI-DH plugin -->
        <property>
          <name>dds.sec.auth.plugin</name>
          <value>builtin.PKI-DH</value>
        </property>

        <!-- Activate Access:Permissions plugin -->
        <property>
          <name>dds.sec.access.plugin</name>
          <value>builtin.Access-Permissions</value>
        </property>

        <!-- User Custom Property to send externally -->
        <property>
          <name>Custom Property Name</name>
          <value>Custom value</value>
          <propagate>true</propagate>
        </property>
      </properties>
    </propertiesPolicy>
  </rtps>
</participant>

```

PublishModeQosPolicy

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

It also configures the name of the flow controller to use when asynchronous publishing is used. It should be the name of a flow controller registered on the creation of the DomainParticipant. See *FlowControllersQos*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	<i>PublishModeQosPolicyKind</i>	SYNCHRONOUS_PUBLISH_MODE
flow_controller_name	const char *	FASTDDS_FLOW_CONTROLLER_DEFAULT

Note: This QoS Policy concerns to DataWriter entities.

It cannot be changed on enabled entities.

PublishModeQosPolicyKind

There are two possible values (see `PublishModeQosPolicyKind`):

- `SYNCHRONOUS_PUBLISH_MODE`: The data is sent in the context of the user thread that calls the write operation.
- `ASYNCHRONOUS_PUBLISH_MODE`: An internal thread takes the responsibility of sending the data asynchronously. The write operation returns before the data is actually sent.

Example

C++

```
PublishModeQosPolicy publish_mode;  
//The PublishModeQosPolicy is default constructed with kind = SYNCHRONOUS  
//Change the kind to ASYNCHRONOUS  
publish_mode.kind = ASYNCHRONOUS_PUBLISH_MODE;
```

XML

```
<data_writer profile_name="writer_profile_qos_publishmode">  
  <qos>  
    <publishMode>  
      <kind>ASYNCHRONOUS</kind>  
    </publishMode>  
  </qos>  
</data_writer>
```

ReaderResourceLimitsQos

This QoS Policy states the limits for the matched *DataWriters*' resource limited collections based on the maximum number of DataWriters that are going to match with the *DataReader*. See `ReaderResourceLimitsQos`.

List of QoS Policy data members:

Data Member Name	Type
<code>matched_publisher_allocation</code>	<i>ResourceLimitedContainerConfig</i>

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::fastdds::ResourceLimitedContainerConfig::fixed_size_configuration(1u);
```

XML

```
<data_reader profile_name="alloc_qos_example_sub">
  <!-- we know we will only have one matching publisher -->
  <matchedPublishersAllocation>
    <initial>1</initial>
    <maximum>1</maximum>
    <increment>0</increment>
  </matchedPublishersAllocation>
</data_reader>
```

RTPSEndpointQos

This QoS Policy configures the aspects of an RTPS endpoint, such as the list of locators, the identifiers, and the history memory policy. See RTPSEndpointQos.

List of QoS Policy data members:

Data Member Name	Type	Default Value
unicast_locator_list	LocatorList	Empty List
multicast_locator_list	LocatorList	Empty List
remote_locator_list	LocatorList	Empty List
external_unicast_locators	ExternalLocators	Empty
ignore_non_matching_locators	bool	false
user_defined_id	int16_t	-1
entity_id	int16_t	-1
history_memory_policy	<i>MemoryManagementPolicy</i>	PREALLOCATED_WITH_REALLOC_MEMORY_MODE

- **unicast_locator_list**: Defines the list of unicast locators associated to the DDS Entity. DataReaders and DataWriters inherit the list of unicast locators set in the DomainParticipant, but it can be changed by means of this QoS.
- **multicast_locator_list**: Stores the list of multicast locators associated to the DDS Entity. By default, DataReaders and DataWriters do not use any multicast locator, but it can be changed by means of this QoS.
- **remote_locator_list**: States the list of remote locators associated to the DDS Entity.
- **external_unicast_locators**: Defines the *External Locators* to announce for the communication with this DDS Entity.

- `ignore_non_matching_locators`: Defines whether to ignore locators received on announcements from other DDS entities when they don't match with any of the locators announced by this DDS Entity.
- `user_defined_id`: Establishes the unique identifier used for `StaticEndpointDiscovery`.
- `entity_id`: The user can specify the identifier for the endpoint.
- `history_memory_policy`: Indicates the way the memory is managed in terms of dealing with the `CacheChanges`.

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

It cannot be changed on enabled entities.

MemoryManagementPolicy

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

- `PREALLOCATED_MEMORY_MODE`: This option sets the size to the maximum of each data type. It produces the largest memory footprint but the smallest allocation count.
- `PREALLOCATED_WITH_REALLOC_MEMORY_MODE`: This option set the size to the default for each data type and it requires reallocation when a bigger message arrives. It produces a lower memory footprint at the expense of increasing the allocation count.
- `DYNAMIC_RESERVE_MEMORY_MODE`: This option allocates the size dynamically at the time of message arrival. It produces the least memory footprint but the highest allocation count.
- `DYNAMIC_REUSABLE_MEMORY_MODE`: This option is similar to `DYNAMIC_RESERVE_MEMORY_MODE`, but the allocated memory is reused for future messages.

Example

C++

```
RTPSEndpointQos endpoint;
//Add new unicast locator with port 7800
eprosima::fastrtps::rtps::Locator_t new_unicast_locator;
new_unicast_locator.port = 7800;
endpoint.unicast_locator_list.push_back(new_unicast_locator);
//Add new multicast locator with IP 239.255.0.4 and port 7900
eprosima::fastrtps::rtps::Locator_t new_multicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(new_multicast_locator, "239.255.0.4");
new_multicast_locator.port = 7900;
endpoint.multicast_locator_list.push_back(new_multicast_locator);
// Add an external locator with IP 100.100.100.10, port 12345, mask 24, externality 1,
// and cost 0
eprosima::fastdds::rtps::LocatorWithMask external_locator;
external_locator.kind = LOCATOR_KIND_UDPv4;
external_locator.port = 12345;
external_locator.mask(24);
endpoint.external_unicast_locators[1][0].push_back(external_locator);
// Drop non matching locators
```

(continues on next page)

(continued from previous page)

```

endpoint.ignore_non_matching_locators = true;
//Set 3 as user defined id
endpoint.user_defined_id = 3;
//Set 4 as entity id
endpoint.entity_id = 4;
//The RTPSEndpointQos is default constructed with history_memory_policy = PREALLOCATED
//Change the history_memory_policy to DYNAMIC_RESERVE
endpoint.history_memory_policy = eprosima::fastrtps::rtps::DYNAMIC_RESERVE_MEMORY_MODE;

```

XML

```

<data_writer profile_name="writer_xml_conf_unicast_locators_profile">
  <userDefinedID>3</userDefinedID>
  <entityID>2</entityID> <!-- Int16 -->
  <unicastLocatorList>
    <locator>
      <udp4>
        <port>7800</port>
      </udp4>
    </locator>
  </unicastLocatorList>
  <multicastLocatorList>
    <locator>
      <udp4>
        <address>239.255.0.4</address>
        <port>7900</port>
      </udp4>
    </locator>
  </multicastLocatorList>
  <external_unicast_locators>
    <udp4 externality="1" cost="0" mask="24">
      <address>100.100.100.10</address>
      <port>12345</port>
    </udp4>
  </external_unicast_locators>
  <ignore_non_matching_locators>true</ignore_non_matching_locators>
  <!-- The history memory policy is changed to DYNAMIC_RESERVE -->
  <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</data_writer>

<data_reader profile_name="reader_xml_conf_unicast_locators_profile">
  <userDefinedID>5</userDefinedID>
  <entityID>4</entityID> <!-- Int16 -->
  <unicastLocatorList>
    <locator>
      <udp4>
        <port>7800</port>
      </udp4>
    </locator>
  </unicastLocatorList>

```

(continues on next page)

(continued from previous page)

```

<multicastLocatorList>
  <locator>
    <udp4>
      <address>239.255.0.4</address>
      <port>7900</port>
    </udp4>
  </locator>
</multicastLocatorList>
<external_unicast_locators>
  <udp4 externality="1" cost="0" mask="24">
    <address>100.100.100.10</address>
    <port>12345</port>
  </udp4>
</external_unicast_locators>
<ignore_non_matching_locators>true</ignore_non_matching_locators>
<historyMemoryPolicy>PREALLOCATED_WITH_REALLOC</historyMemoryPolicy>
</data_reader>

```

RTPSReliableReaderQos

This RTPS QoS Policy allows the configuration of several RTPS reliable reader's aspects. See `RTPSReliableReaderQos`.

List of QoS Policy data members:

Data Member Name	Type
<code>times</code>	<i>ReaderTimes</i>
<code>disable_positive_ACKs</code>	<i>DisablePositiveACKsQosPolicy</i>

- `times`: Defines the duration of the RTPSReader events. See *ReaderTimes* for further details.
- `disable_positive_ACKs`: Configures the settings to disable the positive acks. See *DisablePositiveACKsQosPolicy* for further details.

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

Only the duration Data Member of the *DisablePositiveACKsQosPolicy* and the `times` Data Member can be modified on enabled entities.

ReaderTimes

This structure defines the times associated with the Reliable Readers' events. See `ReaderTimes`.

List of structure members:

Member Name	Type	Default Value
<code>initialAcknackDelay</code>	<code>Duration_t</code>	70 ms
<code>heartbeatResponseDelay</code>	<code>Duration_t</code>	5 ms

- `initialAcknackDelay`: Defines the duration of the initial acknack delay.
- `heartbeatResponseDelay`: Establishes the duration of the delay applied when a heartbeat message is received.

Example

C++

```
RTPSReliableReaderQos reliable_reader_qos;
//The RTPSReliableReaderQos is default constructed with initialAcknackDelay = 70 ms
//Change the initialAcknackDelay to 70 nanoseconds
reliable_reader_qos.times.initialAcknackDelay = {0, 70};
//The RTPSReliableWriterQos is default constructed with heartbeatResponseDelay = 5 ms
//Change the heartbeatResponseDelay to 5 nanoseconds
reliable_reader_qos.times.heartbeatResponseDelay = {0, 5};
//You can also change the DisablePositiveACKsQosPolicy. For further details see
//DisablePositiveACKsQosPolicy section.
reliable_reader_qos.disable_positive_ACKs.enabled = true;
```

XML

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

RTPSReliableWriterQos

This RTPS QoS Policy allows the configuration of several RTPS reliable writer's aspects. See `RTPSReliableWriterQos`.

List of QoS Policy data members:

Data Member Name	Type
<code>times</code>	<i>WriterTimes</i>
<code>disable_positive_acks</code>	<i>DisablePositiveACKsQosPolicy</i>
<code>disable_heartbeat_piggyback</code>	<i>DisableHeartbeatPiggyback</i>

- `times`: Defines the duration of the RTPSWriter events. See *WriterTimes* for further details.
- `disable_positive_acks`: Configures the settings to disable the positive acks. See *DisablePositiveACKsQosPolicy* for further details.

- `disable_heartbeat_piggyback`: Configures the settings to disable the heartbeat piggyback mechanism. See [DisableHeartbeatPiggyback](#) for further details.

Note: This QoS Policy concerns to [DataWriter](#) entities.

Only the `duration` Data Member of the [DisablePositiveACKsQosPolicy](#) and the `times` Data Member can be modified on enabled entities.

WriterTimes

This structure defines the times associated with the Reliable Writers' events.

List of structure members:

Member Name	Type	Default Value
<code>initialHeartbeatDelay</code>	<code>Duration_t</code>	12ms
<code>heartbeatPeriod</code>	<code>Duration_t</code>	3s
<code>nackResponseDelay</code>	<code>Duration_t</code>	5ms
<code>nackSupressionDuration</code>	<code>Duration_t</code>	0s

- `initialHeartbeatDelay`: Defines duration of the initial heartbeat delay.
- `heartbeatPeriod`: Specifies the interval between periodic heartbeats.
- `nackResponseDelay`: Establishes the duration of the delay applied to the response of an ACKNACK message.
- `nackSupressionDuration`: The RTPSWriter ignores the nack messages received after sending the data until the duration time elapses.

DisableHeartbeatPiggyback

Besides sending heartbeats periodically using the `heartbeatPeriod` (see [WriterTimes](#)), reliable DataWriters also use a mechanism to append a heartbeat submessage in the same message where data is being delivered to the DataReaders. This mechanism acts in specific situations where the reliable communication state must be up to date to maintain optimal communication:

- When the DataWriter sends as many bytes to the *socket* as the length of the *socket* buffer, a heartbeat submessage is appended after the last data.
- When the DataWriter's history is full, the DataWriter starts to append heartbeat submessages after each data.

This mechanism can be disabled using this policy.

Example

C++

```
RTPSReliableWriterQos reliable_writer_qos;
//The RTPSReliableWriterQos is default constructed with initialHeartbeatDelay = 12 ms
//Change the initialHeartbeatDelay to 20 nanoseconds
reliable_writer_qos.times.initialHeartbeatDelay = {0, 20};
```

(continues on next page)

(continued from previous page)

```
//The RTPSReliableWriterQos is default constructed with heartbeatPeriod = 3 s
//Change the heartbeatPeriod to 5 seconds
reliable_writer_qos.times.heartbeatPeriod = {5, 0};
//The RTPSReliableWriterQos is default constructed with nackResponseDelay = 5 ms
//Change the nackResponseDelay to 10 nanoseconds
reliable_writer_qos.times.nackResponseDelay = {0, 10};
//The RTPSReliableWriterQos is default constructed with nackSupressionDuration = 0 s
//Change the nackSupressionDuration to 20 nanoseconds
reliable_writer_qos.times.nackSupressionDuration = {0, 20};
//You can also change the DisablePositiveACKsQosPolicy. For further details see
↳ DisablePositiveACKsQosPolicy section.
reliable_writer_qos.disable_positive_acks.enabled = true;
//The RTPSReliableWriterQos is default constructed with disable_heartbeat_piggyback =
↳ false
//Disable the heartbeat piggyback mechanism.
reliable_writer_qos.disable_heartbeat_piggyback = true;
```

XML

```
<data_writer profile_name="pub_profile_name">
  <times> <!-- writerTimesType -->
    <initialHeartbeatDelay> <!-- DURATION -->
      <nanosec>20</nanosec>
    </initialHeartbeatDelay>
    <heartbeatPeriod> <!-- DURATION -->
      <sec>5</sec>
    </heartbeatPeriod>
    <nackResponseDelay> <!-- DURATION -->
      <nanosec>10</nanosec>
    </nackResponseDelay>
    <nackSupressionDuration> <!-- DURATION -->
      <nanosec>20</nanosec>
    </nackSupressionDuration>
  </times>

  <!--You can also change the values of DisablePositiveACKsQosPolicy.-->
  <!--See DisablePositiveACKsQosPolicy section for further details-->
  <qos>
    <!--Disable heartbeat piggyback mechanism.-->
    <disable_heartbeat_piggyback>true</disable_heartbeat_piggyback>
  </qos>
</data_writer>
```

ThreadSettings

This structure is part of other QoS policies, and allows controlling some OS settings for the threads created. The default values will leave the default OS settings on the created threads. Changing these values may require special permissions.

Data Member Name	Type	Default Value
<code>scheduling_policy</code>	<code>int32_t</code>	-1
<code>priority</code>	<code>int32_t</code>	-2 ³¹
<code>affinity</code>	<code>uint32_t</code>	0
<code>stack_size</code>	<code>int32_t</code>	-1

- `scheduling_policy`: Configures the scheduling policy used for the thread. This value is platform specific and it is used as-is to configure the specific platform thread. It is ignored on Windows platforms.
- `priority`: Configures the thread's priority. This value is platform specific and it is used as-is to configure the specific platform thread.
- `affinity`: On some systems (Windows, Linux), this is a bit mask for setting the threads affinity to each core individually. On MacOS, this sets the affinity tag for the thread, and the OS tries to share the L2 cache between threads with the same affinity. This value is platform specific and it is used as-is to configure the specific platform thread.
- `stack_size`: Configures the thread's stack size in bytes. This value is platform specific and it is used as-is to configure the specific platform thread.

The *ThreadSettings* for reception threads in port-based transport descriptors can also be configured by associating each thread-specific configuration with its corresponding port number.

Example

The following example illustrate a thread settings configuration:

C++

```
ThreadSettings thread_settings;  
thread_settings.scheduling_policy = 2;  
thread_settings.priority = 10;  
thread_settings.affinity = 4;  
thread_settings.stack_size = 2000;
```

XML

```
<thread_settings>  
  <scheduling_policy>2</scheduling_policy>  
  <priority>10</priority>  
  <affinity>4</affinity>  
  <stack_size>2000</stack_size>  
</thread_settings>
```

The subsequent example depicts a reception threads settings configuration:

C++

```
PortBasedTransportDescriptor::ReceptionThreadsConfigMap reception_threads_config;
reception_threads_config[20000].scheduling_policy = 1;
reception_threads_config[20000].priority = 30;
reception_threads_config[20000].affinity = 2;
reception_threads_config[20000].stack_size = 1024;
reception_threads_config[20001].scheduling_policy = 2;
reception_threads_config[20001].priority = 10;
reception_threads_config[20001].affinity = 6;
reception_threads_config[20001].stack_size = 4096;
```

XML

```
<transport_descriptors>
  <transport_descriptor>
    <transport_id>test_transport</transport_id>
    <type>UDIPv4</type>
    <reception_threads>
      <reception_thread port="20000">
        <scheduling_policy>1</scheduling_policy>
        <priority>30</priority>
        <affinity>2</affinity>
        <stack_size>1024</stack_size>
      </reception_thread>
      <reception_thread port="20001">
        <scheduling_policy>2</scheduling_policy>
        <priority>10</priority>
        <affinity>6</affinity>
        <stack_size>4096</stack_size>
      </reception_thread>
    </reception_threads>
  </transport_descriptor>
</transport_descriptors>
```

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<TransportDescriptor>	Empty vector
use_builtin_transports	bool	true
send_socket_buffer_size	uint32_t	0
listen_socket_buffer_size	uint32_t	0
builtin_transports_reception_thread_settings	ThreadSettings	

- 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 UDIPv4 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.
- `builtin_transports_reception_threads()`: The *ThreadSettings* for the reception threads of the builtin transports.

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
<code>maxMessageSize</code>	<code>uint32_t</code>
<code>maxInitialPeersRange</code>	<code>uint32_t</code>

- `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;
// [OPTIONAL] Set ThreadSettings for the builtin transports reception threads
transport.builtin_transports_reception_threads_ = eprosima::fastdds::rtps::ThreadSettings
↳ {2, 2, 2, 2};
```

XML

```

<transport_descriptors>
  <transport_descriptor>
    <transport_id>my_transport</transport_id>
    <type>UDPv4</type>
    <sendBufferSize>9216</sendBufferSize>
    <receiveBufferSize>9216</receiveBufferSize>
  </transport_descriptor>
</transport_descriptors>

<participant profile_name="my_transport">
  <rtps>
    <userTransports>
      <transport_id>my_transport</transport_id>
    </userTransports>
    <useBuiltinTransports>false</useBuiltinTransports>
  </rtps>
</participant>

```

Note: *TransportConfigQos* can also be configured modifying the builtin transports configuration by selecting one of the available builtin transports options. See *Managing the Builtin Transports* or `setup_transports()`.

TypeConsistencyQos

This QoS Policy allows the configuration of the *XTypes extension QoS* on the *DataReader*. See *TypeConsistencyQos*.

List of QoS Policy data members:

Data Member Name	Type
<code>type_consistency</code>	<i>TypeConsistencyEnforcementQosPolicy</i>
<code>representation</code>	<i>DataRepresentationQosPolicy</i>

- `type_consistency`: It states the rules for the data types compatibility. See *TypeConsistencyEnforcementQosPolicy* for further details.
- `representation`: It specifies the data representations valid for the entities. See *DataRepresentationQosPolicy* for further details.

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

It cannot be changed on enabled entities.

Example

C++

```
TypeConsistencyQos consistency_qos;
//You can change the DataRepresentationQosPolicy. For further details see DataRepresentationQosPolicySection section.
consistency_qos.representation.m_value.push_back(DataRepresentationId_t::XCDR2_DATA_
REPRESENTATION);
//You can change the TypeConsistencyEnforcementQosPolicy. For further details see TypeConsistencyEnforcementQosPolicy section.
consistency_qos.type_consistency.m_kind = TypeConsistencyKind::ALLOW_TYPE_COERCION;
```

XML

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

WireProtocolConfigQos

This QoS Policy allows the configuration of the wire protocol. See `WireProtocolConfigQos`.

List of QoS Policy data members:

Data Member Name	Type	Default Value
prefix	GuidPrefix_t	0
participant_id	int32_t	-1
builtin	BuiltinAttributes	
port	PortParameters	
default_unicast_locator_list	LocatorList	Empty List
default_multicast_locator_list	LocatorList	Empty List
default_external_unicast_locators	ExternalLocators	Empty
ignore_non_matching_locators	bool	false

- **prefix**: This data member allows the user to set manually the GUID prefix.
- **participant_id**: It sets the participant identifier. By default, it will be automatically generated by the Domain.
- **builtin**: This data member allows the configuration of the built-in parameters.
- **port**: This data member allows the configuration of the port parameters and gains related to the RTPS protocol (*Well Known Ports*).
- **default_unicast_locator_list**: States the default list of unicast locators to be used for any endpoint defined inside the RTPSParticipant in the case that it was defined without unicast locators. This list should include at least one locator.
- **default_multicast_locator_list**: Stores the default list of multicast locators to be used for any endpoint defined inside the RTPSParticipant in the case that it was defined without multicast locators. This list is usually left empty.
- **default_external_unicast_locators**: Defines the *External Locators* to be used for any endpoint defined inside the participant in the case that it was defined without unicast locators.

- `ignore_non_matching_locators`: Defines whether to ignore locators received on announcements from other DDS participants when they don't match with any of the locators announced by this DDS participant.

Note: This QoS Policy concerns to DomainParticipant entities.

Important: The only mutable field on enabled entities is `m_DiscoveryServers`, which is contained in `discovery_config` within `builtin` (see *Modifying remote servers list at run time*).

Example

C++

```
WireProtocolConfigQos wire_protocol;
//Set the guid prefix
std::istringstream("72.61.73.70.66.61.72.6d.74.65.73.74") >> wire_protocol.prefix;
//Configure Builtin Attributes
wire_protocol.builtin.discovery_config.discoveryProtocol =
    eprosima::fastrtps::rtps::DiscoveryProtocol_t::SERVER;
//Add locator to unicast list
eprosima::fastrtps::rtps::Locator_t server_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(server_locator, "192.168.10.57");
server_locator.port = 56542;
wire_protocol.builtin.metatrafficUnicastLocatorList.push_back(server_locator);
// Add a metatraffic external locator with IP 100.100.100.10, port 34567, mask 24,
↳externality 1, and cost 0
eprosima::fastdds::rtps::LocatorWithMask meta_external_locator;
meta_external_locator.kind = LOCATOR_KIND_UDPv4;
meta_external_locator.port = 34567;
meta_external_locator.mask(24);
wire_protocol.builtin.metatraffic_external_unicast_locators[1][0].push_back(meta_
↳external_locator);
//Add locator to default unicast locator list
eprosima::fastrtps::rtps::Locator_t unicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(unicast_locator, 192, 168, 1, 41);
unicast_locator.port = 7400;
wire_protocol.default_unicast_locator_list.push_back(unicast_locator);
//Add locator to default multicast locator list
eprosima::fastrtps::rtps::Locator_t multicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(multicast_locator, 192, 168, 1, 41);
multicast_locator.port = 7400;
wire_protocol.default_multicast_locator_list.push_back(multicast_locator);
// Add a default external locator with IP 100.100.100.10, port 23456, mask 24,
↳externality 1, and cost 0
eprosima::fastdds::rtps::LocatorWithMask external_locator;
external_locator.kind = LOCATOR_KIND_UDPv4;
external_locator.port = 23456;
external_locator.mask(24);
wire_protocol.default_external_unicast_locators[1][0].push_back(external_locator);
```

(continues on next page)

(continued from previous page)

```
// Drop non matching locators
wire_protocol.ignore_non_matching_locators = true;
```

XML

```
<participant profile_name="UDP SERVER WP" is_default_profile="true">
  <rtps>
    <prefix>72.61.73.70.66.61.72.6d.74.65.73.74</prefix>
    <builtin>
      <discovery_config>
        <discoveryProtocol>SERVER</discoveryProtocol>
      </discovery_config>
      <metatrafficUnicastLocatorList>
        <locator>
          <udp4>
            <address>192.168.10.57</address>
            <port>56542</port>
          </udp4>
        </locator>
      </metatrafficUnicastLocatorList>
      <metatraffic_external_unicast_locators>
        <udp4 externality="1" cost="0" mask="24">
          <address>100.100.100.10</address>
          <port>34567</port>
        </udp4>
      </metatraffic_external_unicast_locators>
    </builtin>
    <defaultUnicastLocatorList>
      <locator>
        <udp4>
          <!-- Access as physical, like UDP -->
          <port>7400</port>
          <address>192.168.1.41</address>
        </udp4>
      </locator>
    </defaultUnicastLocatorList>

    <defaultMulticastLocatorList>
      <locator>
        <udp4>
          <!-- Access as physical, like UDP -->
          <port>7400</port>
          <address>192.168.1.41</address>
        </udp4>
      </locator>
    </defaultMulticastLocatorList>

    <default_external_unicast_locators>
      <udp4 externality="1" cost="0" mask="24">
        <address>100.100.100.10</address>
```

(continues on next page)

(continued from previous page)

```

        <port>23456</port>
    </udp4>
</default_external_unicast_locators>

    <ignore_non_matching_locators>true</ignore_non_matching_locators>
</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
<code>matched_subscriber_allocation</code>	<i>ResourceLimitedContainerConfig</i>
<code>reader_filters_allocation</code>	<i>ResourceLimitedContainerConfig</i>

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

It cannot be changed on enabled entities.

Example

C++

```

WriterResourceLimitsQos writer_limits;
//Set the maximum size for reader matched resource limits collection to 3 and its
↳allocation configuration to fixed size
writer_limits.matched_subscriber_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(3u);
// Set the maximum number of writer side content filters to 1 and its allocation
↳configuration to fixed size
writer_limits.reader_filters_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(1u);

```

XML

```
<data_writer profile_name="alloc_qos_example_pub_for_topic_1">
  <!-- we know we will have three matching subscribers -->
  <matchedSubscribersAllocation>
    <initial>3</initial>
    <maximum>3</maximum>
    <increment>0</increment>
  </matchedSubscribersAllocation>

  <!-- reader_filters_allocation cannot be configured using XML (yet) -->
</data_writer>
```

XTypes Extensions

This section explain those QoS Policy extensions defined in the [XTypes Specification](#):

- *DataRepresentationQosPolicy*
- *TypeConsistencyEnforcementQosPolicy*

DataRepresentationQosPolicy

This XTypes QoS Policy states which data representations will be used by the *DataWriters* and *DataReaders*.

The DataWriters offer a single data representation that will be used to communicate with the matched DataReaders. The DataReaders can request one or more data representations and in order to have communication with the DataWriter, the offered data representation needs to be contained within the DataReader request. See *DataRepresentationQosPolicy*.

List of QoS Policy data members:

Data Member Name	Type	Default Value
m_value	std::vector< <i>DataRepresentationId</i> >	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

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

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
<code>m_kind</code>	<i>TypeConsistencyKind</i>	<code>ALLOW_TYPE_COERCION</code>
<code>m_ignore_sequence_bounds</code>	<code>bool</code>	<code>true</code>
<code>m_ignore_string_bounds</code>	<code>bool</code>	<code>true</code>
<code>m_ignore_member_names</code>	<code>bool</code>	<code>true</code>
<code>m_prevent_type_widening</code>	<code>bool</code>	<code>true</code>
<code>m_force_type_validation</code>	<code>bool</code>	<code>true</code>

- `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 assignments  
type_enforcement.m_ignore_sequence_bounds = true;  
//Configures the system to ignore the string sizes in assignments  
type_enforcement.m_ignore_string_bounds = true;  
//Configures the system to ignore the member names. Members with same ID could have_  
↪different names  
type_enforcement.m_ignore_member_names = true;  
//Configures the system to allow type widening  
type_enforcement.m_prevent_type_widening = false;  
//Configures the system to not use the complete Type Information in entities match_  
↪process  
type_enforcement.m_force_type_validation = false;
```

XML

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

Status

Each *Entity* is associated with a set of Status objects whose values represent the *communication status* of that Entity. Changes on the status values occur due to communication events related to each of the entities, e.g., when new data arrives, a new participant is discovered, or a remote endpoint is lost. The status is decomposed into several status objects, each concerning a different aspect of the communication, so that each of these status objects can vary independently of the others.

Changes on a status object trigger the corresponding *Listener* callbacks that allow the Entity to inform the application about the event. For a given status object with name `fooStatus`, the entity listener interface defines a callback function `on_foo()` that will be called when the status changes. Beware that some statuses have data members that are reset every time the corresponding listener is called. The only exception to this rule is when the entity has no listener attached, so the callback cannot be called. See the documentation of each status for details.

Conditions and Wait-sets provide the application with an alternative mechanism to make it aware of changes on status objects, by means of a *StatusCondition*. The advantage of this mechanism is that the application can wait for changes on several entities at the same time. It will also help the determinism of your system, as the notification is not processed on an internal thread, as it is done when using listeners.

The entities expose functions to access the value of its statuses. For a given status with name `fooStatus`, the entity exposes a member function `get_foo()` to access the data in its `fooStatus`. The only exceptions are *DataOnReaders* and *DataAvailable*. These getter functions return a read-only struct where all data members are public and accessible to the application. Beware that some statuses have data members that are reset every time the getter function is called by the application. See the documentation of each status for details.

The following subsections describe each of the status objects, their data members, and to which Entity type they concern. The next table offers a quick reference as well as the corresponding bit for each status in the `StatusMask`.

Status Name	Entity	Listener callback	Accessor	Bit
<i>InconsistentTopicStatus</i>	<i>Topic</i>	<code>on_inconsistent_topic()</code>	<code>get_inconsistent_topic_status()</code>	0
<i>OfferedDeadline-MissedStatus</i>	<i>DataWrite</i>	<code>on_offered_deadline_missed()</code>	<code>get_offered_deadline_missed_status()</code>	1
<i>RequestedDeadline-MissedStatus</i>	<i>DataRead</i>	<code>on_requested_deadline_missed()</code>	<code>get_requested_deadline_missed_status()</code>	2
<i>OfferedIncompatible-QosStatus</i>	<i>DataWrite</i>	<code>on_offered_incompatible_qos()</code>	<code>get_offered_incompatible_qos_status()</code>	5
<i>RequestedIncompatible-QosStatus</i>	<i>DataRead</i>	<code>on_requested_incompatible_qos()</code>	<code>get_requested_incompatible_qos_status()</code>	6
<i>SampleLostStatus</i>	<i>DataRead</i>	<code>on_sample_lost()</code>	<code>get_sample_lost_status()</code>	7
<i>SampleRejectedStatus</i>	<i>DataRead</i>	<code>on_sample_rejected()</code>	<code>get_sample_rejected_status()</code>	8
<i>DataOnReaders</i>	<i>Subscriber</i>	<code>on_data_on_readers()</code>	N/A	9
<i>DataAvailable</i>	<i>DataRead</i>	<code>on_data_available()</code>	N/A	10
<i>LivelinessLostStatus</i>	<i>DataWrite</i>	<code>on_liveliness_lost()</code>	<code>get_liveliness_lost_status()</code>	11
<i>LivelinessChangedStatus</i>	<i>DataRead</i>	<code>on_liveliness_changed()</code>	<code>get_liveliness_changed_status()</code>	12
<i>PublicationMatched-Status</i>	<i>DataWrite</i>	<code>on_publication_matched()</code>	<code>get_publication_matched_status()</code>	13
<i>SubscriptionMatched-Status</i>	<i>DataRead</i>	<code>on_subscription_matched()</code>	<code>get_subscription_matched_status()</code>	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
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>

- `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
<code>alive_count</code>	<code>int32_t</code>
<code>not_alive_count</code>	<code>int32_t</code>
<code>alive_count_change</code>	<code>int32_t</code>
<code>not_alive_count_change</code>	<code>int32_t</code>
<code>last_publication_handle</code>	<code>InstanceHandle_t</code>

- `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 assert 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 be incremented by one for each instance for which data was not received in the deadline period.
- **total_count_change**: The change in **total_count** since the last time **on_requested_deadline_missed()** was called or the status was read. It can only have zero or positive values.
- **last_instance_handle**: Handle to the last instance that missed the deadline. If no deadline was ever missed, it will have value **c_InstanceHandle_Unknown**.

RequestedIncompatibleQosStatus

This status changes every time the DataReader finds a DataWriter that matches the Topic and has a common partition, but with a QoS configuration incompatible with the one defined on the DataReader. See **RequestedIncompatibleQosStatus**.

List of status data members:

Data Member Name	Type
total_count	int32_t
total_count_change	int32_t
last_policy_id	QosPolicyId_t
policies	QosPolicyCountSeq

- `total_count`: Total cumulative count of DataWriters found matching the Topic and with a common partition, but with a QoS configuration that is incompatible with the one defined on the DataReader.
- `total_count_change`: The change in `total_count` since the last time `on_requested_incompatible_qos()` was called or the status was read. It can only have zero or positive values.
- `last_policy_id`: The policy ID of one of the policies that was found to be incompatible with the current DataReader. If more than one policy happens to be incompatible, only one of them will be reported in this member.
- `policies`: A collection that holds, for each policy, the total number of times that the policy was found to be incompatible with the one offered by a remote DataWriter that matched the Topic and with a common partition. See [QosPolicyCountSeq](#) and [QosPolicyCount](#) for more information the information that is stored for each policy.

QosPolicyCountSeq

Holds a [QosPolicyCount](#) for each [Policy](#), indexed by its `QosPolicyId_t`. Therefore, the Qos Policy with ID `N` will be at position `N` in the sequence. See [QosPolicyCountSeq](#).

```
DataReader* data_reader =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);

// Get how many times ReliabilityQosPolicy was not compatible with a remote writer
RequestedIncompatibleQosStatus status;
data_reader->get_requested_incompatible_qos_status(status);
uint32_t incompatible_reliability_count = status.policies[RELIABILITY_QOS_POLICY_ID].
    ↪count;
```

QosPolicyCount

This structure holds a counter for a policy. See [QosPolicyCount](#).

List of data members:

Data Member Name	Type
<code>policy_id</code>	<code>QosPolicyId_t</code>
<code>count</code>	<code>int32_t</code>

- `policy_id`: The ID of the policy.
- `count`: The counter value for the policy.

SampleLostStatus

This status changes every time a new data sample is lost and will never be received. See `SampleLostStatus`.

There are two different criteria for considering a sample as lost depending on the `reliability()`:

- When using `BEST_EFFORT_RELIABILITY_QOS`, a not yet received sample is considered lost whenever a sample with a greater sequence number is received.
- When using `RELIABLE_RELIABILITY_QOS`, a not yet received sample is considered lost whenever the `DataWriter` informs, through an RTPS HEARTBEAT submessage, that the sample is not available anymore.

List of status data members:

Data Member Name	Type
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>

- `total_count`: Total cumulative count of lost samples under the Topic of the current `DataReader`.
- `total_count_change`: The change in `total_count` since the last time `on_sample_lost()` was called or the status was read. It can only be positive or zero.

SampleRejectedStatus

This status changes every time an incoming data sample is rejected by the `DataReader`. The reason for the rejection is defined by *[SampleRejectedStatusKind](#)*. For further information see `SampleRejectedStatus`.

List of status data members:

Data Member Name	Type
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>
<code>last_reason</code>	<code>SampleRejectedStatusKind</code>
<code>last_instance_handle</code>	<code>InstanceHandle_t</code>

- `total_count`: Total cumulative count of rejected samples under the Topic of the current `DataReader`.
- `total_count_change`: The change in `total_count` since the last time `on_sample_rejected()` was called or the status was read. It can only be positive or zero.
- `last_reason`: The reason for rejecting the last rejected sample. If no sample was ever rejected, it will have value `NOT_REJECTED`. See *[SampleRejectedStatusKind](#)* for further details.
- `last_instance_handle`: Handle to the last instance whose sample was rejected. If no sample was ever rejected, it will have value `c_InstanceHandle_Unknown`.

SampleRejectedStatusKind

In Fast DDS, samples can be rejected due to resource limit reasons. However, the fact that the samples are rejected does not imply that they are lost, i.e. a rejected sample may be accepted in the future.

`SampleRejectedStatusKind` specifies the reason of the rejection:

- `NOT_REJECTED` specifies that the samples were not rejected.
- `REJECTED_BY_SAMPLES_LIMIT` specifies that the samples were rejected because there were not enough resources to store them. This can happen even when there are free resources if those resources must be guaranteed to be available for other samples. This situation, which arises in the RTPS layer, occurs when there are yet to be received samples with lower sequence number and there is not enough resources for all of them (because `max_samples` has been reached).
- `REJECTED_BY_INSTANCES_LIMIT` specifies that the samples were rejected because there were not enough resources to allocate the samples' instances. This situation, which arises in the DDS layer, more precisely in the `DataReader`'s history, occurs when the sample corresponds to a new instance for which the middleware should reserve resources but the history's number of instances has already reached `max_instances`.
- `REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT` specifies that the samples were rejected because there were not enough resources within their instance to store them. This situation, which arises in the DDS layer, more precisely in the `DataReader`'s history, occurs when the `DataReader` is configured with `KEEP_ALL_HISTORY_QOS` and the instance's number of samples has reached `max_samples_per_instance`.

SubscriptionMatchedStatus

This status changes every time the `DataReader` finds a `DataWriter` that matches the `Topic` and has a common partition and a compatible QoS, or has ceased to be matched with a `DataWriter` that was previously considered to be matched. See `SubscriptionMatchedStatus`.

List of status data members:

Data Member Name	Type
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>
<code>current_count</code>	<code>int32_t</code>
<code>current_count_change</code>	<code>int32_t</code>
<code>last_publication_handle</code>	<code>InstanceHandle_t</code>

- `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
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>

- `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
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>
<code>last_instance_handle</code>	<code>InstanceHandle_t</code>

- `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 be incremented by one for each instance for which data was not sent in the deadline period.
- `total_count_change`: The change in `total_count` since the last time `on_offered_deadline_missed()` was called or the status was read. It can only have zero or positive values.
- `last_instance_handle`: Handle to the last instance that missed the deadline. If no deadline was ever missed, it will have value `c_InstanceHandle_Unknown`.

OfferedIncompatibleQosStatus

This status changes every time the DataWriter finds a DataReader that matches the Topic and has a common partition, but with a QoS configuration that is incompatible with the one defined on the DataWriter. See *OfferedIncompatibleQosStatus*.

List of status data members:

Data Member Name	Type
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>
<code>last_policy_id</code>	<code>QosPolicyId_t</code>
<code>policies</code>	<code>QosPolicyCountSeq</code>

- `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.

PublicationMatchedException

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 `PublicationMatchedException`.

List of status data members:

Data Member Name	Type
<code>total_count</code>	<code>int32_t</code>
<code>total_count_change</code>	<code>int32_t</code>
<code>current_count</code>	<code>int32_t</code>
<code>current_count_change</code>	<code>int32_t</code>
<code>last_subscription_handle</code>	<code>InstanceHandle_t</code>

- `total_count`: Total cumulative count of remote DataReaders that have been discovered publishing on the same Topic and has a common partition and a compatible QoS. They may not all be matched at the moment.
- `total_count_change`: The change in `total_count` since the last time `on_publication_matched()` was called or the status was read. It can only have zero or positive values.
- `current_count`: The number of remote DataReaders currently matched to the DataWriter.
- `current_count_change`: The change in `current_count` since the last time `on_publication_matched()` was called or the status was read. It can have positive or negative values.
- `last_subscription_handle`: Handle to the last DataReader that matched the DataWriter. If no matching ever happened, it will have value `c_InstanceHandle_Unknown`.

Conditions and Wait-sets

Conditions (in conjunction with wait-sets) provide an alternative mechanism to allow the middleware to notify communication status changes (including arrival of data) to the application.

This mechanism is wait-based. Its general use pattern is as follows:

- The application indicates which relevant information it wants to get, by means of *Condition* objects (*GuardCondition*, *StatusCondition*, or *ReadCondition*) and attaching them to a *Wait-set* via the `attach_condition()` call.
- It then waits on that *Wait-set* via the `wait()` call until the trigger value of one or several *Condition* objects become true.
- It then uses the result of the `wait()` (i.e., the list of *Condition* objects with `trigger_value == true`) to actually get the information by calling:
 - `get_status_changes()`, then checking if any of the changes is relevant using the `StatusMask::is_active()` method on the result and finally calling `get_<communication_status>` on the relevant Entity, when the condition is a *StatusCondition* and the status changes refer to plain communication status. Refer to *Status* for additional information on the different statuses that can be queried.
 - `get_status_changes()` and then `Subscriber::get_datareaders()` on the relevant Subscriber, when the condition is a *StatusCondition* and the status changes refer to *DataOnReaders*.
 - `get_status_changes()` and then `DataReader::read()/DataReader::take()` on the relevant DataReader, when the condition is a *StatusCondition* and the status changes refer to *DataAvailable*.
 - Directly `DataReader::read_w_condition()/DataReader::take_w_condition()` on the DataReader with the *Condition* as a parameter, when it is a *ReadCondition*
- When a Condition is no longer relevant it can be detached from a *Wait-set* via the `detach_condition()` call.

The first step is usually done in an initialization phase, while the others are put in the application main loop.

```
class ApplicationJob
{
    WaitSet wait_set_;
    GuardCondition terminate_condition_;
    std::thread thread_;

    void main_loop()
    {
        // Main loop is repeated until the terminate condition is triggered
        while (false == terminate_condition_.get_trigger_value())
        {
            // Wait for any of the conditions to be triggered
            ReturnCode_t ret_code;
            ConditionSeq triggered_conditions;
            ret_code = wait_set_.wait(triggered_conditions, eprosima::fastrtps::c_
↪TimeInfinite);
            if (RETCODE_OK != ret_code)
            {
                // ... handle error
                continue;
            }
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

// Process triggered conditions
for (Condition* cond : triggered_conditions)
{
    StatusCondition* status_cond = dynamic_cast<StatusCondition*>(cond);
    if (nullptr != status_cond)
    {
        Entity* entity = status_cond->get_entity();
        StatusMask changed_statuses = entity->get_status_changes();

        // Process status. Liveliness changed and data available are
        ↪depicted as an example
        if (changed_statuses.is_active(StatusMask::liveliness_changed()))
        {
            std::cout << "Liveliness changed reported for entity " <<
                entity->get_instance_handle() <<
                std::endl;
        }

        if (changed_statuses.is_active(StatusMask::data_available()))
        {
            std::cout << "Data available on reader " << entity->get_instance_
            ↪handle() << std::endl;

            FooSeq data_seq;
            SampleInfoSeq info_seq;
            DataReader* reader = static_cast<DataReader*>(entity);

            // Process all the samples until no one is returned
            while (RETCODE_OK == reader->take(data_seq, info_seq,
                LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
                ANY_VIEW_STATE, ANY_INSTANCE_STATE))
            {
                // Both info_seq.length() and data_seq.length() will have
                ↪the number of samples returned
                for (FooSeq::size_type n = 0; n < info_seq.length(); ++n)
                {
                    // Only samples with valid data should be accessed
                    if (info_seq[n].valid_data &&
                        reader->is_sample_valid(&data_seq[n], &info_
                    ↪seq[n]))
                    {
                        // Process sample on data_seq[n]
                    }
                }

                // must return the loaned sequences when done processing
                reader->return_loan(data_seq, info_seq);
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    }

public:

    ApplicationJob(
        const std::vector<DataReader*>& readers,
        const std::vector<DataWriter*>& writers)
    {
        // Add a GuardCondition, so we can signal the processing thread to stop
        wait_set_.attach_condition(terminate_condition_);

        // Add the status condition of every reader and writer
        for (DataReader* reader : readers)
        {
            wait_set_.attach_condition(reader->get_statuscondition());
        }
        for (DataWriter* writer : writers)
        {
            wait_set_.attach_condition(writer->get_statuscondition());
        }

        thread_ = std::thread(&ApplicationJob::main_loop, this);
    }

    ~ApplicationJob()
    {
        // Signal the GuardCondition to force the WaitSet to wake up
        terminate_condition_.set_trigger_value(true);
        // Wait for the thread to finish
        thread_.join();
    }
};

// Application initialization
ReturnCode_t ret_code;
std::vector<DataReader*> application_readers;
std::vector<DataWriter*> application_writers;

// Create the participant, topics, readers, and writers.
ret_code = create_dds_application(application_readers, application_writers);
if (RETCODE_OK != ret_code)
{
    // ... handle error
    return;
}

{
    ApplicationJob main_loop_thread(application_readers, application_writers);

    // ... wait for application termination signaling (signal handler, user input, etc)

```

(continues on next page)

(continued from previous page)

```
// ... Destructor of ApplicationJob takes care of stopping the processing thread
}  
  
// Destroy readers, writers, topics, and participant  
destroy_dds_application();
```

Calling the `wait()` operation on the *Wait-set* will block the calling thread if the trigger value of all the conditions attached to it are false. The thread will wake up, and the `wait()` operation will return `RETCODE_OK`, whenever the trigger value of any of the attached conditions becomes true.

GuardCondition

A condition for which the trigger value is completely controlled by the application via its `set_trigger_value()` operation.

StatusCondition

A condition that triggers whenever there are changes on the communication statuses of an Entity.

The sensitivity of the *StatusCondition* to a particular communication status is controlled by the list of `enabled_statuses` set on the condition by means of the `set_enabled_statuses()` operation.

ReadCondition

A condition that triggers whenever the `DataReader` that created it contains at least a sample with `SampleState`, `ViewState`, and `InstanceState` matching those of the `ReadCondition`.

The fact that the trigger value of a `ReadCondition` is dependent on the presence of samples on the associated `DataReader` implies that a single take operation can potentially change the trigger value of several `ReadCondition` conditions. For example, if all samples are taken, any `ReadCondition` associated with the `DataReader` that were triggered before, will see their trigger value changed to false. Note that this does not guarantee that `WaitSet` objects that were separately attached to those conditions will not be woken up. Once we have `trigger_value == true` on a condition, it may wake up the attached *Wait-set*. The condition transitioning to `trigger_value == false` does not necessarily ‘unwake up’ the *Wait-set*, as ‘unwakening’ may not be possible in general. The consequence is that an application blocked on a *Wait-set* may return from the wait with a list of conditions, some of which are no longer triggered. This also may be the consequence of user actions. A user manually calling `set_trigger_value()` could potentially trigger the same behavior. This is unavoidable if multiple threads are concurrently waiting on separate *Wait-set* objects and taking data associated with the same `DataReader` entity.

To elaborate further, consider the following example: A `ReadCondition` that has a `sample_state_mask = {NOT_READ}` will have `trigger_value == true` whenever a new sample arrives and will transition to false as soon as all the newly-arrived samples are either read (so their status changes to `READ`) or taken (so they are no longer managed by the `DataReader`). However, if the same `ReadCondition` had a `sample_state_mask = {READ, NOT_READ}`, then the `trigger_value` would only become false once all the newly-arrived samples are taken (it is not sufficient to read them as that would only change the `SampleState` to `READ` which overlaps the mask on the `ReadCondition`).

7.18.2 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
<i>UserDataQosPolicy</i>	<code>user_data()</code>	Yes
<i>EntityFactoryQosPolicy</i>	<code>entity_factory()</code>	Yes
<i>ParticipantResourceLimitsQos</i>	<code>allocation()</code>	No
<i>PropertyPolicyQos</i>	<code>properties()</code>	No
<i>WireProtocolConfigQos</i>	<code>wire_protocol()</code>	No*
<i>TransportConfigQos</i>	<code>transport()</code> and <code>setup_transports()</code>	No
<i>FlowControllersQos</i>	<code>flow_controllers()</code>	No
<i>ThreadSettings</i>	<code>builtin_controllers_sender_thread()</code>	No
<i>ThreadSettings</i>	<code>timed_events_thread()</code>	No
<i>ThreadSettings</i>	<code>discovery_server_thread()</code>	No
<i>ThreadSettings</i>	<code>typelookup_service_thread()</code>	No
<i>ThreadSettings</i>	<code>security_log_thread()</code>	No

Important: The only mutable field in *WireProtocolConfigQos* is `m_DiscoveryServers`, which is contained in `discovery_config` within `builtin` (see *Modifying remote servers list at run time*).

Important: Upon the call to `create_participant()`, if Fast DDS is compiled with statistics support (enabled by default, see *CMake options*), the internal `DomainParticipantQos` may differ from the input `DomainParticipantQos` (see *Statistics Module Settings*). This entails that applications willing to further modify the `DomainParticipantQos` after `DomainParticipant` creation should:

1. Retrieve the internal `DomainParticipantQos` by the means of `DomainParticipant::get_qos()`.
 2. Perform the desired modifications.
 3. Update the `DomainParticipantQos` by the means of `DomainParticipant::set_qos()`.
-

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

The QoS value of a previously created `DomainParticipant` can be modified using the `DomainParticipant::set_qos()` member function. Trying to modify an immutable `QosPolicy` on an already enabled `DomainParticipant` will result on an error. In such case, no changes will be applied and the `DomainParticipant` will keep its previous `DomainParticipantQos`.

```
// Create a DomainParticipant with default DomainParticipantQos
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↳DEFAULT);
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) !=
    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) !=
    RETCODE_OK)
{
    // Error
    return;
}

// Create a Topic with the new default TopicQos.
DomainParticipant* participant_with_qos_type2 =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DomainParticipantQos to the original default constructed values
if (DomainParticipantFactory::get_instance()->set_default_participant_qos(PARTICIPANT_
↪QOS_DEFAULT)
    != RETCODE_OK)

```

(continues on next page)

(continued from previous page)

```

{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (DomainParticipantFactory::get_instance()->set_default_participant_
    ↪ qos(DomainParticipantQos())
    != 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) != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (participant->set_qos(DomainParticipantFactory::get_instance()->get_default_
    ↪ participant_qos())
    != RETCODE_OK)
{
    // Error
    return;
}

```

Note: The value PARTICIPANT_QOS_DEFAULT has different meaning depending on where it is used:

- On `create_participant()` and `DomainParticipant::set_qos()` it refers to the default `DomainParticipantQos` as returned by `get_default_participant_qos()`.
- On `set_default_participant_qos()` it refers to the default constructed `DomainParticipantQos()`.

DomainParticipantListener

`DomainParticipantListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *DomainParticipant*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`DomainParticipantListener` inherits from *TopicListener*, *PublisherListener*, and *SubscriberListener*. Therefore, it has the ability to react to every kind of event that is reported to any of its attached Entities. Since events are always notified to the most specific Entity Listener that can handle the event, callbacks that `DomainParticipantListener` inherits from other Listeners will only be called if no other Entity was able to handle the event, either because it has no Listener attached, or because the callback is disabled by the `StatusMask` on the Entity.

Additionally, `DomainParticipantListener` adds the following non-standard callbacks:

- `on_participant_discovery()`: A new `DomainParticipant` is discovered in the same domain, a previously known `DomainParticipant` has been removed, or some `DomainParticipant` has changed its QoS. This method provides an overload with an additional boolean output parameter so a discovery callback can tell the middleware if a newly discovered participant has to be ignored via the use of the `ignore_participant()`. This overload should be used when there is a need to ignore participants inside the discovery callback, since calling `ignore_participant()` inside the listener might deadlock. If both callbacks are implemented, the discovery callback with the `should_be_ignored` boolean flag takes precedence. The second discovery callback is only executed if the discovered `DomainParticipant` is not ignored in the first callback (`should_be_ignored` parameter returns false).
- `on_data_reader_discovery()`: A new *DataReader* is discovered in the same domain, a previously known `DataReader` has been removed, or some `DataReader` has changed its QoS.
- `on_data_writer_discovery()`: A new *DataWriter* is discovered in the same domain, a previously known `DataWriter` has been removed, or some `DataWriter` has changed its QoS.
- `onParticipantAuthentication()`: Informs about the result of the authentication process of a remote `DomainParticipant` (either on failure or success).

Important: For more information about callbacks and its hierarchy, please refer to *Listener*.

```
class CustomDomainParticipantListener : public DomainParticipantListener
{
public:
    CustomDomainParticipantListener()
        : DomainParticipantListener()
    {
    }

    virtual ~CustomDomainParticipantListener()
    {
    }
}
```

(continues on next page)

(continued from previous page)

```

    void on_participant_discovery(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::ParticipantDiscoveryInfo&& info,
        bool& should_be_ignored) override
    {
        should_be_ignored = false;
        if (info.status ==
↪ eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOVERED_PARTICIPANT)
        {
            std::cout << "New participant discovered" << std::endl;
            // The following line can be modified to evaluate whether the discovered
↪ participant should be ignored
            // (usually based on fields present in the discovery information)
            bool ignoring_condition = false;
            if (ignoring_condition)
            {
                should_be_ignored = true; // Request the ignoring of the discovered
↪ participant
            }
        }
        else if (info.status ==
↪ eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::REMOVED_PARTICIPANT ||
            info.status ==
↪ eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DROPPED_PARTICIPANT)
        {
            std::cout << "Participant lost" << std::endl;
        }
    }

#if HAVE_SECURITY
    void onParticipantAuthentication(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::ParticipantAuthenticationInfo&& info) override
    {
        if (info.status ==
↪ eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::AUTHORIZED_PARTICIPANT)
        {
            std::cout << "A participant was authorized" << std::endl;
        }
        else if (info.status ==
↪ eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::UNAUTHORIZED_PARTICIPANT)
        {
            std::cout << "A participant failed authorization" << std::endl;
        }
    }

#endif // if HAVE_SECURITY

    void on_data_reader_discovery(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info,

```

(continues on next page)

(continued from previous page)

```

        bool& should_be_ignored) override
    {
        should_be_ignored = false;
        if (info.status == eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERED_
↪READER)
        {
            std::cout << "New datareader discovered" << std::endl;
            // The following line can be modified to evaluate whether the discovered_
↪datareader should be ignored
            // (usually based on fields present in the discovery information)
            bool ignoring_condition = false;
            if (ignoring_condition)
            {
                should_be_ignored = true; // Request the ignoring of the discovered_
↪datareader
            }
        }
        else if (info.status == eprosima::fastrtps::rtps::ReaderDiscoveryInfo::REMOVED_
↪READER)
        {
            std::cout << "Datareader lost" << std::endl;
        }
    }

    void on_data_writer_discovery(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info,
        bool& should_be_ignored) override
    {
        should_be_ignored = false;
        if (info.status == eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERED_
↪WRITER)
        {
            std::cout << "New datawriter discovered" << std::endl;
            // The following line can be modified to evaluate whether the discovered_
↪datawriter should be ignored
            // (usually based on fields present in the discovery information)
            bool ignoring_condition = false;
            if (ignoring_condition)
            {
                should_be_ignored = true; // Request the ignoring of the discovered_
↪datawriter
            }
        }
        else if (info.status == eprosima::fastrtps::rtps::WriterDiscoveryInfo::REMOVED_
↪WRITER)
        {
            std::cout << "Datawriter lost" << 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
<i>EntityFactoryQosPolicy</i>	<code>entity_factory()</code>	Yes
<i>ThreadSettings</i>	<code>shm_watchdog_thread()</code>	No
<i>ThreadSettings</i>	<code>file_watch_threads()</code>	No

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) != 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) != RETCODE_OK)
{

```

(continues on next page)

(continued from previous page)

```

    // Error
    return;
}

// Create a DomainParticipant with the new DomainParticipantFactoryQos.
// The returned DomainParticipant is disabled and will need to be enabled explicitly
DomainParticipant* disabled_participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == disabled_participant)
{
    // Error
    return;
}

```

Loading profiles from an XML file

To create Entities based on XML profiles, the file containing such profiles must be loaded first.

If the profile is described in one of the default loaded files, it will be automatically available on initialization. Otherwise, `load_XML_profiles_file()` member function can be used to load the profiles in the XML. See section [XML profiles](#) for more information regarding XML profile format and automatic loading.

Once loaded, the name of the profiles can be used to create Entities that will have QoS settings according to the profile specifications.

```

// Load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Profiles can now be used to create Entities
DomainParticipant* participant_with_profile =
    DomainParticipantFactory::get_instance()->create_participant_with_profile(0,
    ↪"participant_profile");
if (nullptr == participant_with_profile)
{
    // Error
    return;
}

```

Creating a DomainParticipant

Creation of a *DomainParticipant* is done with the `create_participant()` member function on the *DomainParticipantFactory* singleton, that acts as a factory for the *DomainParticipant*. Then, when the lifecycle of the participant finishes, every participant must be deleted with `delete_participant()`. Please refer to [Deleting a DomainParticipant](#) for further details on the deletion of a participant.

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;
}
```

Default profile DomainParticipant creation

If there is a profile already exported in the environment (please refer to *XML profiles* for related information), creating a DomainParticipant with the `create_participant_with_default_profile()` member function on the *DomainParticipantFactory* singleton would use that settings to configure the participant. If the profile has not been exported, the DomainParticipant will be created with the default values per *DomainParticipantQos*, and 0 as DomainId.

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_default_profile()` will return a null pointer if there was an error during the operation. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *XML profiles*.

```
// Create a DomainParticipant using the environment profile and no Listener
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant_with_default_
    ↪profile();
if (nullptr == participant)
{
    // Error
    return;
}

// Create a DomainParticipant using the environment profile and a custom Listener.
// CustomDomainParticipantListener inherits from DomainParticipantListener.
CustomDomainParticipantListener custom_listener;
DomainParticipant* participant_with_custom_listener =
    DomainParticipantFactory::get_instance()->create_participant_with_default_
    ↪profile(
        &custom_listener, StatusMask::none());
if (nullptr == participant_with_custom_listener)
{
    // Error
    return;
}
```

Deleting a DomainParticipant

A DomainParticipant shall be deleted with the `delete_participant()` member function on the *DomainParticipantFactory* singleton.

Important: A DomainParticipant can only be deleted if all Entities belonging to the participant (Publisher, Subscriber or Topic) have already been deleted. Otherwise, the function will issue an error and the DomainParticipant will not be deleted. This can be performed either by using the `delete_contained_entities()` member function of the *DomainParticipant* or manually deleting each entity with the corresponding `delete_` method from the DomainParticipant

e.g `delete_publisher()`, `delete_subscriber()`, `delete_topic()`, etc.

```
// Create a DomainParticipant
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

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

// Delete entities created by the DomainParticipant
if (participant->delete_contained_entities() != RETCODE_OK)
{
    // DomainParticipant failed to delete the entities it created.
    return;
}

// Delete the DomainParticipant
if (DomainParticipantFactory::get_instance()->delete_participant(participant) != 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">
  <data_writer profile_name="pub_11">
    <qos>
      <partition>
        <names>
          <name>Partition_1</name>
          <name>Partition_2</name>
        </names>
      </partition>
    </qos>
  </data_writer>

  <data_writer profile_name="pub_12">
    <qos>
      <partition>
        <names>
          <name>*</name>
        </names>
      </partition>
    </qos>
  </data_writer>

  <data_writer profile_name="pub_22">
    <qos>
      <partition>
        <names>
          <name>Partition*</name>
        </names>
      </partition>
    </qos>
  </data_writer>

```


7.18.3 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
<i>PresentationQosPolicy</i>	<code>presentation()</code>	Yes
<i>PartitionQosPolicy</i>	<code>partition()</code>	Yes
<i>GroupDataQosPolicy</i>	<code>group_data()</code>	Yes
<i>EntityFactoryQosPolicy</i>	<code>entity_factory()</code>	Yes

Refer to the detailed description of each *QosPolicy* class for more information about their usage and default values.

The QoS value of a previously created *Publisher* can be modified using the `Publisher::set_qos()` member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher with default PublisherQos
Publisher* publisher =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
```

(continues on next page)

(continued from previous page)

```

PublisherQos qos = publisher->get_qos();

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

// Assign the new Qos to the object
publisher->set_qos(qos);

```

Default PublisherQos

The default *PublisherQos* refers to the value returned by the `get_default_publisher_qos()` member function on the *DomainParticipant* instance. The special value `PUBLISHER_QOS_DEFAULT` can be used as QoS argument on `create_publisher()` or `Publisher::set_qos()` member functions to indicate that the current default *PublisherQos* should be used.

When the system starts, the default *PublisherQos* is equivalent to the default constructed value `PublisherQos()`. The default *PublisherQos* can be modified at any time using the `set_default_publisher_qos()` member function on the *DomainParticipant* instance. Modifying the default *PublisherQos* will not affect already existing *Publisher* instances.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
PublisherQos qos_type1 = participant->get_default_publisher_qos();

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

// Set as the new default PublisherQos
if (participant->set_default_publisher_qos(qos_type1) != RETCODE_OK)
{
    // Error
    return;
}

// Create a Publisher with the new default PublisherQos.
Publisher* publisher_with_qos_type1 =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher_with_qos_type1)
{
    // Error
    return;
}

```

(continues on next page)

(continued from previous page)

```

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

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

// Set as the new default PublisherQos
if (participant->set_default_publisher_qos(qos_type2) != 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)
    != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (participant->set_default_publisher_qos(PublisherQos())
    != RETCODE_OK)
{
    // Error
    return;
}

```

set_default_publisher_qos() member function also accepts the special value PUBLISHER_QOS_DEFAULT as input argument. This will reset the current default PublisherQos to default constructed value PublisherQos().

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a custom PublisherQos

```

(continues on next page)

(continued from previous page)

```

PublisherQos custom_qos;

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

// Create a publisher with a custom PublisherQos
Publisher* publisher = participant->create_publisher(custom_qos);
if (nullptr == publisher)
{
    // Error
    return;
}

// Set the QoS on the publisher to the default
if (publisher->set_qos(PUBLISHER_QOS_DEFAULT) != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (publisher->set_qos(participant->get_default_publisher_qos())
    != RETCODE_OK)
{
    // Error
    return;
}

```

Note: The value PUBLISHER_QOS_DEFAULT has different meaning depending on where it is used:

- On `create_publisher()` and `Publisher::set_qos()` it refers to the default *PublisherQos*, as returned by `get_default_publisher_qos()`.
- On `set_default_publisher_qos()` it refers to the default constructed `PublisherQos()`.

PublisherListener

`PublisherListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *Publisher*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`PublisherListener` inherits from *DataWriterListener*. Therefore, it has the ability to react to all events that are reported to the *DataWriter*. Since events are always notified to the most specific Entity Listener that can handle the event, callbacks that `PublisherListener` inherits from `DataWriterListener` will only be called if the triggering `DataWriter` has no Listener attached, or if the callback is disabled by the `StatusMask` on the `DataWriter`.

`PublisherListener` does not add any new callback. Please, refer to the *DataWriterListener* for the list of inherited callbacks and override examples.

Creating a Publisher

A *Publisher* always belongs to a *DomainParticipant*. Creation of a Publisher is done with the `create_publisher()` member function on the *DomainParticipant* instance, that acts as a factory for the Publisher.

Mandatory arguments are:

- The *PublisherQos* describing the behavior of the Publisher. If the provided value is `PUBLISHER_QOS_DEFAULT`, the value of the *Default PublisherQos* is used.

Optional arguments are:

- A Listener derived from *PublisherListener*, implementing the callbacks that will be triggered in response to events and state changes on the Publisher. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the *PublisherListener*. By default all events are enabled.

`create_publisher()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ➔DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher with default PublisherQos and no Listener
// The value PUBLISHER_QOS_DEFAULT is used to denote the default QoS.
Publisher* publisher_with_default_qos =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher_with_default_qos)
{
    // Error
    return;
}

// A custom PublisherQos can be provided to the creation method
PublisherQos custom_qos;

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

Publisher* publisher_with_custom_qos =
    participant->create_publisher(custom_qos);
if (nullptr == publisher_with_custom_qos)
{
    // Error
    return;
}

// Create a Publisher with default QoS and a custom Listener.
```

(continues on next page)

(continued from previous page)

```
// CustomPublisherListener inherits from PublisherListener.
// The value PUBLISHER_QOS_DEFAULT is used to denote the default QoS.
CustomPublisherListener custom_listener;
Publisher* publisher_with_default_qos_and_custom_listener =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT, &custom_listener);
if (nullptr == publisher_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a Publisher

Instead of using a *PublisherQos*, the name of a profile can be used to create a Publisher with the `create_publisher_with_profile()` member function on the `DomainParticipant` instance.

Mandatory arguments are:

- A string with the name that identifies the Publisher.

Optional arguments are:

- A Listener derived from *PublisherListener*, implementing the callbacks that will be triggered in response to events and state changes on the Publisher. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the `PublisherListener`. By default all events are enabled.

`create_publisher_with_profile()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher using a profile and no Listener
Publisher* publisher_with_profile =
    participant->create_publisher_with_profile("publisher_profile");
if (nullptr == publisher_with_profile)
{
    // Error
}
```

(continues on next page)

(continued from previous page)

```

    return;
}

// Create a Publisher using a profile and a custom Listener.
// CustomPublisherListener inherits from PublisherListener.
CustomPublisherListener custom_listener;
Publisher* publisher_with_profile_and_custom_listener =
    participant->create_publisher_with_profile("publisher_profile", &custom_
↪listener);
if (nullptr == publisher_with_profile_and_custom_listener)
{
    // Error
    return;
}

```

Deleting a Publisher

A Publisher can be deleted with the `delete_publisher()` member function on the `DomainParticipant` instance where the Publisher was created.

Note: A Publisher can only be deleted if all Entities belonging to the Publisher (DataWriters) have already been deleted. Otherwise, the function will issue an error and the Publisher will not be deleted. This can be performed by using the `delete_contained_entities()` member function of the *Publisher*.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Publisher
Publisher* publisher =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher)
{
    // Error
    return;
}

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

// Delete the entities the Publisher created.
if (publisher->delete_contained_entities() != RETCODE_OK)
{

```

(continues on next page)

(continued from previous page)

```

    // Publisher failed to delete the entities it created.
    return;
}

// Delete the Publisher
if (participant->delete_publisher(publisher) != RETCODE_OK)
{
    // Error
    return;
}

```

DataWriter

A `DataSource` is attached to exactly one *Publisher* that acts as a factory for it. Additionally, each `DataSource` is bound to a single *Topic* since its creation. This *Topic* must exist prior to the creation of the `DataSource`, and must be bound to the data type that the `DataSource` wants to publish.

The effect of creating a new `DataSource` 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 `DataSource` is created, the application can inform of changes in the data value using the `write()` member function on the `DataSource`. These changes will be transmitted to all subscriptions matched with this publication.

DataSourceQos

`DataSourceQos` controls the behavior of the `DataSource`. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
<i>DurabilityQosPolicy</i>	<code>durability()</code>	No
<i>DurabilityServiceQosPolicy</i>	<code>durability_service()</code>	No
<i>DeadlineQosPolicy</i>	<code>deadline()</code>	Yes
<i>LatencyBudgetQosPolicy</i>	<code>latency_budget()</code>	Yes
<i>LivelinessQosPolicy</i>	<code>liveliness()</code>	No
<i>ReliabilityQosPolicy</i>	<code>reliability()</code>	No (*)
<i>DestinationOrderQosPolicy</i>	<code>destination_order()</code>	No
<i>HistoryQosPolicy</i>	<code>history()</code>	No
<i>ResourceLimitsQosPolicy</i>	<code>resource_limits()</code>	No
<i>TransportPriorityQosPolicy</i>	<code>transport_priority()</code>	Yes
<i>LifespanQosPolicy</i>	<code>lifespan()</code>	Yes
<i>UserDataQosPolicy</i>	<code>user_data()</code>	Yes
<i>OwnershipQosPolicy</i>	<code>ownership()</code>	No
<i>OwnershipStrengthQosPolicy</i>	<code>ownership_strength()</code>	Yes
<i>WriterDataLifecycleQosPolicy</i>	<code>writer_data_lifecycle()</code>	Yes
<i>PublishModeQosPolicy</i>	<code>publish_mode()</code>	No
<i>DataRepresentationQosPolicy</i>	<code>representation()</code>	No
<i>PropertyPolicyQos</i>	<code>properties()</code>	No
<i>RTPSReliableWriterQos</i>	<code>reliable_writer_qos()</code>	Yes (*)
<i>RTPSEndpointQos</i>	<code>endpoint()</code>	No
<i>WriterResourceLimitsQos</i>	<code>writer_resource_limits()</code>	No
<i>DataSharingQosPolicy</i>	<code>data_sharing()</code>	No

The following non-consolidated property-assigned QoS apply to DataWriters:

Property name	Non-consolidated QoS
fastdds.push_mode	<i>DataWriter operating mode QoS Policy</i>
partitions	<i>Endpoint Partitions</i>

Refer to the detailed description of each `QoSPolicy` class for more information about their usage and default values.

Note: Reliability kind (whether the publication is reliable or best effort) is not mutable. However, the `max_blocking_time` data member of *ReliabilityQoSPolicy* can be modified any time.

Note: Not all data members of `RTPSReliableWriterQos` are mutable, please refer to *RTPSReliableWriterQos* for more information.

The QoS value of a previously created `DataWriter` can be modified using the `DataWriter::set_qos()` member function.

```
// Create a DataWriter with default DataWriterQos
DataWriter* data_writer =
    publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
DataWriterQos qos = data_writer->get_qos();

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

// Assign the new Qos to the object
data_writer->set_qos(qos);
```

Default DataWriterQos

The default *DataWriterQos* refers to the value returned by the `get_default_datawriter_qos()` member function on the `Publisher` instance. The special value `DATAWRITER_QOS_DEFAULT` can be used as QoS argument on `create_datawriter()` or `DataWriter::set_qos()` member functions to indicate that the current default `DataWriterQos` should be used.

When the system starts, the default `DataWriterQos` is equivalent to the default constructed value `DataWriterQos()`. The default `DataWriterQos` can be modified at any time using the `set_default_datawriter_qos()` member function on the `Publisher` instance. Modifying the default `DataWriterQos` will not affect already existing `DataWriter` instances.

```
// Get the current QoS or create a new one from scratch
DataWriterQos qos_type1 = publisher->get_default_datawriter_qos();
```

(continues on next page)

(continued from previous page)

```

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

// Set as the new default DataWriterQos
if (publisher->set_default_datawriter_qos(qos_type1) != 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) != RETCODE_OK)
{
    // Error
    return;
}

// Create a DataWriter with the new default DataWriterQos.
DataWriter* data_writer_with_qos_type2 =
    publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DataWriterQos to the original default constructed values
if (publisher->set_default_datawriter_qos(DATAWRITER_QOS_DEFAULT)
    != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (publisher->set_default_datawriter_qos(DataWriterQos())

```

(continues on next page)

(continued from previous page)

```

        != 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) != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (data_writer->set_qos(publisher->get_default_datawriter_qos())
    != RETCODE_OK)
{
    // Error
    return;
}

```

Note: The value DATAWRITER_QOS_DEFAULT has different meaning depending on where it is used:

- On create_datawriter() and DataWriter::set_qos() it refers to the default DataWriterQos as returned by get_default_datawriter_qos().
- On set_default_datawriter_qos() it refers to the default constructed DataWriterQos().

DataWriterListener

`DataWriterListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *DataWriter*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`DataWriterListener` defines the following callbacks:

- `on_publication_matched()`: The `DataWriter` has found a *DataReader* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a `DataReader` that was previously considered to be matched.
- `on_offered_deadline_missed()`: The `DataWriter` failed to provide data within the deadline period configured on its *DataWriterQos*. It will be called for each deadline period and data instance for which the `DataWriter` failed to provide data.
- `on_offered_incompatible_qos()`: The `DataWriter` has found a `DataReader` that matches the *Topic* and has a common partition, but with a requested QoS that is incompatible with the one defined on the `DataWriter`.
- `on_liveliness_lost()`: The `DataWriter` did not respect the liveliness configuration on its *DataWriterQos*, and therefore, `DataReader` entities will consider the `DataWriter` as no longer *active*.
- `on_unacknowledged_sample_removed()`: The `DataWriter` has removed a sample that has not been acknowledged by every matched `DataReader`.

on_unacknowledged_sample_removed callback

`on_unacknowledged_sample_removed()` non-standard callback notifies the user when a sample has been removed without being sent/received by the matched `DataReaders`. This could happen in constrained networks or if the publication throughput is too demanding. This callback can be used to detect these situations so the publishing application can apply some solution to ease this issue like reducing the publication rate.

The criteria to consider that a sample has been removed without being acknowledged depends on the *ReliabilityQosPolicy*:

- **BEST_EFFORT_RELIABILITY_QOS** `DataWriters` will notify that a sample has been removed while unacknowledged if the sample has not been sent through the transport.
- **RELIABLE_RELIABILITY_QOS** `DataWriters` consider samples to have been removed unacknowledged if not every matched `DataReader` has confirmed its reception by sending the corresponding meta-traffic **ACK** message. Consequently, a sample that is notified as removed unacknowledged might be received by one or more `DataReaders`, but not by every matched one, or at least, the **ACK** message has not been received at the moment of sample removal. A race condition is inevitable in this case, because when the sample is removed, the **ACK** from some matched `DataReader` is missing, but that means that it might have been lost in the transmission or that the message is still coming through and it will be received after the sample removal. Thus, this criteria may include false positives, but from the user's point of view, it is more meaningful to know when the sample has not been acknowledged by every matched `DataReader` even if some samples are erroneously notified.

A specific case must be considered for reliable `DataWriters` with *DisablePositiveACKsQosPolicy* enabled. This policy disables the sending of positive **ACK** messages, unless the sample has been lost in which case the matched `DataReader` notifies the loss with a negative **NACK** message. If no **NACK** has been received in the time defined in the QoS policy, the sample is considered to be received. Again, this is prone to race conditions because the **NACK** message might be on its way or have been lost in the network. For this specific case, where **ACK** messages are not going to be received, the reliable `DataWriter` uses the same criteria as the best effort `DataWriter`.

```

class CustomDataWriterListener : public DataWriterListener
{
public:
    CustomDataWriterListener()
        : DataWriterListener()
    {
    }

    virtual ~CustomDataWriterListener()
    {
    }

    void on_publication_matched(
        DataWriter* writer,
        const PublicationMatchedExceptionInfo& info) override
    {
        static_cast<void>(writer);
        if (info.current_count_change == 1)
        {
            std::cout << "Matched a remote Subscriber for one of our Topics" <<
↳std::endl;
        }
        else if (info.current_count_change == -1)
        {
            std::cout << "Unmatched a remote Subscriber" << std::endl;
        }
    }

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

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

    void on_liveliness_lost(
        DataWriter* writer,
        const LivelinessLostStatus& status) override
    {
        static_cast<void>(writer);

```

(continues on next page)

(continued from previous page)

```

        static_cast<void>(status);
        std::cout << "Liveliness lost. Matched Subscribers will consider us offline" <<
        ↪std::endl;
    }

    void on_unacknowledged_sample_removed(
        DataWriter* writer,
        const InstanceHandle_t& instance) override
    {
        static_cast<void>(writer);
        static_cast<void>(instance);
        std::cout << "Sample removed unacknowledged" << std::endl;
    }
};

```

Creating a DataWriter

A *DataWriter* always belongs to a *Publisher*. Creation of a DataWriter is done with the `create_datawriter()` member function on the Publisher instance, that acts as a factory for the DataWriter.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- The *DataWriterQos* describing the behavior of the DataWriter. If the provided value is `DATAWRITER_QOS_DEFAULT`, the value of the *Default DataWriterQos* is used. If the provided value is `DATAWRITER_QOS_USE_TOPIC_QOS`, the values of the default QoS and the provided TopicQoS are used, whereby any policy that is set on the TopicQoS overrides the corresponding policy on the default QoS.

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;

```

(continues on next page)

(continued from previous page)

```

// 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;
}

// Create a DataWriter with default QoS and a custom TopicQoS.
// The value DATAWRITER_QOS_USE_TOPIC_QOS is used to denote the default QoS
// and to override the TopicQoS.
Topic* topic;
DataWriter* data_writer_with_default_qos_and_custom_topic_qos =
    publisher->create_datawriter(topic, DATAWRITER_QOS_USE_TOPIC_QOS);
if (nullptr == data_writer_with_default_qos_and_custom_topic_qos)
{
    // 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;
}
```

Creating a DataWriter with a custom PayloadPool

A custom *PayloadPool* can be passed as an argument during the creation of a *DataWriter*. This allows for customizing the management of the information exchanged between DataWriters and DataReaders. The same configuration can be set in the *opposite endpoint*.

```
// A DataWriterQos must be provided to the creation method
DataWriterQos qos;

// Create PayloadPool
std::shared_ptr<eprosima::fastrtps::rtps::IPayloadPool> payload_pool =
    std::dynamic_pointer_cast<eprosima::fastrtps::rtps::IPayloadPool>(std::make_
    ↪ shared<CustomPayloadPool>());

DataWriter* data_writer = publisher->create_datawriter(topic, qos, nullptr,
    ↪ StatusMask::all(), payload_pool);
if (nullptr == data_writer)
{
    // Error
    return;
}
```


This configuration can be performed also in the *RTPS layer*. The *customization example* applies both layers.

Deleting a DataWriter

A DataWriter can be deleted with the `delete_datawriter()` member function on the *Publisher* instance where the DataWriter was created.

```
// Create a DataWriter
DataWriter* data_writer =
    publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == data_writer)
{
    // Error
    return;
}

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

// Delete the DataWriter
if (publisher->delete_datawriter(data_writer) != RETCODE_OK)
{
    // Error
    return;
}
```

Publishing data

The user informs of a change in the value of a data instance with the `write()` member function on the *DataWriter*. This change will then be communicated to every *DataReader* matched with the DataWriter. As a side effect, this operation asserts liveness 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);
```

(continues on next page)

(continued from previous page)

```

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()) != RETCODE_OK)
{
    // Error
    return;
}

// The data instance can be reused to publish new values,
// but delete it at the end to avoid leaks
custom_type_support->deleteData(data);

```

Blocking of the write operation

If the reliability kind is set to `RELIABLE` on the *DataWriterQos*, the `write()` operation may block. Specifically, if the limits specified in the configured resource limits have been reached, the `write()` operation will block waiting for space to become available. Under these circumstances, the reliability `max_blocking_time` configures the maximum time the write operation may block waiting. If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits, the write operation will fail and return `TIMEOUT`.

Borrowing a data buffer

When the user calls `write()` with a new sample value, the data is copied from the given sample to the `DataWriter`'s memory. For large data types this copy can consume significant time and memory resources. Instead, the `DataWriter` can loan a sample from its memory to the user, and the user can fill this sample with the required values. When `write()` is called with such a loaned sample, the `DataWriter` does not copy its contents, as it already owns the buffer.

To use loaned data samples in publications, perform the following steps:

1. Get a reference to a loaned sample using `loan_sample()`.
2. Use the reference to build the data sample.

3. Write the sample using `write()`.

Once `write()` has been called with a loaned sample, the loan is considered returned, and it is not safe to make any changes on the contents of the sample.

If function `loan_sample()` is called but the sample is never written, the loan must be returned to the `DataWriter` using `discard_loan()`. Otherwise the `DataWriter` may run out of samples.

```
// Borrow a data instance
void* data = nullptr;
if (RETCODE_OK == data_writer->loan_sample(data))
{
    bool error = false;

    // Fill the data values
    // (...)

    if (error)
    {
        // Return the loan without publishing
        data_writer->discard_loan(data);
        return;
    }

    // Publish the new value
    if (data_writer->write(data, eprosima::fastrtps::rtps::InstanceHandle_t()) !=
    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);
```

7.18.4 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
<i>PresentationQosPolicy</i>	<code>presentation()</code>	Yes
<i>PartitionQosPolicy</i>	<code>partition()</code>	Yes
<i>GroupDataQosPolicy</i>	<code>group_data()</code>	Yes
<i>EntityFactoryQosPolicy</i>	<code>entity_factory()</code>	Yes

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

The QoS value of a previously created `Subscriber` can be modified using the `Subscriber::set_qos()` member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Subscriber with default SubscriberQos
Subscriber* subscriber =
    participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
SubscriberQos qos = subscriber->get_qos();

// Modify QoS attributes
qos.entity_factory().autoenable_created_entities = false;

// Assign the new QoS to the object
subscriber->set_qos(qos);
```

Default SubscriberQos

The default *SubscriberQos* refers to the value returned by the `get_default_subscriber_qos()` member function on the *DomainParticipant* instance. The special value `SUBSCRIBER_QOS_DEFAULT` can be used as QoS argument on `create_subscriber()` or `Subscriber::set_qos()` member functions to indicate that the current default *SubscriberQos* should be used.

When the system starts, the default *SubscriberQos* is equivalent to the default constructed value `SubscriberQos()`. The default *SubscriberQos* can be modified at any time using the `set_default_subscriber_qos()` member function on the *DomainParticipant* instance. Modifying the default *SubscriberQos* will not affect already existing *Subscriber* instances.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ←DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
SubscriberQos qos_type1 = participant->get_default_subscriber_qos();

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

// Set as the new default SubscriberQos
if (participant->set_default_subscriber_qos(qos_type1) != 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) != RETCODE_OK)
{
    // Error
```

(continues on next page)

(continued from previous page)

```

    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)
    != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (participant->set_default_subscriber_qos(SubscriberQos())
    != RETCODE_OK)
{
    // Error
    return;
}

```

set_default_subscriber_qos() member function also accepts the special value SUBSCRIBER_QOS_DEFAULT as input argument. This will reset the current default SubscriberQos to default constructed value SubscriberQos().

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a custom SubscriberQos
SubscriberQos custom_qos;

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

// Create a subscriber with a custom SubscriberQos
Subscriber* subscriber = participant->create_subscriber(custom_qos);
if (nullptr == subscriber)
{
    // Error

```

(continues on next page)

(continued from previous page)

```

    return;
}

// Set the QoS on the subscriber to the default
if (subscriber->set_qos(SUBSCRIBER_QOS_DEFAULT) != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (subscriber->set_qos(participant->get_default_subscriber_qos())
    != RETCODE_OK)
{
    // Error
    return;
}

```

Note: The value SUBSCRIBER_QOS_DEFAULT has different meaning depending on where it is used:

- On `create_subscriber()` and `Subscriber::set_qos()` it refers to the default `SubscriberQos` as returned by `get_default_subscriber_qos()`.
- On `set_default_subscriber_qos()` it refers to the default constructed `SubscriberQos()`.

SubscriberListener

`SubscriberListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *Subscriber*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`SubscriberListener` inherits from *DataReaderListener*. Therefore, it has the ability to react to all events that are reported to the *DataReader*. Since events are always notified to the most specific Entity Listener that can handle the event, callbacks that `SubscriberListener` inherits from `DataReaderListener` will only be called if the triggering `DataReader` has no Listener attached, or if the callback is disabled by the `StatusMask` on the `DataReader`.

Additionally, `SubscriberListener` adds the following callback:

- `on_data_on_readers()`: New data is available on any `DataReader` belonging to this `Subscriber`. There is no queuing of invocations to this callback, meaning that if several new data changes are received at once, only one callback invocation may be issued for all of them, instead of one per change. If the application is retrieving the received data on this callback, it must keep *reading data* until no new changes are left.

Important: For more information about callbacks and its hierarchy, please refer to *Listener*.

```

class CustomSubscriberListener : public SubscriberListener
{
public:

```

(continues on next page)

(continued from previous page)

```

CustomSubscriberListener()
    : SubscriberListener()
{
}

virtual ~CustomSubscriberListener()
{
}

virtual void on_data_on_readers(
    Subscriber* sub)
{
    static_cast<void>(sub);
    std::cout << "New data available" << std::endl;
}
};

```

Creating a Subscriber

A *Subscriber* always belongs to a *DomainParticipant*. Creation of a Subscriber is done with the `create_subscriber()` member function on the *DomainParticipant* instance, that acts as a factory for the Subscriber.

Mandatory arguments are:

- The *SubscriberQos* describing the behavior of the Subscriber. If the provided value is `SUBSCRIBER_QOS_DEFAULT`, the value of the *Default SubscriberQos* is used.

Optional arguments are:

- A Listener derived from *SubscriberListener*, implementing the callbacks that will be triggered in response to events and state changes on the Subscriber. By default empty callbacks are used.
- A *StatusMask* that activates or deactivates triggering of individual callbacks on the *SubscriberListener*. By default all events are enabled.

`create_subscriber()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↳DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Subscriber with default SubscriberQos and no Listener
// The value SUBSCRIBER_QOS_DEFAULT is used to denote the default QoS.
Subscriber* subscriber_with_default_qos =
    participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);

```

(continues on next page)

(continued from previous page)

```

if (nullptr == subscriber_with_default_qos)
{
    // Error
    return;
}

// A custom SubscriberQos can be provided to the creation method
SubscriberQos custom_qos;

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

Subscriber* subscriber_with_custom_qos =
    participant->create_subscriber(custom_qos);
if (nullptr == subscriber_with_custom_qos)
{
    // Error
    return;
}

// Create a Subscriber with default QoS and a custom Listener.
// CustomSubscriberListener inherits from SubscriberListener.
// The value SUBSCRIBER_QOS_DEFAULT is used to denote the default QoS.
CustomSubscriberListener custom_listener;
Subscriber* subscriber_with_default_qos_and_custom_listener =
    participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT, &custom_listener);
if (nullptr == subscriber_with_default_qos_and_custom_listener)
{
    // Error
    return;
}

```

Profile based creation of a Subscriber

Instead of using a `SubscriberQos`, the name of a profile can be used to create a `Subscriber` with the `create_subscriber_with_profile()` member function on the `DomainParticipant` instance.

Mandatory arguments are:

- A string with the name that identifies the `Subscriber`.

Optional arguments are:

- A `Listener` derived from `SubscriberListener`, implementing the callbacks that will be triggered in response to events and state changes on the `Subscriber`. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the `SubscriberListener`. By default all events are enabled.

`create_subscriber_with_profile()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See [Loading profiles from an XML file](#).

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Subscriber using a profile and no Listener
Subscriber* subscriber_with_profile =
    participant->create_subscriber_with_profile("subscriber_profile");
if (nullptr == subscriber_with_profile)
{
    // Error
    return;
}

// Create a Subscriber using a profile and a custom Listener.
// CustomSubscriberListener inherits from SubscriberListener.
CustomSubscriberListener custom_listener;
Subscriber* subscriber_with_profile_and_custom_listener =
    participant->create_subscriber_with_profile("subscriber_profile", &custom_
    ↪listener);
if (nullptr == subscriber_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a Subscriber

A Subscriber can be deleted with the `delete_subscriber()` member function on the `DomainParticipant` instance where the Subscriber was created.

Note: A Subscriber can only be deleted if all Entities belonging to the Subscriber (DataReaders) have already been deleted. Otherwise, the function will issue an error and the Subscriber will not be deleted. This can be performed by using the `delete_contained_entities()` member function of the *Subscriber*.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
```

(continues on next page)

(continued from previous page)

```

    return;
}

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

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

// Delete the entities the subscriber created
if (subscriber->delete_contained_entities() != RETCODE_OK)
{
    // Subscriber failed to delete the entities it created
    return;
}

// Delete the Subscriber
if (participant->delete_subscriber(subscriber) != 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
<i>DurabilityQosPolicy</i>	<code>durability()</code>	No
<i>DurabilityServiceQosPolicy</i>	<code>durability_service()</code>	Yes
<i>DeadlineQosPolicy</i>	<code>deadline()</code>	Yes
<i>LatencyBudgetQosPolicy</i>	<code>latency_budget()</code>	Yes
<i>LivelinessQosPolicy</i>	<code>liveliness()</code>	No
<i>ReliabilityQosPolicy</i>	<code>reliability()</code>	No (*)
<i>DestinationOrderQosPolicy</i>	<code>destination_order()</code>	No
<i>HistoryQosPolicy</i>	<code>history()</code>	No
<i>ResourceLimitsQosPolicy</i>	<code>resource_limits()</code>	No
<i>LifespanQosPolicy</i>	<code>lifespan()</code>	Yes
<i>UserDataQosPolicy</i>	<code>user_data()</code>	Yes
<i>OwnershipQosPolicy</i>	<code>ownership()</code>	No
<i>PropertyPolicyQos</i>	<code>properties()</code>	Yes
<i>RTPSEndpointQos</i>	<code>endpoint()</code>	Yes
<i>ReaderResourceLimitsQos</i>	<code>reader_resource_limits()</code>	Yes
<i>RTPSEndpointTimeBasedFilterQosPolicy</i>	<code>time_based_filter()</code>	Yes
<i>ReaderDataLifecycleQosPolicy</i>	<code>reader_data_lifecycle()</code>	Yes
<i>RTPSReliableReaderQos</i>	<code>reliable_reader_qos()</code>	Yes (*)
<i>TypeConsistencyQos</i>	<code>type_consistency()</code>	Yes
<i>DataSharingQosPolicy</i>	<code>data_sharing()</code>	No
<code>boolean</code>	<code>expects_inline_qos()</code>	Yes

The following non-consolidated property-assigned QoS apply to DataReaders:

Property name	Non-consolidated QoS
partitions	<i>Endpoint Partitions</i>

Refer to the detailed description of each QosPolicy class for more information about their usage and default values.

Note: Reliability kind (whether the publication is reliable or best effort) is not mutable. However, the `max_blocking_time` data member of *ReliabilityQosPolicy* can be modified any time.

Note: Not all data members of *RTPSReliableReaderQos* are mutable, please refer to *RTPSReliableReaderQos* for more information.

The QoS value of a previously created *DataReader* can be modified using the `DataReader::set_qos()` member function.

```
// Create a DataReader with default DataReaderQos
DataReader* data_reader =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
```

(continues on next page)

(continued from previous page)

```

    // 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) != RETCODE_OK)
{
    // Error
    return;
}

// Create a DataReader with the new default DataReaderQos.
DataReader* data_reader_with_qos_type1 =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader_with_qos_type1)
{
    // Error
    return;
}

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

// Modify QoS attributes

```

(continues on next page)

(continued from previous page)

```

// (...)

// Set as the new default DataReaderQos
if (subscriber->set_default_datareader_qos(qos_type2) != RETCODE_OK)
{
    // Error
    return;
}

// Create a DataReader with the new default DataReaderQos.
DataReader* data_reader_with_qos_type2 =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DataReaderQos to the original default constructed values
if (subscriber->set_default_datareader_qos(DATAREADER_QOS_DEFAULT)
    != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (subscriber->set_default_datareader_qos(DataReaderQos())
    != 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

```

(continues on next page)

(continued from previous page)

```

if (data_reader->set_qos(DATAREADER_QOS_DEFAULT) != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (data_reader->set_qos(subscriber->get_default_datareader_qos())
    != RETCODE_OK)
{
    // Error
    return;
}

```

Note: The value `DATAREADER_QOS_DEFAULT` has different meaning depending on where it is used:

- On `create_datareader()` and `DataReader::set_qos()` it refers to the default `DataReaderQos` as returned by `get_default_datareader_qos()`.
- On `set_default_datareader_qos()` it refers to the default constructed `DataReaderQos()`.

DataReaderListener

`DataReaderListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *DataReader*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`DataReaderListener` defines the following callbacks:

- `on_data_available()`: There is new data available for the application on the `DataReader`. There is no queuing of invocations to this callback, meaning that if several new data changes are received at once, only one callback invocation may be issued for all of them, instead of one per change. If the application is retrieving the received data on this callback, it must keep *reading data* until no new changes are left.
- `on_subscription_matched()`: The `DataReader` has found a *DataWriter* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a `DataWriter` that was previously considered to be matched. It is also triggered when a matched `DataWriter` has changed its *DataWriterQos*.
- `on_requested_deadline_missed()`: The `DataReader` did not receive data within the deadline period configured on its *DataReaderQos*. It will be called for each deadline period and data instance for which the `DataReader` missed data.
- `on_requested_incompatible_qos()`: The `DataReader` has found a `DataWriter` that matches the *Topic* and has a common partition, but with a QoS that is incompatible with the one defined on the `DataReader`.
- `on_liveliness_changed()`: The liveliness status of a matched `DataWriter` has changed. Either a `DataWriter` that was *inactive* has become *active* or the other way around.
- `on_sample_rejected()`: A received data sample was rejected. See *SampleRejectedStatus* for further information.
- `on_sample_lost()`: A data sample was lost and will never be received. See *SampleLostStatus* for further information.

Important: For more information about callbacks and its hierarchy, please refer to [Listener](#).

```
class CustomDataReaderListener : public DataReaderListener
{
public:
    CustomDataReaderListener()
        : DataReaderListener()
    {
    }

    virtual ~CustomDataReaderListener()
    {
    }

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

    void on_subscription_matched(
        DataReader* reader,
        const SubscriptionMatchedStatus& info) override
    {
        static_cast<void>(reader);
        if (info.current_count_change == 1)
        {
            std::cout << "Matched a remote DataWriter" << std::endl;
        }
        else if (info.current_count_change == -1)
        {
            std::cout << "Unmatched a remote DataWriter" << std::endl;
        }
    }

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

    void on_liveliness_changed(
        DataReader* reader,
        const LivelinessChangedStatus& info) override
    {
    }
}
```

(continues on next page)

(continued from previous page)

```

static_cast<void>(reader);
if (info.alive_count_change == 1)
{
    std::cout << "A matched DataWriter has become active" << std::endl;
}
else if (info.not_alive_count_change == 1)
{
    std::cout << "A matched DataWriter has become inactive" << std::endl;
}
}

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

void on_requested_incompatible_qos(
    DataReader* reader,
    const RequestedIncompatibleQosStatus& info) override
{
    std::cout << "Found a remote Topic with incompatible QoS (QoS ID: " << info.last_
policy_id <<
    ")" << std::endl;
}

void on_sample_lost(
    DataReader* reader,
    const SampleLostStatus& info) override
{
    static_cast<void>(reader);
    static_cast<void>(info);
    std::cout << "A data sample was lost and will not be received" << std::endl;
}

};

```

Creating a DataReader

A *DataReader* always belongs to a *Subscriber*. Creation of a DataReader is done with the `create_datareader()` member function on the Subscriber instance, that acts as a factory for the DataReader.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- The *DataReaderQos* describing the behavior of the DataReader. If the provided value is `DATAREADER_QOS_DEFAULT`, the value of the *Default DataReaderQos* is used. If the provided value is `DATAREADER_QOS_USE_TOPIC_QOS`, the values of the default QoS and the provided *TopicQoS* are used, whereby any policy that is set on the *TopicQoS* overrides the corresponding policy on the default QoS.

Optional arguments are:

- A Listener derived from *DataReaderListener*, implementing the callbacks that will be triggered in response to events and state changes on the DataReader. By default empty callbacks are used.
- A StatusMask that activates or deactivates triggering of individual callbacks on the DataReaderListener. By default all events are enabled.

`create_datareader()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DataReader with default DataReaderQos and no Listener
// The value DATAREADER_QOS_DEFAULT is used to denote the default QoS.
DataReader* data_reader_with_default_qos =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader_with_default_qos)
{
    // Error
    return;
}

// A custom DataReaderQos can be provided to the creation method
DataReaderQos custom_qos;

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

DataReader* data_reader_with_custom_qos =
    subscriber->create_datareader(topic, custom_qos);
if (nullptr == data_reader_with_custom_qos)
{
    // Error
    return;
}

// Create a DataReader with default QoS and a custom Listener.
// CustomDataReaderListener inherits from DataReaderListener.
// The value DATAREADER_QOS_DEFAULT is used to denote the default QoS.
CustomDataReaderListener custom_listener;
DataReader* data_reader_with_default_qos_and_custom_listener =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT, &custom_listener);
if (nullptr == data_reader_with_default_qos_and_custom_listener)
{
    // Error
    return;
}

// Create a DataReader with default QoS and a custom TopicQos.
// The value DATAREADER_QOS_USE_TOPIC_QOS is used to denote the default QoS
// and to override the TopicQos.
Topic* topic;
DataReader* data_reader_with_default_qos_and_custom_topic_qos =
    subscriber->create_datareader(topic, DATAREADER_QOS_USE_TOPIC_QOS);
if (nullptr == data_reader_with_default_qos_and_custom_topic_qos)
{
```

(continues on next page)

(continued from previous page)

```
// Error
return;
}
```

Profile based creation of a DataReader

Instead of using a `DataReaderQos`, the name of a profile can be used to create a `DataReader` with the `create_datareader_with_profile()` member function on the `Subscriber` instance.

Mandatory arguments are:

- A Topic bound to the data type that will be transmitted.
- A string with the name that identifies the `DataReader`.

Optional arguments are:

- A Listener derived from `DataReaderListener`, implementing the callbacks that will be triggered in response to events and state changes on the `DataReader`. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the `DataReaderListener`. By default all events are enabled.

`create_datareader_with_profile()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DataReader using a profile and no Listener
DataReader* data_reader_with_profile =
    subscriber->create_datareader_with_profile(topic, "data_reader_profile");
if (nullptr == data_reader_with_profile)
{
    // Error
    return;
}

// Create a DataReader using a profile and a custom Listener.
// CustomDataReaderListener inherits from DataReaderListener.
CustomDataReaderListener custom_listener;
DataReader* data_reader_with_profile_and_custom_listener =
    subscriber->create_datareader_with_profile(topic, "data_reader_profile", &custom_listener);
if (nullptr == data_reader_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Creating a DataWriter with a custom PayloadPool

A custom *PayloadPool* can be passed as an argument during the creation of a *DataReader*. This allows for customizing the management of the information exchanged between DataWriters and DataReaders. The same configuration can be set in the *opposite endpoint*.

```
// A DataReaderQos must be provided to the creation method
DataReaderQos qos;

// Create PayloadPool
std::shared_ptr<CustomPayloadPool> payload_pool = std::make_shared<CustomPayloadPool>();

DataReader* data_reader = subscriber->create_datareader(topic, qos, nullptr,
↳ StatusMask::all(), payload_pool);
if (nullptr == data_reader)
{
    // Error
    return;
}
```

This configuration can be performed also in the *RTPS layer*. The *customization example* applies both layers.

Deleting a DataReader

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

Note: A DataReader can only be deleted if all Entities belonging to the DataReader (QueryConditions) have already been deleted. Otherwise, the function will issue an error and the DataReader will not be deleted. This can be performed by using the `delete_contained_entities()` member function of the *DataReader*.

```
// Create a DataReader
DataReader* data_reader =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
    // Error
    return;
}

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

// Delete the entities the DataReader created
if (data_reader->delete_contained_entities() != RETCODE_OK)
{
    // DataReader failed to delete the entities it created.
    return;
}

// Delete the DataReader
```

(continues on next page)

(continued from previous page)

```

if (subscriber->delete_datareader(data_reader) != 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 its interpretation. For example, if the *valid_data* value is *false*, the *DataReader* is not informing the application about a new value in the data instance, but a change on its status, and the returned data value must be discarded.

Please, refer to the section *Accessing received data* for more information regarding how received data can be accessed on the *DataReader*.

The following sections describe the data members of *SampleInfo* and the meaning of each one in relation to the returned sample data.

- *sample_state*
- *view_state*
- *instance_state*
- *disposed_generation_count*
- *no_writers_generation_count*
- *sample_rank*
- *generation_rank*
- *absolute_generation_rank*
- *source_timestamp*
- *instance_handle*
- *publication_handle*
- *valid_data*
- *sample_identity*
- *related_sample_identity*

sample_state

sample_state indicates whether or not the corresponding data sample has already been read previously. It can take one of these values:

- **READ**: This is the first time this data sample has been retrieved.
- **NOT_READ**: The data sample has already been *read* or *taken* previously.

view_state

`view_state` indicates whether or not this is the very first sample of this data instance that the `DataReader` retrieves. It can take one of these values:

- **NEW**: This is the first time a sample of this instance is retrieved.
- **NOT_NEW**: Other samples of this instance have been retrieved previously.

instance_state

`instance_state` indicates whether the instance is currently in existence or it has been disposed. In the latter case, it also provides information about the reason for the disposal. It can take one of these values:

- **ALIVE**: The instance is currently in existence.
- **NOT_ALIVE_DISPOSED**: A remote *DataWriter* disposed the instance.
- **NOT_ALIVE_NO_WRITERS**: The `DataReader` disposed the instance because no remote `DataWriter` that was publishing the instance is *alive*.

disposed_generation_count

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

no_writers_generation_count

`no_writers_generation_count` indicates the number of times the instance had become alive after it was disposed as **NOT_ALIVE_NO_WRITERS**.

sample_rank

`sample_rank` indicates the number of samples of the same instance that have been received after this one. For example, a value of 5 means that there are 5 newer samples available on the `DataReader`.

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

generation_rank

`generation_rank` indicates the number of times the instance was disposed and become alive again between the time the sample was received and the time the most recent sample of the same instance that is still held in the collection was received.

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

absolute_generation_rank

`absolute_generation_rank` indicates the number of times the instance was disposed and become alive again between the time the sample was received and the time the most recent sample of the same instance (which may not be in the collection) was received.

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

source_timestamp

`source_timestamp` holds the time stamp provided by the DataWriter when the sample was published.

instance_handle

`instance_handle` handles of the local instance.

publication_handle

`publication_handle` handles of the DataWriter that published the data change.

valid_data

`valid_data` is a boolean that indicates whether the data sample contains a change in the value or not. Samples with this value set to false are used to communicate a change in the instance status, e.g., a change in the liveliness of the instance. In this case, the data sample should be dismissed as all the relevant information is in the data members of `SampleInfo`.

sample_identity

`sample_identity` is an extension for requester-replier configuration. It contains the DataWriter and the sequence number of the current message, and it is used by the replier to fill the *related_sample_identity* when it sends the reply.

related_sample_identity

`related_sample_identity` is an extension for requester-replier configuration. On reply messages, it contains the *sample_identity* of the related request message. It is used by the requester to be able to link each reply to the appropriate request.

Accessing received data

The application can access and consume the data values received on the *DataReader* by *reading* or *taking*.

- **Reading** is done with any of the following member functions:
 - `DataReader::read_next_sample()` reads the next, non-previously accessed data value available on the `DataReader`, and stores it in the provided data buffer.
 - `DataReader::read()`, `DataReader::read_instance()`, and `DataReader::read_next_instance()` provide mechanisms to get a collection of samples matching certain conditions.
- **Taking** is done with any of the following member functions:
 - `DataReader::take_next_sample()` reads the next, non-previously accessed data value available on the `DataReader`, and stores it in the provided data buffer.
 - `DataReader::take()`, `DataReader::take_instance()`, and `DataReader::take_next_instance()` provide mechanisms to get a collection of samples matching certain conditions.

When taking data, the returned samples are also removed from the `DataReader`, so they are no longer accessible.

When there is no data in the `DataReader` matching the required conditions, all the operations will return `NO_DATA` and output parameter will remain unchanged.

In addition to the data values, the data access operations also provide `SampleInfo` instances with additional information that help interpreting the returned data values, like the originating *DataWriter* or the publication time stamp. Please, refer to the *SampleInfo* section for an extensive description of its contents.

Loaning and Returning Data and SampleInfo Sequences

The `DataReader::read()` and `DataReader::take()` operations (and their variants) return information to the application in two sequences:

- Received DDS data samples in a sequence of the data type
- Corresponding information about each DDS sample in a `SampleInfo` sequence

These sequences are parameters that are passed by the application code into the `DataReader::read()` and `DataReader::take()` operations. When the passed sequences are empty (they are initialized but have a maximum length of 0), the middleware will fill those sequences with memory directly loaned from the receive queue itself. There is no copying of the data or `SampleInfo` when the contents of the sequences are loaned. This is certainly the most efficient way for the application code to retrieve the data.

When doing so, however, the code must return the loaned sequences back to the middleware, so that they can be reused by the receive queue. If the application does not return the loan by calling the `DataReader::return_loan()` operation, then Fast DDS will eventually run out of memory to store DDS data samples received from the network for that `DataReader`. See the code below for an example of borrowing and returning loaned sequences.

```
// Sequences are automatically initialized to be empty (maximum == 0)
FooSeq data_seq;
SampleInfoSeq info_seq;

// with empty sequences, a take() or read() will return loaned
// sequence elements
ReturnCode_t ret_code = data_reader->take(data_seq, info_seq,
                                         LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
                                         ANY_VIEW_STATE, ANY_INSTANCE_STATE);
```

(continues on next page)

(continued from previous page)

```
// process the returned data

// must return the loaned sequences when done processing
data_reader->return_loan(data_seq, info_seq);
```

Processing returned data

After calling the `DataReader::read()` or `DataReader::take()` operations, accessing the data on the returned sequences is quite easy. The sequences API provides a `length()` operation returning the number of elements in the collections. The application code just needs to check this value and use the `[]` operator to access the corresponding elements. Elements on the DDS data sequence should only be accessed when the corresponding element on the `SampleInfo` sequence indicate that valid data is present. When using Data Sharing, it is also important to check that the sample is valid (i.e, not replaced, refer to *DataReader and DataWriter history coupling* for further information in this regard).

```
// Sequences are automatically initialized to be empty (maximum == 0)
FooSeq data_seq;
SampleInfoSeq info_seq;

// with empty sequences, a take() or read() will return loaned
// sequence elements
ReturnCode_t ret_code = data_reader->take(data_seq, info_seq,
                                           LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
                                           ANY_VIEW_STATE, ANY_INSTANCE_STATE);

// process the returned data
if (ret_code == RETCODE_OK)
{
    // Both info_seq.length() and data_seq.length() will have the number of samples_
    ↪returned
    for (FooSeq::size_type n = 0; n < info_seq.length(); ++n)
    {
        // Only samples with valid data should be accessed
        if (info_seq[n].valid_data && data_reader->is_sample_valid(&data_seq[n], &
    ↪info_seq[n]))
        {
            // Do something with data_seq[n]
        }
    }

    // must return the loaned sequences when done processing
    data_reader->return_loan(data_seq, info_seq);
}
```

Accessing data on callbacks

When the DataReader receives new data values from any matching DataWriter, it informs the application through two Listener callbacks:

- `on_data_available()`.
- `on_data_on_readers()`.

These callbacks can be used to retrieve the newly arrived data, as in the following example.

```
class CustomizedDataReaderListener : public DataReaderListener
{
public:
    CustomizedDataReaderListener()
        : DataReaderListener()
    {
    }

    virtual ~CustomizedDataReaderListener()
    {
    }

    void on_data_available(
        DataReader* reader) override
    {
        // Create a data and SampleInfo instance
        Foo data;
        SampleInfo info;

        // Keep taking data until there is nothing to take
        while (reader->take_next_sample(&data, &info) == RETCODE_OK)
        {
            if (info.valid_data)
            {
                // Do something with the data
                std::cout << "Received new data value for topic "
                    << reader->get_topicdescription()->get_name()
                    << std::endl;
            }
            else
            {
                std::cout << "Remote writer for topic "
                    << reader->get_topicdescription()->get_name()
                    << " is dead" << std::endl;
            }
        }
    }
};
```

Note: If several new data changes are received at once, the callbacks may be triggered just once, instead of once per

change. The application must keep *reading* or *taking* until no new changes are available.

Accessing data with a waiting thread

Wait-sets and DataAvailable status condition

Instead of relying on the Listener to try and get new data values, the application can also dedicate a thread to wait until any new data is available on the DataReader. This can be done using a wait-set to wait for a change on the *DataAvailable* status.

```
// Create a DataReader
DataReader* data_reader =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
    // Error
    return;
}

// Prepare a wait-set to wait for data on the DataReader
WaitSet wait_set;
StatusCondition& condition = data_reader->get_statuscondition();
condition.set_enabled_statuses(StatusMask::data_available());
wait_set.attach_condition(condition);

// Create a data and SampleInfo instance
Foo data;
SampleInfo info;

// Define a timeout of 5 seconds
eprosima::fastrtps::Duration_t timeout (5, 0);

// Loop reading data as it arrives
// This will make the current thread to be dedicated exclusively to
// waiting and reading data until the remote DataWriter dies
while (true)
{
    ConditionSeq active_conditions;
    if (RETCODE_OK == wait_set.wait(active_conditions, timeout))
    {
        while (RETCODE_OK == data_reader->take_next_sample(&data, &info))
        {
            if (info.valid_data)
            {
                // Do something with the data
                std::cout << "Received new data value for topic "
                    << topic->get_name()
                    << std::endl;
            }
            else
            {

```

(continues on next page)

(continued from previous page)

```

        // If the remote writer is not alive, we exit the reading loop
        std::cout << "Remote writer for topic "
                  << topic->get_name()
                  << " is dead" << std::endl;

        break;
    }
}
else
{
    std::cout << "No data this time" << std::endl;
}
}

```

DataReader non-blocking calls

The same could be achieved using the `DataReader::wait_for_unread_message()` member function, that blocks until a new data sample is available or the given timeout expires. If no new data was available after the timeout expired, it will return with value `false`. This function returning with value `true` means there is new data available on the *DataReader* ready for the application to retrieve.

```

// Create a DataReader
DataReader* data_reader =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == data_reader)
{
    // Error
    return;
}

// Create a data and SampleInfo instance
Foo data;
SampleInfo info;

// Define a timeout of 5 seconds
eprosima::fastrtps::Duration_t timeout (5, 0);

// Loop reading data as it arrives
// This will make the current thread to be dedicated exclusively to
// waiting and reading data until the remote DataWriter dies
while (true)
{
    if (data_reader->wait_for_unread_message(timeout))
    {
        if (RETCODE_OK == data_reader->take_next_sample(&data, &info))
        {
            if (info.valid_data)
            {
                // Do something with the data
                std::cout << "Received new data value for topic "
                          << topic->get_name()

```

(continues on next page)

(continued from previous page)

```

        << std::endl;
    }
    else
    {
        // If the remote writer is not alive, we exit the reading loop
        std::cout << "Remote writer for topic "
                  << topic->get_name()
                  << " is dead" << std::endl;
        break;
    }
}
}
else
{
    std::cout << "No data this time" << std::endl;
}
}

```

7.18.5 Topic

A Topic conceptually fits between publications and subscriptions. Each publication channel must be unambiguously identified by the subscriptions in order to receive only the data flow they are interested in, and not data from other publications. A Topic serves this purpose, allowing publications and subscriptions that share the same Topic to match and start communicating. In that sense, the Topic acts as a description for a data flow.

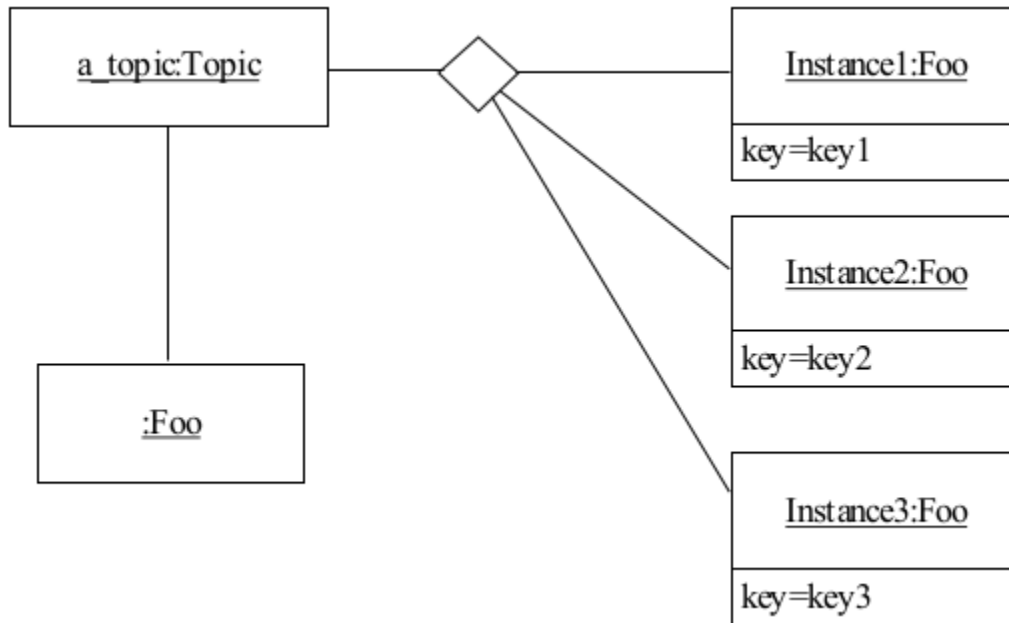
Publications are always linked to a single *Topic*, while subscriptions are linked to a broader concept of *TopicDescription*.

Fig. 8: Topic class diagram

Topics, keys and instances

By definition, a Topic is linked to a single data type, so each data sample related to a Topic could be understood as an update on the information described by the data type. However, it is possible to include a logical separation and have, within the same Topic, several instances referring to the same data type. Thus, the received data sample will be an update for a specific instance of that Topic. Therefore, a Topic identifies data of a single type, ranging from one single instance to a whole collection of instances of that given type, as shown in the figure below.

The different instances gathered under the same topic are distinguishable by means of one or more data fields that form the key to that data set. The key description has to be indicated to the middleware. The rule is simple: different data values with the same key value represent successive data samples for the same instance, while different data values with different keys represent different topic instances. If no key is provided, the data set associated with the Topic is restricted to a single instance. Please refer to *Data types with a key* for more information about how to set the key in *eProsima Fast DDS*.

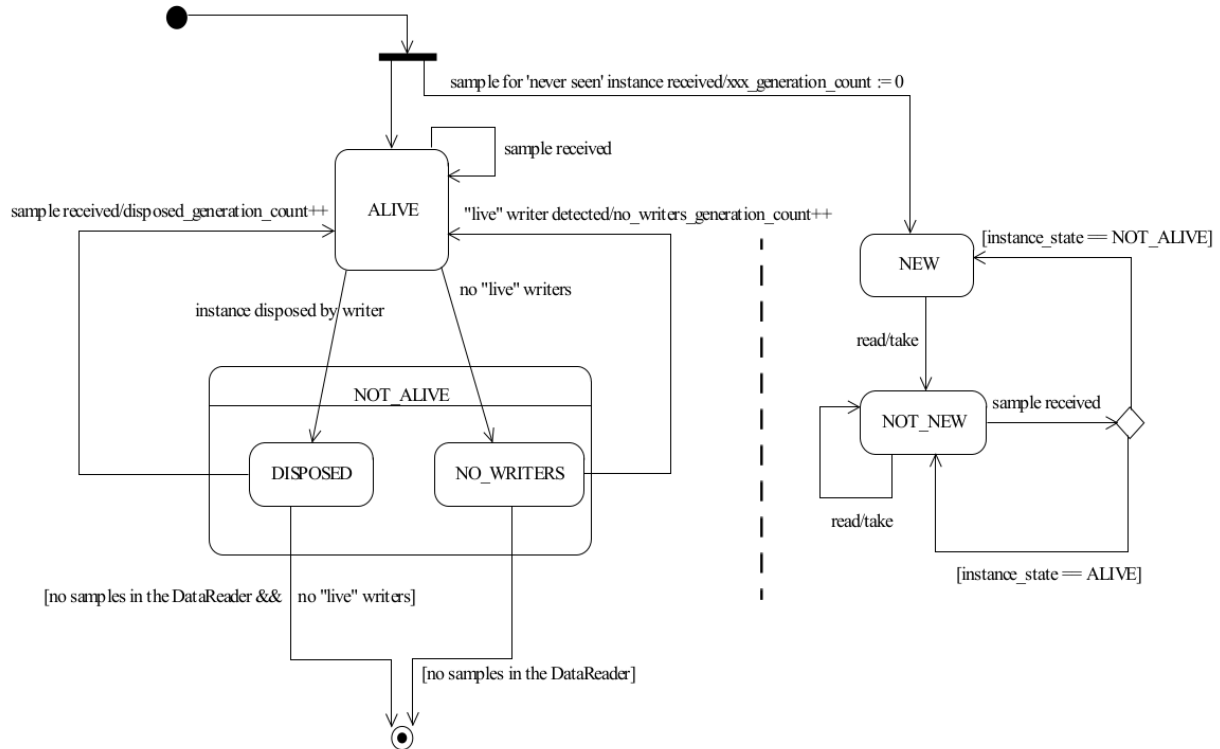


Instance advantages

The advantage of using instances instead of creating a new *DataWriter*, *DataReader*, and *Topic* is that the corresponding entity is already created and discovered. Consequently, there is less memory usage, and no new discovery (with the related metatraffic involved as explained in *Discovery*) is necessary. Another advantage is that several QoS are applied per topic instance; e.g. the *HistoryQosPolicy* is kept for each instance in the *DataWriter*. Thus, instances could be tuned to a wide range of applications.

Instance lifecycle

When reading or taking data from the *DataReader* (as explained in *Accessing received data*), a *SampleInfo* is also returned. This *SampleInfo* provides additional information about the instance lifecycle, specifically with the *view_state*, *instance_state*, *disposed_generation_count*, and *no_writers_generation_count*. The diagram below shows the state-chart of *instance_state* and *view_state* for a single instance.



Practical applications

This section provides a couple of examples to help clarify the use of DDS instances.

Commercial flights tracking

Airspace and the air traffic going through it are typically managed by the air traffic controllers that are in charge of organizing the air traffic, preventing collisions, and providing information. In this scenario, each air traffic control center takes responsibility for a specific flight area and delivers the data to the airspace traffic management system, which unifies the flight information.

Any time an air traffic control center discovers a plane coming into its controlled flight zone, tracking information about that specific flight is notified to the airspace traffic management center. Such a flow of information could be implemented by means of DDS by creating a specific Topic where the information related to the flight location is published. In that case, the management center would be required to create, if not existing previously, the corresponding Topic and DataReader to have access to the flight information, with the corresponding memory consumption and discovery metatraffic required. On the other hand, a cleverer implementation could leverage topic instances to relay the information from the local air traffic control centers to the airspace traffic management center. The topic instances might be identified using the airline name and the flight number (i.e. *IBERIA 1234*) as Topic instance key. The sample data being relayed would be the location of each flight being tracked at any given time. The following IDL defines the data described model:

```

struct FlightPosition
{
    // Unique ID: airline name
    @key string<256> airline_name;

    // Unique ID: flight number
    @key short flight_number;
}
  
```

(continued from previous page)

```
// Coordinates
double latitude;
double longitude;
double altitude;
};
```

Once a new flight is discovered by a control center, the corresponding instance is registered into the system:

```
// Create data sample
FlightPosition first_flight_position;

// Specify the flight instance
first_flight_position.airline_name("IBERIA");
first_flight_position.flight_number(1234);

// Register instance
eprosima::fastrtps::rtps::InstanceHandle_t first_flight_handle =
    data_writer->register_instance(&first_flight_position);
```

register_instance() returns an InstanceHandle_t which can be used to efficiently call the next operations (i.e. write(), dispose(), or unregister_instance()) over the instance. The returned InstanceHandle_t contains the instance keyhash so it does not have to be recalculated again from the data sample. In case of following this approach, the application must take charge of mapping the instance handles to the corresponding instances.

```
// Update position value received from the plane
first_flight_position.latitude(39.08);
first_flight_position.longitude(-84.21);
first_flight_position.altitude(1500);

// Write sample to the instance
data_writer->write(&first_flight_position, first_flight_handle);
```

On the other hand, the user application could directly call the DataWriter instance operations with a NIL instance handle. In this case, the instance handle would be calculated every time an operation is done over the instance, which can be time consuming depending on the specific data type being used.

```
// New data sample
FlightPosition second_flight_position;

// New instance
second_flight_position.airline_name("RYANAIR");
second_flight_position.flight_number(4321);

// Update plane location
second_flight_position.latitude(40.02);
second_flight_position.longitude(-84.32);
second_flight_position.altitude(5000);

// Write sample directly without registering the instance
data_writer->write(&second_flight_position);
```


Warning: The correct management of the instance handles in the user application is paramount. Otherwise, a sample corresponding to a different instance could wrongly update the instance which handle the user has passed to the operation (if a non NIL instance is provided, the instance handle is not recalculated, trusting that the one passed by the user is the correct one). The following code updates the first instance of this example with the information coming from the second instance.

```
data_writer->write(&second_flight_position, first_flight_handle);
```

Once the plane leaves the controlled area, the air traffic control center may unregister the instance. Unregistering implies that the DataWriter for this specific center has no more information about the unregistered instance, and in this way the matched DataReaders in the management center are notified. The flight is still in the air but out of scope of this particular DataWriter. The instance is alive but no longer tracked by this center.

```
data_writer->unregister_instance(&first_flight_position, first_flight_handle);
data_writer->unregister_instance(&second_flight_position, HANDLE_NIL);
```

Finally, when the flight lands, the instance may be disposed. This means, in this specific example, that as far as the DataWriter knows, the instance no longer exists and should be considered not alive. With this operation, the DataWriter conveys this information to the matched DataReaders.

```
data_writer->dispose(&first_flight_position, first_flight_handle);
data_writer->dispose(&second_flight_position, HANDLE_NIL);
```

From the management center point of view, the samples are read using the same DataReader subscribed to the Topic where the instances are being published. However, `valid_data` must be checked to ensure that the sample received contains a data sample. Otherwise, a change of the instance state is being notified. *Instance lifecycle* contains a diagram showing the instance statechart.

```
if (RETCODE_OK == data_reader->take_next_sample(&data, &info))
{
    if (info.valid_data)
    {
        // Data sample has been received
    }
    else if (info.instance_state == NOT_ALIVE_DISPOSED_INSTANCE_STATE)
    {
        // A remote DataWriter has disposed the instance
    }
    else if (info.instance_state == NOT_ALIVE_NO_WRITERS_INSTANCE_STATE)
    {
        // None of the matched DataWriters are writing in the instance.
        // The instance can be safely disposed.
    }
}
```

Relational databases

Consider now that the air traffic management center wants to keep a database with the flights being tracked. Using DDS instances, maintaining a relational database is almost direct. The instance key (unique identifier of the instance) is analogous to the primary key of the database. Thus, the airspace traffic management center can keep the latest update for each instance in a table like the one below:

Instance handle [PK]	Data
1	Position1
2	Position2
3	Position3
4	Position4
5	Position5

In this case, every time a new sample is received, the corresponding instance entry in the database will be updated with the latest known location. Disposing the instance may translate in erasing the corresponding data from the database. In this scenario, registering and unregistering the instances does not reflect in the database, although if the `instance_state` and `view_state` are also persisted, then the instance lifecycle could be tracked as well. A DataWriter communicating that it is going to be publishing data about a specific instance is of no interest to the database until a new data is received and then an insert is directly done with the new discovered instance.

Historical data can also be stored in the relational database, even though depending on the use case, a time series database might be considered to improve efficiency. In the scenario being considered, the sample timestamp could be used, besides the instance handle, as primary key to be able to access the historical tracking data of an specific flight.

Instance handle [PK]	Source Timestamp [PK]	Data
1	1	Position1
2	1	Position2
1	2	Position3
1	3	Position4
2	2	Position5

In this case, looking for a specific instance handle would return the flight tracking information:

Instance handle [Fixed]	Source Timestamp	Data
1	1	Position1
1	2	Position3
1	3	Position4

Whereas looking for a specific timestamp would allow to have a picture of the different flight locations at a specific time:

Instance handle	Source Timestamp [Fixed]	Data
1	2	Position3
2	2	Position5

TopicDescription

TopicDescription is an abstract class that serves as the base for all classes describing a data flow. Applications will not create instances of **TopicDescription** directly, they must create instances of one of its specializations instead. At the moment, the only specializations implemented are *Topic*, and *ContentFilteredTopic*.

Topic

A **Topic** is a specialization of the broader concept of *TopicDescription*. A Topic represents a single data flow between *Publisher* and *Subscriber*, providing:

- The name to identify the data flow.
- The data type that is transmitted on that flow.
- The QoS values related to the data itself.

The behavior of the Topic can be modified with the QoS values specified on *TopicQos*. The QoS values can be set at the creation of the Topic, or modified later with the `Topic::set_qos()` member function.

Like other Entities, Topic accepts a Listener that will be notified of status changes on the Topic.

Please refer to *Creating a Topic* for more information about how to create a Topic.

TopicQos

TopicQos controls the behavior of the Topic. Internally it contains the following QosPolicy objects:

QosPolicy class	Accessor	Mutable
<i>TopicDataQosPolicy</i>	<code>topic_data()</code>	Yes
<i>DurabilityQosPolicy</i>	<code>durability()</code>	Yes
<i>DurabilityServiceQosPolicy</i>	<code>durability_service()</code>	Yes
<i>DeadlineQosPolicy</i>	<code>deadline()</code>	Yes
<i>LatencyBudgetQosPolicy</i>	<code>latency_budget()</code>	Yes
<i>LivelinessQosPolicy</i>	<code>liveliness()</code>	Yes
<i>ReliabilityQosPolicy</i>	<code>reliability()</code>	Yes
<i>DestinationOrderQosPolicy</i>	<code>destination_order()</code>	Yes
<i>HistoryQosPolicy</i>	<code>history()</code>	Yes
<i>ResourceLimitsQosPolicy</i>	<code>resource_limits()</code>	Yes
<i>TransportPriorityQosPolicy</i>	<code>transport_priority()</code>	Yes
<i>LifespanQosPolicy</i>	<code>lifespan()</code>	Yes
<i>OwnershipQosPolicy</i>	<code>ownership()</code>	Yes
<i>DataRepresentationQosPolicy</i>	<code>representation()</code>	Yes

Refer to the detailed description of each QosPolicy-api class for more information about their usage and default values.

The QoS value of a previously created Topic can be modified using the `Topic::set_qos()` member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↳DEFAULT);
if (nullptr == participant)
{
```

(continues on next page)

(continued from previous page)

```

    // Error
    return;
}

// Create a Topic with default TopicQos
Topic* topic =
    participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
TopicQos qos = topic->get_qos();

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

// Assign the new QoS to the object
topic->set_qos(qos);

```

Default TopicQos

The default *TopicQos* refers to the value returned by the `get_default_topic_qos()` member function on the *DomainParticipant* instance. The special value `TOPIC_QOS_DEFAULT` can be used as QoS argument on `create_topic()` or `Topic::set_qos()` member functions to indicate that the current default TopicQos should be used.

When the system starts, the default TopicQos is equivalent to the default constructed value `TopicQos()`. The default TopicQos can be modified at any time using the `get_default_topic_qos()` member function on the *DomainParticipant* instance. Modifying the default TopicQos will not affect already existing Topic instances.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
TopicQos qos_type1 = participant->get_default_topic_qos();

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

// Set as the new default TopicQos
if (participant->set_default_topic_qos(qos_type1) != RETCODE_OK)
{

```

(continues on next page)

(continued from previous page)

```

    // Error
    return;
}

// Create a Topic with the new default TopicQos.
Topic* topic_with_qos_type1 =
    participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic_with_qos_type1)
{
    // Error
    return;
}

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

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

// Set as the new default TopicQos
if (participant->set_default_topic_qos(qos_type2) != 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)
    != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if (participant->set_default_topic_qos(TopicQos())
    != 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) != RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if (topic->set_qos(participant->get_default_topic_qos())
    != RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value TOPIC_QOS_DEFAULT has different meaning depending on where it is used:

- On `create_topic()` and `Topic::set_qos()` it refers to the default `TopicQos` as returned by `get_default_topic_qos()`.
 - On `get_default_topic_qos()` it refers to the default constructed `TopicQos()`.
-

ContentFilteredTopic

A `ContentFilteredTopic` is a specialization of the broader concept of *TopicDescription*. A `ContentFilteredTopic` is a `Topic` with filtering properties. It makes it possible to subscribe to a `Topic` while at the same time specify interest on a subset of the `Topic`'s data.

Important: Note that a `ContentFilteredTopic` can only be used to create a `DataReader`, not a `DataWriter`.

A `ContentFilteredTopic` provides a relationship between a *Topic*, called the related topic, and some user-defined filtering properties:

- A **filter expression**, which establishes a logical expression on the content of the related topic. It is similar to the `WHERE` clause in a SQL statement.
- A list of **expression parameters**, which give values to the parameters present in the filter expression. There must be one parameter string for each parameter in the filter expression.

Note that a `ContentFilteredTopic` is *not* an `Entity`, and thus it has neither QoS nor listener. A `DataReader` created with a `ContentFilteredTopic` will use the QoS from the related topic. Multiple `DataReaders` can be created for the same `ContentFilteredTopic`, and changing the filter properties of a `ContentFilteredTopic` will affect all `DataReaders` using it.

Please refer to *Filtering data on a Topic* and *Where is filtering applied: writer vs reader side* for more information about how to use `ContentFilteredTopic`.

TopicListener

`TopicListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *Topic*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`TopicListener` has the following callback:

- `on_inconsistent_topic()`: A remote `Topic` is discovered with the same name but different characteristics as another locally created `Topic`.

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

```
class CustomTopicListener : public TopicListener
{
public:
    CustomTopicListener()
        : TopicListener()
    {
    }

    virtual ~CustomTopicListener()
    {
    }
}
```

(continues on next page)

(continued from previous page)

```

void on_inconsistent_topic(
    Topic* topic,
    InconsistentTopicStatus status) override
{
    static_cast<void>(topic);
    static_cast<void>(status);
    std::cout << "Inconsistent topic received discovered" << std::endl;
}
};

```

Definition of data types

The definition of the data type exchanged in a *Topic* is divided in two classes: the `TypeSupport` and the `TopicDataType`.

`TopicDataType` describes the data type exchanged between a publication and a subscription, i.e., the data corresponding to a `Topic`. The user has to create a specialized class for each specific type that will be used by the application.

Any specialization of `TopicDataType` must be registered in the *DomainParticipant* before it can be used to create `Topic` objects. A `TypeSupport` object encapsulates an instance of `TopicDataType`, providing the functions needed to register the type and interact with the publication and subscription. To register the data type, create a new `TypeSupport` with a `TopicDataType` instance and use the `register_type()` member function on the `TypeSupport`. Then the `Topic` can be created with the registered type name.

Note: Registering two different data types on the same `DomainParticipant` with identical names is not allowed and will issue an error. However, it is allowed to register the same data type within the same `DomainParticipant`, with the same or different names. If the same data type is registered twice on the same `DomainParticipant` with the same name, the second registering will have no effect, but will not issue any error.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↳DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Register the data type in the DomainParticipant.
// If nullptr is used as name argument, the one returned by the type itself is used
TypeSupport custom_type_support(new CustomDataType());
custom_type_support.register_type(participant, nullptr);

// The previous instruction is equivalent to the following one
// Even if we are registering the same data type with the same name twice, no error will
    ↳be issued
custom_type_support.register_type(participant, custom_type_support.get_type_name());

```

(continues on next page)

(continued from previous page)

```

// Create a Topic with the registered type.
Topic* topic =
    participant->create_topic("topic_name", custom_type_support.get_type_name(),
    ↪ TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create an alias for the same data type using a different name.
custom_type_support.register_type(participant, "data_type_name");

// We can now use the aliased name to If no name is given, it uses the name returned by
    ↪ the type itself
Topic* another_topic =
    participant->create_topic("other_topic_name", "data_type_name", TOPIC_QOS_
    ↪ DEFAULT);
if (nullptr == another_topic)
{
    // Error
    return;
}

```

Dynamic data types

Instead of directly writing the specialized `TopicDataType` class, it is possible to dynamically define data types following the OMG Extensible and Dynamic Topic Types for DDS interface. Data types can also be described on an XML file that is dynamically loaded.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪ DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Load the XML file with the type description
DomainParticipantFactory::get_instance()->load_XML_profiles_file("example_type.xml");

// Retrieve the an instance of the desired type
DynamicType::_ref_type dyn_type;
DomainParticipantFactory::get_instance()->get_dynamic_type_builder_from_xml_by_name(
    ↪ "DynamicType", dyn_type);

// Register dynamic type
TypeSupport dyn_type_support(new DynamicPubSubType(dyn_type));
dyn_type_support.register_type(participant, nullptr);

```

(continues on next page)

(continued from previous page)

```
// 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 *XTypes* 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 *XTypes*.
- Manually implementing the `getKey()` member function on the `TopicDataType` and setting the `m_isGetKeyDefined` data member value to `true`.

Data types with key are used to define data sub flows on a single Topic. Data values with the same key on the same Topic represent data from the same sub-flow, while data values with different keys on the same Topic represent data from different sub-flows. The middleware keeps these sub-flows separated, but all will be restricted to the same QoS values of the Topic. If no key is provided, the data set associated with the Topic is restricted to a single flow.

Creating a Topic

A *Topic* always belongs to a *DomainParticipant*. Creation of a Topic is done with the `create_topic()` member function on the `DomainParticipant` instance, that acts as a factory for the Topic.

Mandatory arguments are:

- A string with the name that identifies the Topic.
- The name of the registered *data type* that will be transmitted.
- The *TopicQos* describing the behavior of the Topic. If the provided value is `TOPIC_QOS_DEFAULT`, the value of the *Default TopicQos* is used.

Optional arguments are:

- A Listener derived from *TopicListener*, implementing the callbacks that will be triggered in response to events and state changes on the Topic. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the `TopicListener`. By default all events are enabled.

`create_topic()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic with default TopicQos and no Listener
// The symbol TOPIC_QOS_DEFAULT is used to denote the default QoS.
Topic* topic_with_default_qos =
    participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic_with_default_qos)
{
    // Error
    return;
}

// A custom TopicQos can be provided to the creation method
TopicQos custom_qos;

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

Topic* topic_with_custom_qos =
    participant->create_topic("TopicName", "DataTypeName", custom_qos);
if (nullptr == topic_with_custom_qos)
{
    // Error
    return;
}

// Create a Topic with default QoS and a custom Listener.
// CustomTopicListener inherits from TopicListener.
// The symbol TOPIC_QOS_DEFAULT is used to denote the default QoS.
CustomTopicListener custom_listener;
Topic* topic_with_default_qos_and_custom_listener =
    participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT, &
    ↪custom_listener);
if (nullptr == topic_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a Topic

Instead of using a `TopicQos`, the name of a profile can be used to create a `Topic` with the `create_topic_with_profile()` member function on the `DomainParticipant` instance.

Mandatory arguments are:

- A string with the name that identifies the `Topic`.
- The name of the registered *data type* that will be transmitted.
- The name of the profile to be applied to the `Topic`.

Optional arguments are:

- A `Listener` derived from `TopicListener`, implementing the callbacks that will be triggered in response to events and state changes on the `Topic`. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the `TopicListener`. By default all events are enabled.

`create_topic_with_profile()` will return a null pointer if there was an error during the operation, e.g. if the provided `QoS` is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic using a profile and no Listener
Topic* topic_with_profile =
    participant->create_topic_with_profile("TopicName", "DataTypeName", "topic_
    ↪profile");
if (nullptr == topic_with_profile)
{
    // Error
    return;
}

// Create a Topic using a profile and a custom Listener.
// CustomTopicListener inherits from TopicListener.
CustomTopicListener custom_listener;
Topic* topic_with_profile_and_custom_listener =
    participant->create_topic_with_profile("TopicName", "DataTypeName", "topic_
    ↪profile", &custom_listener);
```

(continues on next page)

(continued from previous page)

```

if (nullptr == topic_with_profile_and_custom_listener)
{
    // Error
    return;
}

```

Deleting a Topic

A Topic can be deleted with the `delete_topic()` member function on the `DomainParticipant` instance where the Topic was created.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    →DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic
Topic* topic =
    participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

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

// Delete the Topic
if (participant->delete_topic(topic) != RETCODE_OK)
{
    // Error
    return;
}

```

Filtering data on a Topic

Creating a ContentFilteredTopic

A *ContentFilteredTopic* always belongs to a *DomainParticipant*. Creation of a `ContentFilteredTopic` is done with the `create_contentfilteredtopic()` member function on the `DomainParticipant` instance, that acts as a factory for the `ContentFilteredTopic`.

Mandatory arguments are:

- A string with the name that identifies the `ContentFilteredTopic`.

- The related Topic being filtered.
- A string with the filter expression indicating the conditions for a sample to be returned.
- A list of strings with the value of the parameters present on the filter expression.

Note: The number of parameter values cannot exceed the maximum set by the `expression_parameters` QoS configuration. The default (and absolute) maximum allowed as set by the OMG DDS Standard is 100.

Optional arguments are:

- A string with the name of the filter class to use for the filter creation. This allows the user to create filters different from the standard SQL like one (please refer to *Using custom filters*). Defaults to `FASTDDS_SQLFILTER_NAME` (DDSSQL).

Important: Setting an empty string as filter expression results in the disabling of the filtering. This can be used to enable/disable the DataReader filtering capabilities at any given time by simply *updating the filter expression*.

`create_contentfilteredtopic()` will return a null pointer if there was an error during the operation, e.g. if the related Topic belongs to a different DomainParticipant, a Topic with the same name already exists, syntax errors on the filter expression, or missing parameter values. It is advisable to check that the returned value is a valid pointer.

Note: Different filter classes may impose different requirements on the related Topic, the expression, or the parameters. The default filter class, in particular, requires that a TypeObject for the related Topic's type has been registered. When using *fastddsgen* the TypeObject registration code is generated by default.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↳DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create the Topic.
/* IDL
 *
 * struct HelloWorld
 * {
 *     long index;
 *     string message;
 * }
 *
 */
Topic* topic =
    participant->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
```

(continues on next page)

(continued from previous page)

```

    return;
}

// Create a ContentFilteredTopic using an expression with no parameters
std::string expression = "message like 'Hello*'";
std::vector<std::string> parameters;
ContentFilteredTopic* filter_topic =
    participant->create_contentfilteredtopic("HelloWorldFilteredTopic1", topic,
    ↪expression, parameters);
if (nullptr == filter_topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic using an expression with parameters
expression = "message like %0 or index > %1";
parameters.push_back("*world*");
parameters.push_back("20");
ContentFilteredTopic* filter_topic_with_parameters =
    participant->create_contentfilteredtopic("HelloWorldFilteredTopic2", topic,
    ↪expression, parameters);
if (nullptr == filter_topic_with_parameters)
{
    // Error
    return;
}

// The ContentFilteredTopic instances can then be used to create DataReader objects.
Subscriber* subscriber =
    participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}

DataReader* reader_on_filter = subscriber->create_datareader(filter_topic, DATAREADER_
    ↪QOS_DEFAULT);
if (nullptr == reader_on_filter)
{
    // Error
    return;
}

DataReader* reader_on_filter_with_parameters =
    subscriber->create_datareader(filter_topic_with_parameters, DATAREADER_QOS_
    ↪DEFAULT);
if (nullptr == reader_on_filter_with_parameters)
{
    // Error
    return;
}

```

(continues on next page)

(continued from previous page)

}

Updating the filter expression and parameters

A ContentFilteredTopic provides several member functions for the management of the filter expression and the expression parameters:

- The filter expression can be retrieved with the `get_filter_expression()` member function.
- The expression parameters can be retrieved with the `get_expression_parameters()` member function.
- The expression parameters can be modified using the `set_expression_parameters()` member function. The same constraints as when creating a ContentFilteredTopic apply.
- The filter expression can be modified along with the expression parameters using the `set_filter_expression()` member function.

```
// This lambda prints all the information of a ContentFilteredTopic
auto print_filter_info = [](
    const ContentFilteredTopic* filter_topic)
{
    std::cout << "ContentFilteredTopic info for '" << filter_topic->get_name() <
    << "':" << std::endl;
    std::cout << "  - Related Topic: " << filter_topic->get_related_topic()->get_
    name() << std::endl;
    std::cout << "  - Expression:      " << filter_topic->get_filter_expression() <
    << std::endl;
    std::cout << "  - Parameters:" << std::endl;

    std::vector<std::string> parameters;
    filter_topic->get_expression_parameters(parameters);
    size_t i = 0;
    for (const std::string& parameter : parameters)
    {
        std::cout << "      " << i++ << ": " << parameter << std::endl;
    }
};

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic
/* IDL
 *
 * struct HelloWorld
 * {
```

(continues on next page)

(continued from previous page)

```

*     long index;
*     string message;
* }
*
*/
Topic* topic =
    participant->create_topic("HelloWorldTopic", "HelloWorldTopic", TOPIC_QOS_
↪DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic
ContentFilteredTopic* filter_topic =
    participant->create_contentfilteredtopic("HelloWorldFilteredTopic", topic,
↪"index > 10", {});
if (nullptr == filter_topic)
{
    // Error
    return;
}

// Print the information
print_filter_info(filter_topic);

// Use the ContentFilteredTopic on DataReader objects.
// (...)

// Update the expression
if (RETCODE_OK !=
    filter_topic->set_filter_expression("message like %0 or index > %1", {"'Hello*'",
↪ "15"}))
{
    // Error
    return;
}

// Print the updated information
print_filter_info(filter_topic);

// Update the parameters
if (RETCODE_OK !=
    filter_topic->set_expression_parameters({"'*world*'", "222"}))
{
    // Error
    return;
}

// Print the updated information
print_filter_info(filter_topic);

```

Deleting a ContentFilteredTopic

A ContentFilteredTopic can be deleted with the `delete_contentfilteredtopic()` member function on the DomainParticipant instance where the ContentFilteredTopic was created.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create a Topic
/* IDL
 *
 * struct HelloWorld
 * {
 *     long index;
 *     string message;
 * }
 */
Topic* topic =
    participant->create_topic("HelloWorldTopic", "HelloWorldTopic", TOPIC_QOS_
    ↪DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic
ContentFilteredTopic* filter_topic =
    participant->create_contentfilteredtopic("HelloWorldFilteredTopic", topic,
    ↪"index > 10", {});
if (nullptr == filter_topic)
{
    // Error
    return;
}

// Use the ContentFilteredTopic on DataReader objects.
// (...)

// Delete the ContentFilteredTopic
if (RETCODE_OK != participant->delete_contentfilteredtopic(filter_topic))
{
    // Error
    return;
}
```

The default SQL-like filter

Filter expressions used by *ContentFilteredTopic* API may use a subset of SQL syntax, extended with the possibility to use program variables in the SQL expression. This section shows this default SQL-like syntax and how to use it.

- *Grammar*
- *Like condition*
- *Match condition*
- *Type comparisons*
- *Example*

Grammar

The allowed SQL expressions are defined with the BNF-grammar below.

The following conventions are made:

- “Terminals” are quoted.
- TOKENS are typeset in code block with black font color.

```

Expression      ::=  FilterExpression
FilterExpression ::=  Condition
Condition       ::=  Predicate |
                    Condition "AND" Condition |
                    Condition "OR" Condition |
                    "NOT" Condition |
                    "(" Condition ")"
Predicate       ::=  ComparisonPredicate |
                    BetweenPredicate
ComparisonPredicate ::=  FIELDNAME RelOp Parameter |
                        Parameter RelOp FIELDNAME |
                        FIELDNAME RelOp FIELDNAME
BetweenPredicate  ::=  FIELDNAME "BETWEEN" Range |
                        FIELDNAME "NOT BETWEEN" Range
RelOp             ::=  "=" | ">" | ">=" | "<" | "<=" |
                        "<>" | "!=" | like | match
Range            ::=  Parameter "AND" Parameter
Parameter        ::=  BOOLEANVALUE |
                        INTEGERVALUE |
                        CHARVALUE |
                        FLOATVALUE |
                        STRINGVALUE |
                        ENUMERATEDVALUE |
                        PARAMETER

```

“Terminals” and TOKENS are case sensitive but both uppercase and lowercase are supported.

The syntax and meaning of the tokens used in the SQL grammar is described as follows:

- **FIELDNAME**: is a reference to a field in the data-structure. The dot `.` is used to navigate through nested structures. The number of dots that may be used in a FIELDNAME is unlimited. The FIELDNAME can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the

corresponding structure.

```
FIELDNAME      ::=  FieldNamePart ( "." FieldNamePart )*
FieldNamePart  ::=  Identifier ( "[" Integer "]" )?
```

An example of FIELDNAMES:

Filter expression

```
"points[0] = 0 AND color.red < 100"
```

Associated IDL

```
struct Color
{
    octet red;
    octet green;
    octet blue;
};

struct Shape
{
    long points[4];
    Color color;
};
```

- **BOOLEANVALUE**: Can either be *true* or *false*, case sensitive.

```
BOOLEANVALUE ::=  ["TRUE", "true", "FALSE", "false"]
```

- **INTEGERVALUE**: Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. A hexadecimal number is preceded by `0x` and must be a valid hexadecimal expression.

```
INTEGERVALUE ::=  ([ "+", "-" ])? Integer
Integer      ::=  ([ "0"-"9" ])+ | [ "0x", "0X" ] ([ "0"-"9", "A"-"F", "a"-"f" ])+
```

An example of INTEGERVALUE:

```
value = -10
```

- **CHARVALUE**: A single character enclosed between single quotes.

```
CHARVALUE ::=  "'" Character "'"
Character  ::=  ~[ "\n" ]
```

An example of CHARVALUE:

```
value = 'c'
```

- **FLOATVALUE**: Any series of digits, optionally preceded by a plus or minus sign and optionally including

a floating point (.). A power-of-ten expression may be postfixed, which has the syntax `e:sup:n`, where `n` is a number, optionally preceded by a plus or minus sign.

```

FLOATVALUE ::=  ([ "+", "-" ])? (Integer Exponent | Integer Fractional | Integer Fractional Exponent)
Fractional  ::=  "." Integer
Exponent    ::=  ["e", "E"] ([ "+", "-" ])? Integer

```

An example of `FLOATVALUE`:

```
value = 10.1e-10
```

- **STRINGVALUE**: Any series of characters encapsulated in single quotes, except a new-line character or a right quote. A string starts with a left or right quote, but ends with a right quote.

```
STRINGVALUE ::=  [ "'" ] ~[ "'", "\r", "\n" ] [ "'" ]
```

An example of `STRINGVALUE`:

```
value = 'This is a string'
```

- **ENUMERATEDVALUE**: An enumerated value is a reference to a value declared within an enumeration. Enumerated values consist of the name of the enumeration label enclosed in single quotes. The name used for the enumeration label must correspond to the label names specified in the IDL definition of the enumeration.

```
ENUMERATEDVALUE ::=  [ "'" ] ~[ "'", "\r", "\n" ] [ "'" ]
```

An example of `ENUMERATEDVALUE`:

Filter expression

```
value = 'ENUM_VALUE_1'
```

Associated IDL

```

enum MyEnum
{
    ENUM_VALUE_1,
    ENUM_VALUE_2,
    ENUM_VALUE_3
};

struct Enumerators
{
    MyEnum value;
};

```

- **PARAMETER**: A parameter is of the form `%n`, where `n` represents a natural number (zero included) smaller than 100. It refers to the `n + 1` th argument in the given context.

```
PARAMETER ::=  [ "%" ] [ "0"-"9" ] ( [ "0"-"9" ] )?
```

An example of PARAMETER:

```
value = %1
```

Like condition

The `like` operator is similar as the one defined by SQL. This operator can only be used with strings. There are two wildcards that could be used in conjunction with this operator

- The percent sign % (or its alias *) represents zero, one, or multiple characters.
- The underscore sign _ (or its alias ?) represents one single character.

All wildcards can also be used in combinations.

An example of like operator

Filter expression

```
"str like '%bird%'"
```

Associated IDL

```
struct Like
{
    string str;
};
```

where string `There are birds flying` will return true.

Match condition

The match operator performs a full-text search using a regular expression. This operator can only be used with strings. It uses the [Basic Regular Expression \(BRE\)](#) defined by POSIX.

An example of match operator

Filter expression

```
"str match '^The'"
```

Associated IDL

```
struct Like
{
    string str;
};
```

where string `There are birds flying` will return true.

Type comparisons

For the supported operators in the grammar, next table shows the type compatibility.

Operator1 Operator2	BOOLEAN	INTEGER	FLOAT	CHAR	STRING	ENUM
BOOLEAN						
INTEGER						
FLOAT						
CHAR						
STRING						
ENUM						*

(*) Only for the same enumerated type.

Example

Assuming Topic Shape has next IDL definition.

```
struct Shape
{
    long x,
    long y,
    long z,
    long width,
    long height
};
```

An example of filter expression would be:

```
"x < 23 AND y > 50 AND width BETWEEN %0 AND %1"
```

A *ContentFilteredTopic* may be created using this filter expression as explained in section *Creating a ContentFilteredTopic*.

```
ContentFilteredTopic* sql_filter_topic =
    participant->create_contentfilteredtopic("Shape", topic,
        "x < 23 AND y > 50 AND width BETWEEN %0 AND %1",
        {"10", "20"});
```

In this example parameters are used. Internally the *ContentFilteredTopic* will be created with the filter expression below, after setting the provided parameters.

```
"x < 23 AND y > 50 AND width BETWEEN 10 AND 20"
```

Using custom filters

Fast DDS API supports the creation and later registration of user's custom filters to be used in the creation of a `ContentFilteredTopic`. Required steps for using a Custom Filter are:

- *Creating the Custom Filter*
- *Creating the Factory for the Custom Filter*
- *Registering the Factory*
- *Creating a `ContentFilteredTopic` using the Custom Filter*

Creating the Custom Filter

A custom filter must be implemented by a class which inherits from `IContentFilter`. Only one function must be implemented, overriding `evaluate()`. Each time a sample is received by a `DataReader`, this function is called with next arguments.

- `payload` - The serialized payload of the sample which the custom filter has to evaluate.
- `sample_info` - The extra information which accompanies the sample.
- `reader_guid` - The GUID of the reader for which the filter is being evaluated.

The function returns a boolean where `true` implies the sample is accepted and `false` rejects the sample.

Next snippet code shows an example of Custom Filter which deserialize the `index` field from a serialized sample and rejects samples where `index > low_mark_` and `index < high_mark_`.

```
class MyCustomFilter : public IContentFilter
{
public:

    MyCustomFilter(
        int low_mark,
        int high_mark)
        : low_mark_(low_mark)
        , high_mark_(high_mark)
    {
    }

    bool evaluate(
        const SerializedPayload& payload,
        const FilterSampleInfo& sample_info,
        const GUID_t& reader_guid) const override
    {
        // Deserialize the `index` field from the serialized sample.
        /* IDL
         *
         * struct HelloWorld
         * {
         *     long index;
         *     string message;
         * }
         */
        eprosima::fastcdr::FastBuffer fastbuffer(reinterpret_cast<char*>(payload.data), 1,
```

(continues on next page)

(continued from previous page)

```

↪payload.length);
    eprosima::fastcdr::Cdr deser(fastbuffer);
    // Deserialize encapsulation.
    deser.read_encapsulation();
    int index = 0;

    // Deserialize `index` field.
    try
    {
        deser >> index;
    }
    catch (eprosima::fastcdr::exception::NotEnoughMemoryException& exception)
    {
        return false;
    }

    // Custom filter: reject samples where index > low_mark_ and index < high_mark_.
    if (index > low_mark_ && index < high_mark_)
    {
        return false;
    }

    return true;
}

private:

    int low_mark_ = 0;
    int high_mark_ = 0;

};

```

Creating the Factory for the Custom Filter

Fast DDS creates filters through a factory. Therefore a factory which provides instantiating of a Custom Filter must be implemented.

A Custom Filter's factory has to inherit from `IContentFilterFactory`. This interface requires two functions to be implemented.

Each time a Custom Filter has to be created or updated, `create_contentfilteredtopic()` calls internally `create_content_filter()` with these arguments:

- `filter_class_name` - Filter class name for which the factory is being called. It allows using the same factory for different filter classes.
- `type_name` - Type name of the topic being filtered.
- `data_type` - Type support object of the topic being filtered.
- `filter_expression` - Custom filter expression.
- `filter_parameters` - Values to set for the filter parameters (where custom filter expression has its pattern to substitute them).

- `filter_instance` - When a filter is being created, it will be `nullptr` on input, and will have the pointer to the created filter instance on output. When a filter is being updated, it will have a previously returned pointer on input.

This function should return the result of the operation.

When a Custom Filter should be removed, `delete_contentfilteredtopic()` calls internally `delete_content_filter()`. The factory must remove the provided Custom Filter's instance.

Next snippet code shows an example of Custom Filter's factory which manages instances of the Custom Filter implemented in the previous section.

```
class MyCustomFilterFactory : public IContentFilterFactory
{
public:

    ReturnCode_t create_content_filter(
        const char* filter_class_name, // My custom filter class name is 'MY_CUSTOM_
        ↪FILTER'.
        const char* type_name, // This custom filter only supports one type:
        ↪'HelloWorld'.
        const TopicDataType* /*data_type*/, // Not used in this implementation.
        const char* filter_expression, // This Custom Filter doesn't implement a
        ↪filter expression.
        const ParameterSeq& filter_parameters, // Always need two parameters to be
        ↪set: low_mark and high_mark.
        IContentFilter*& filter_instance) override
    {
        // Check the ContentFilteredTopic should be created by my factory.
        if (0 != strcmp(filter_class_name, "MY_CUSTOM_FILTER"))
        {
            return RETCODE_BAD_PARAMETER;
        }

        // Check the ContentFilteredTopic is created for the unique type this Custom
        ↪Filter supports.
        if (0 != strcmp(type_name, "HelloWorld"))
        {
            return RETCODE_BAD_PARAMETER;
        }

        // Check that the two mandatory filter parameters are set.
        if (2 != filter_parameters.length())
        {
            return RETCODE_BAD_PARAMETER;
        }

        // If there is an update, delete previous instance.
        if (nullptr != filter_instance)
        {
            delete(dynamic_cast<MyCustomFilter*>(filter_instance));
        }

        // Instantiation of the Custom Filter.
        filter_instance = new MyCustomFilter(std::stoi(filter_parameters[0]), ↪
```

(continues on next page)

(continued from previous page)

```

↪std::stoi(filter_parameters[1]));

    return RETCODE_OK;
}

ReturnCode_t delete_content_filter(
    const char* filter_class_name,
    IContentFilter* filter_instance) override
{
    // Check the ContentFilteredTopic should be created by my factory.
    if (0 != strcmp(filter_class_name, "MY_CUSTOM_FILTER"))
    {
        return RETCODE_BAD_PARAMETER;
    }

    // Deletion of the Custom Filter.
    delete(dynamic_cast<MyCustomFilter*>(filter_instance));

    return RETCODE_OK;
}
};

```

Registering the Factory

To be able to use the Custom Filter in an application, the Custom Filter's factory must be registered in the DomainParticipant. Next snippet code shows how to register a factory through API function `register_content_filter_factory()`.

```

// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create Custom Filter Factory
MyCustomFilterFactory* factory = new MyCustomFilterFactory();

// Registration of the factory
if (RETCODE_OK !=
    participant->register_content_filter_factory("MY_CUSTOM_FILTER", factory))
{
    // Error
    return;
}

```

Creating a ContentFilteredTopic using the Custom Filter

Creating a ContentFilteredTopic explains how to create a ContentFilteredTopic. In the case of using a Custom Filter, create_contentfilteredtopic() has an overload adding an argument to select the Custom Filter.

Next snippet code shows how to create a ContentFilteredTopic using the Custom Filter.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Create the Topic.
Topic* topic =
    participant->create_topic("HelloWorldTopic", "HelloWorld", TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create a ContentFilteredTopic selecting the Custom Filter and using no expression.
↪with two parameters
// Filter expression cannot be an empty one even when it is not used by the custom.
↪filter, as that effectively
// disables any filtering
std::string expression = " ";
std::vector<std::string> parameters;
parameters.push_back("10"); // Parameter for low_mark
parameters.push_back("20"); // Parameter for low_mark
ContentFilteredTopic* filter_topic =
    participant->create_contentfilteredtopic("HelloWorldFilteredTopic1", topic,
    ↪expression, parameters,
    "MY_CUSTOM_FILTER");
if (nullptr == filter_topic)
{
    // Error
    return;
}

// The ContentFilteredTopic instances can then be used to create DataReader objects.
Subscriber* subscriber =
    participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr == subscriber)
{
    // Error
    return;
}
```

(continues on next page)

(continued from previous page)

```

DataReader* reader_on_filter = subscriber->create_datareader(filter_topic, DATAREADER_
↪QOS_DEFAULT);
if (nullptr == reader_on_filter)
{
    // Error
    return;
}

```

Important: Even though this specific custom filtering example is not using the filter expression, mind that the expression cannot be an empty string as that disables filtering as explained in [Creating a ContentFilteredTopic](#).

Note: Deleting a ContentFilteredTopic which uses a Custom Filter is done exactly in the same manner explained in [Deleting a ContentFilteredTopic](#).

Where is filtering applied: writer vs reader side

Content filters may be evaluated on either side, as the DataWriter obtains the filter expression from the DataReader during discovery. Filtering on the writer side can save network bandwidth at the cost of increasing CPU usage on the writer.

Conditions for writer side filtering

A DataWriter will perform filter evaluation in the DataReader stead whenever all of the following conditions are met; filtering will otherwise be performed by the DataReader.

- The DataWriter has infinite liveliness. See [LivelinessQosPolicy](#).
- Communication with the DataReader is neither *intra-process* nor *data-sharing*.
- The DataReader is not using multicast.
- The DataWriter is filtering for no more DataReaders than the maximum value set on `reader_filters_allocation`.
 - There is a *resource-limit policy* on [DataWriterQos](#) that controls the allocation behavior of writer-side filtering resources. Setting a maximum value of 0 disables filter evaluation on the writer side. A maximum value of 32 (the default value) means the writer will perform filter evaluation for up to 32 readers.
 - If the DataWriter is evaluating filters for `writer_resource_limits.reader_filters_allocation.maximum DataReaders`, and a new filtered DataReader is created, then the filter for the newly created DataReader will be evaluated on the reader side.

Discovery race condition

On applications where the filter expression and/or the expression parameters are updated, there may be a situation where the DataWriter will apply the old version of the filter until it receives updated information through discovery. This may imply that a publication made a short time after the DataReader updated the filter, but before the updated discovery information is received by the DataWriter, may not be sent to the DataReader, even if the new filter would have told otherwise. Publications made after the updated discovery information is received will use the updated filter.

If some critical application considers this race condition issue unbearable, filtering on the writer side can be disabled by setting the maximum value on `reader_filters_allocation` to 0.

Fast DDS-Gen for data types source code generation

eProsima Fast DDS comes with a built-in source code generation tool, *Fast DDS-Gen*, which eases the process of translating an IDL specification of a data type to a functional implementation. Thus, this tool automatically generates the source code of a data type defined using IDL. A basic use of the tool is described below. To learn about all the features that *Fast DDS* offers, please refer to *Fast DDS-Gen* section.

Basic usage

Fast DDS can be executed by calling *fastddsgen* on Linux or *fastddsgen.bat* on Windows. The IDL file containing the data type definition is given with the `<IDLfile>` argument.

Linux
<pre>fastddsgen [<options>] <IDLfile> [<IDLfile> ...]</pre>
Windows
<pre>fastddsgen.bat [<options>] <IDLfile> [<IDLfile> ...]</pre>

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.
- `-no-typeobjectsupport`: It disables the automatic generation of the *TypeObject representation* registration code.
- `-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.hpp*: 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](#)).
- *MyTypeCdrAux.hpp/.ipp*: Auxiliary methods required by Fast CDR for type encoding and decoding.
- *MyTypeTypeObjectSupport.cxx/.hpp*: Auxiliary code required for *TypeObject representation* generation and registration.

7.19 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.

7.19.1 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:

<i>DDS Layer</i>	<i>RTPS Layer</i>
<i>Domain</i>	RTPSDomain
<i>DomainParticipant</i>	RTPSParticipant
<i>DataWriter</i>	RTPSWriter
<i>DataReader</i>	RTPSReader

7.19.2 How to use the RTPS Layer

We will now go over the use of the RTPS Layer like we did with the *DDS Layer* one, explaining the new features it presents.

We recommend you to look at the two examples describing how to use the RTPS layer that come with the distribution while reading this section. They are located in [examples/cpp/rtps/AsSocket](#) and [examples/cpp/rtps/Registered](#)

Managing the Participant

Creating a RTPSParticipant is done with RTPSDomain::createParticipant(). RTPSParticipantAttributes structure is used to configure the RTPSParticipant upon creation.

```
RTPSParticipantAttributes participant_attr;
participant_attr.setName("participant");
RTPSParticipant* participant = RTPSDomain::createParticipant(0, participant_attr);
```

Managing the Writers and Readers

As the RTPS standard specifies, RTPSWriters and RTPSReaders are always associated with a History element. In the *DDS Layer*, its creation and management is hidden, but in the *RTPS Layer*, you have full control over its creation and configuration.

Writers are created with RTPSDomain::createRTPSWriter() and configured with a WriterAttributes structure. They also need a WriterHistory which is configured with a HistoryAttributes structure.

```
HistoryAttributes history_attr;
WriterHistory* history = new WriterHistory(history_attr);
WriterAttributes writer_attr;
RTPSWriter* writer = RTPSDomain::createRTPSWriter(participant, writer_attr, history);
```

Similar to the creation of Writers, Readers are created with RTPSDomain::createRTPSReader() and configured with a ReaderAttributes structure. A HistoryAttributes structure is used to configure the required ReaderHistory. Note that in this case, you can provide a specialization of ReaderListener class that implements your callbacks:

```
class MyReaderListener : public ReaderListener
{
    // Callbacks override
};
MyReaderListener listener;
HistoryAttributes history_attr;
ReaderHistory* history = new ReaderHistory(history_attr);
ReaderAttributes reader_attr;
RTPSReader* reader = RTPSDomain::createRTPSReader(participant, reader_attr, history, &
↪ listener);
```

Using the History to Send and Receive Data

In the RTPS Protocol, Readers and Writers save the data about a topic in their associated Histories. Each piece of data is represented by a Change, which *eprosima Fast DDS* implements as CacheChange_t. Changes are always managed by the History.

You can add a new CacheChange_t to the History of the Writer to send data. The procedure is as follows:

1. Request a CacheChange_t from the Writer with RTPSWriter::new_change(). In order to allocate enough memory, you need to provide a callback that returns the maximum number bytes in the payload.
2. Fill the CacheChange_t with the data.
3. Add it to the History with WriterHistory::add_change().

The Writer will take care of everything to communicate the data to the Readers.


```

//Request a change from the writer
CacheChange_t* change = writer->new_change([]() -> uint32_t
{
    return 255;
}, ALIVE);
//Write serialized data into the change
change->serializedPayload.length = sprintf((char*) change->serializedPayload.data, "My_
↳example string %d", 2) + 1;
//Insert change into the history. The Writer takes care of the rest.
history->add_change(change);

```

If your topic data type has several fields, you will have to provide functions to serialize and deserialize your data in and out of the `CacheChange_t`. *Fast DDS-Gen* does this for you.

You can receive data from within the `ReaderListener::onNewCacheChangeAdded` callback, as we did in the *DDS Layer*:

1. The callback receives a `CacheChange_t` parameter containing the received data.
2. Process the data within the received `CacheChange_t`.
3. Inform the Reader's History that the change is not needed anymore.

```

class MyReaderListener : public ReaderListener
{
public:

    MyReaderListener()
    {
    }

    ~MyReaderListener()
    {
    }

    void onNewCacheChangeAdded(
        RTPSReader* reader,
        const CacheChange_t* const change)
    {
        // The incoming message is enclosed within the `change` in the function parameters
        printf("%s\n", change->serializedPayload.data);
        // Once done, remove the change
        reader->getHistory()->remove_change((CacheChange_t*)change);
    }
};

```

Managing the Builtin Transports

DDS uses the *Transport Layer* to allow communication between DDS entities. *eProsima Fast DDS* comes with five transports already implemented. However, these transports are not always exclusive between them and in some cases they can be used simultaneously.

You can choose what transports you want to use by disabling the use of builtin transports and manually adding them (see *TransportConfigQos*) or using the default builtin transports behavior and selecting one of the configuration options listed below. Each option modifies the kind of transports that will be instantiated.

Builtin Trans-ports Op-tions	Description
NONE	No transport will be instantiated. Hence, the user must manually add the desired transports. Otherwise, the participant creation will fail.
DEFAULT	UD Pv4 and SHM transports will be instantiated. SHM transport has priority over the UD Pv4 transport. Meaning that SHM will always be used when possible.
DEFAULT	UD Pv6 and SHM transports will be instantiated. SHM transport has priority over the UD Pv4 transport. Meaning that SHM will always be used when possible.
SHM	Only a SHM transport will be instantiated.
UD Pv4	Only a UD Pv4 transport will be instantiated.
UD Pv6	Only a UD Pv6 transport will be instantiated.
LARGE_D	UD Pv4, TCPv4, and SHM transports will be instantiated. However, UDP will only be used for multi-cast announcements during the participant discovery phase (see <i>Discovery phases</i>) while the participant liveliness and the application data delivery occurs over TCP or SHM. This configuration is useful when working with large data.(See <i>Large Data mode and Fast DDS over TCP</i>).

```
RTPSParticipantAttributes participant_attr;
participant_attr.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_
↪DATA);
RTPSParticipant* participant = RTPSDomain::createParticipant(0, participant_attr);
```

The same result can also be obtained using the `setup_transports()` wrapper function of the *DomainParticipantQos*, XML profiles (see *RTPS element type*) or the `FASTDDS_BUILTIN_TRANSPORTS` environment variable (see *FAST-DDS_BUILTIN_TRANSPORTS*).

Note: TCPv4 transport is initialized with the following configuration:

- `calculate_crc`, `check_crc` and `apply_security` are set to false.
- `enable_tcp_nodelay` is set to true.
- `keep_alive_thread` and `accept_thread` use the default configuration.

Warning: To obtain a better performance when working with large data messages it is extremely recommended to use the builtin transports configuration options to adjust the transport to the specific needs of the application. Please refer to *Large Data with configuration options* for more information about how to configure it.

7.19.3 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.

7.19.4 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.

7.19.5 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 `SerializedPayload` 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 from 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);
    }
};
```

(continues on next page)

(continued from previous page)

```

        return true;
    }

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

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

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

        // Assign the payload buffer to the CacheChange and update sizes
        cache_change.serializedPayload.data = payload;
        cache_change.serializedPayload.length = data.length;
        cache_change.serializedPayload.max_size = data.length;

        return true;
    }

    bool release_payload(
        CacheChange_t& cache_change) override
    {
        // Ensure precondition
        if (this != cache_change.payload_owner())
        {
            std::cerr << "Trying to release a payload buffer allocated by a different_
↪PayloadPool." << std::endl;
            return false;
        }

        // Dealloc the buffer of the payload
        delete[] cache_change.serializedPayload.data;

        // Reset sizes and pointers
        cache_change.serializedPayload.data = nullptr;
        cache_change.serializedPayload.length = 0;
        cache_change.serializedPayload.max_size = 0;

        // Reset the owner of the payload
        cache_change.payload_owner(nullptr);

        return true;
    }
};

```

(continues on next page)

(continued from previous page)

```

std::shared_ptr<CustomPayloadPool> payload_pool = std::make_shared<CustomPayloadPool>();

// A writer using the custom payload pool
HistoryAttributes writer_history_attr;
WriterHistory* writer_history = new WriterHistory(writer_history_attr);
WriterAttributes writer_attr;
RTPSWriter* writer = RTPSDomain::createRTPSWriter(participant, writer_attr, payload_pool,
↳ writer_history);

// A reader using the same instance of the custom payload pool
HistoryAttributes reader_history_attr;
ReaderHistory* reader_history = new ReaderHistory(reader_history_attr);
ReaderAttributes reader_attr;
RTPSReader* reader = RTPSDomain::createRTPSReader(participant, reader_attr, payload_pool,
↳ reader_history);

// Write and Read operations work as usual, but take the Payloads from the pool.
// 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);

```

7.20 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.

7.20.1 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, it is also possible to configure the periodicity of such announcements (see *Discovery Configuration*).
2. **Endpoint Discovery Phase (EDP):** During this phase, the *DataWriters* and *DataReaders* acknowledge each other. To do that, the *DomainParticipants* share information about their *DataWriters* and *DataReaders* with each other, using the communication channels established during the PDP. This information contains, among other

things, the `Topic` and data type (see [Topic](#)). For two endpoints to match, their topic and data type must coincide. Once `DataWriter` and `DataReader` have matched, they are ready for sending/receiving user data traffic.

Important: It is possible to use the PDP phase to transmit information about the host, user, and process (physical information) in which the `DomainParticipant` is running. Please refer to [Physical Data in Discovery Information](#) for more information about how to configure the transmitted physical data.

7.20.2 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 mechanism uses the Simple Participant Discovery Protocol (SPDP) for the PDP phase (as specified by the [RTPS standard](#)), but allows for skipping the Simple Endpoint Discovery Protocol (SEDP) phase when all the `DataWriters`' and `DataReaders`' IPs and ports, data types, and Topics are known beforehand.
- **Discovery Server:** This discovery mechanism uses a centralized discovery architecture, where a `DomainParticipant`, referred as Server, acts as a hub for meta traffic discovery.
- **Manual Discovery:** This mechanism is only compatible with the RTPS layer. It disables the PDP, letting the user to manually match and unmatch `RTPSParticipants`, `RTPSReaders`, and `RTPSWriters` using whatever external meta-information channel of its choice. Therefore, the user must access the `RTPSParticipant` implemented by the `DomainParticipant` and directly match the RTPS Entities.

7.20.3 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
<i>Discovery Protocol</i>	The discovery protocol to use (see Discovery mechanisms).	<code>DiscoveryProtocol</code>	<code>SIMPLE</code>
<i>Ignore Participant flags</i>	Filter discovery traffic for DomainParticipants in the same process, in different processes, or in different hosts.	<code>ParticipantFilter</code>	<code>NO_FILTER</code>
<i>Lease Duration</i>	Indicates for how much time should a remote <code>DomainParticipant</code> consider the local <code>DomainParticipant</code> to be alive.	<code>Duration_t</code>	20 s
<i>Announcement Period</i>	The period for the <code>DomainParticipant</code> to send PDP announcements.	<code>Duration_t</code>	3 s

Discovery Protocol

Specifies the discovery protocol to use (see *Discovery mechanisms*). The possible values are:

Dis-covery Mechanism	Pos-sible val-ues	Description
Simple	SIMPLE	Simple discovery protocol as specified in the RTPS standard .
Static	STATIC	SPDP with manual EDP specified in XML files.
Dis-covery 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_C	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 <code>sqlite</code> database. A BACKUP server can load the a database on start. This type of sever makes the Discovery Server architecture resilient to server destruction.
Manual	NONE	Disables PDP phase, therefore there is no EDP phase. All matching must be done manually through the <code>addReaderLocator</code> , <code>addReaderProxy</code> , <code>addWriterProxy</code> RTPS layer methods.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    DiscoveryProtocol_t::SIMPLE;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <participant profile_name="participant_discovery_protocol">
      <rtps>
        <builtin>
          <discovery_config>
            <discoveryProtocol>SIMPLE</discoveryProtocol>
          </discovery_config>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</dds>
```

Ignore Participant flags

Defines a filter to ignore some discovery traffic when received. This is useful to add an extra level of DomainParticipant isolation. The possible values are:

Possible values	Description
NO_FILTER	All Discovery traffic is processed.
FILTER_DIFFERENT_HOST	Discovery traffic from another host is discarded.
FILTER_DIFFERENT_PROCESS	Discovery traffic from another process on the same host is discarded.
FILTER_SAME_PROCESS	Discovery traffic from DomainParticipant's own process is discarded.
FILTER_DIFFERENT_PROCESS FILTER_SAME_PROCESS	Discovery traffic from DomainParticipant's own host is discarded.

C++

```
DomainParticipantQos pqos;  
  
pqos.wire_protocol().builtin.discovery_config.ignoreParticipantFlags =  
    static_cast<eprosima::fastrtps::rtps::ParticipantFilteringFlags_t>(  
        ParticipantFilteringFlags_t::FILTER_DIFFERENT_PROCESS |  
        ParticipantFilteringFlags_t::FILTER_SAME_PROCESS);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>  
<dds xmlns="http://www.eprosima.com">  
  <profiles>  
    <participant profile_name="participant_discovery_ignore_flags">  
      <rtps>  
        <builtin>  
          <discovery_config>  
            <ignoreParticipantFlags>FILTER_DIFFERENT_PROCESS | FILTER_SAME_  
PROCESS</ignoreParticipantFlags>  
          </discovery_config>  
        </builtin>  
      </rtps>  
    </participant>  
  </profiles>  
</dds>
```

Note: To configure a *DomainParticipant* to not receive data from its own *DataWriters*, please refer to *Ignore Local Endpoints*.

Lease Duration

Indicates for how much time should a remote DomainParticipant consider the local DomainParticipant to be alive. If the liveliness of the local DomainParticipant has not being asserted within this time, the remote DomainParticipant considers the local DomainParticipant dead and destroys all the information regarding the local DomainParticipant and all its endpoints.

The local DomainParticipant's liveliness is asserted on the remote DomainParticipant any time the remote DomainParticipant receives any kind of traffic from the local DomainParticipant.

The lease duration is specified as a time expressed in seconds and nanosecond using a `Duration_t`.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.leaseDuration = Duration_t(10, 20);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <participant profile_name="participant_discovery_lease_duration">
      <rtps>
        <builtin>
          <discovery_config>
            <leaseDuration>
              <sec>10</sec>
              <nanosec>20</nanosec>
            </leaseDuration>
          </discovery_config>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</dds>
```

Announcement Period

It specifies the periodicity of the DomainParticipant's PDP announcements. For liveliness' sake it is recommend that the announcement period is shorter than the lease duration, so that the DomainParticipant's liveliness is asserted even when there is no data traffic. It is important to note that there is a trade-off involved in the setting of the announcement period, i.e. too frequent announcements will bloat the network with meta traffic, but too scarce ones will delay the discovery of late joiners.

DomainParticipant's announcement period is specified as a time expressed in seconds and nanosecond using a `Duration_t`.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.leaseDuration_announcementperiod =
↳ Duration_t(1, 2);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <participant profile_name="participant_discovery_lease_announcement">
      <rtps>
        <builtin>
          <discovery_config>
            <leaseAnnouncement>
              <sec>1</sec>
              <nanosec>2</nanosec>
            </leaseAnnouncement>
          </discovery_config>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</dds>
```

SIMPLE Discovery Settings

The SIMPLE discovery protocol resolves the establishment of the end-to-end connection between various DDS Entities. *eProsima Fast DDS* implements the SIMPLE discovery protocol to provide compatibility with the [RTPS standard](#). The specification splits up the SIMPLE discovery protocol into two independent protocols:

- **Simple Participant Discovery Protocol (SPDP):** specifies how DomainParticipants discover each other in the network; it announces and detects the presence of DomainParticipants within the same domain.
- **Simple Endpoint Discovery Protocol (SEDP):** defines the protocol adopted by the discovered DomainParticipants for the exchange of information in order to discover the DDS Entities contained in each of them, i.e. the *DataWriter* and *DataReader*.

Name	Description
<i>Initial Announcements</i>	It defines the behavior of the DomainParticipants initial announcements.
<i>Simple EDP Attributes</i>	It defines the use of the SIMPLE protocol as a discovery protocol.
<i>Initial peers</i>	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, 1000000000u);
```

XML

```
<participant profile_name="participant_profile_simple_discovery">
  <rtps>
    <builtin>
      <discovery_config>
        <initialAnnouncements>
          <count>5</count>
          <period>
            <nanosec>1000000000</nanosec>
          </period>
        </initialAnnouncements>
      </discovery_config>
    </builtin>
  </rtps>
</participant>
```

Simple EDP Attributes

Name	Description	Type	De- fault
SIMPLE EDP	It defines the use of the SIMPLE protocol as a discovery protocol for the EDP phase. A DomainParticipant may create DataWriters, DataReaders, both or neither.	bool	true
Publication writer and Subscription reader	It is intended for DomainParticipants that implement only one or more DataWriters, i.e. do not implement DataReaders. It allows the creation of only DataReader discovery related EDP endpoints.	bool	true
Publication reader and Subscription writer	It is intended for DomainParticipants that implement only one or more DataReaders, i.e. do not implement DataWriters. It allows the creation of only DataWriter discovery related EDP endpoints.	bool	true

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol =
    true;
pqos.wire_protocol().builtin.discovery_config.m_simpleEDP.use_
    PublicationWriterANDSubscriptionReader = true;
pqos.wire_protocol().builtin.discovery_config.m_simpleEDP.use_
    PublicationReaderANDSubscriptionWriter = false;
```

XML

```
<participant profile_name="participant_profile_qos_discovery_edp">
  <rtps>
    <builtin>
      <discovery_config>
        <EDP>SIMPLE</EDP>
        <simpleEDP>
          <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
          <PUBREADER_SUBWRITER>false</PUBREADER_SUBWRITER>
        </simpleEDP>
      </discovery_config>
    </builtin>
  </rtps>
</participant>
```

Initial peers

According to the [RTPS standard](#) (Section 9.6.1.1), each RTPSParticipant must listen for incoming Participant Discovery Protocol (PDP) discovery metatraffic in two different ports, one linked to a multicast address and another one linked to a unicast address. *Fast DDS* allows for the configuration of an initial peers list which contains one or more such IP-port address pairs corresponding to remote DomainParticipants PDP discovery listening resources, so that the local DomainParticipant will not only send its PDP traffic to the default multicast address-port specified by its domain, but also to all the IP-port address pairs specified in the initial peers list.

A DomainParticipant's initial peers list contains the list of IP-port address pairs of all other DomainParticipants with which it will communicate. It is a list of addresses that a DomainParticipant will use in the unicast discovery mechanism, together or as an alternative to multicast discovery. Therefore, this approach also applies to those scenarios in which multicast functionality is not available.

According to the [RTPS standard](#) (Section 9.6.1.1), the RTPSParticipants' discovery traffic unicast listening ports are calculated using the following equation: $7400 + 250 * domainID + 10 + 2 * participantID$. Thus, if for example a RTPSParticipant operates in Domain 0 (default domain) and its ID is 1, its discovery traffic unicast listening port would be: $7400 + 250 * 0 + 10 + 2 * 1 = 7412$. By default *eProsima Fast DDS* uses as initial peers the Metatraffic Multicast Locators.

The following constitutes an example configuring an Initial Peers list with one peer on host 192.168.10.13 with DomainParticipant ID 1 in domain 0.

Note: There is also the possibility of not defining the initial peer port. In this case, the discovery information would be sent to every port ranging from *participantID* zero to the *maxInitialPeersRange* value set in the *TransportDe-*

scriptorInterface. Consequently, setting this value to at least the maximum expected number of DomainParticipants will ensure discovery and communication.

C++

```
DomainParticipantQos qos;

// configure an initial peer on host 192.168.10.13.
// The port number corresponds to the well-known port for metatraffic unicast
// on participant ID `1` and domain `0`.
Locator_t initial_peer;
IPLocator::setIPv4(initial_peer, "192.168.10.13");
initial_peer.port = 7412;
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer);
```

XML

```
<!--
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
-->
  <participant profile_name="initial_peers_example_profile" is_default_profile="true
  <-->
    <rtps>
      <builtin>
        <initialPeersList>
          <locator>
            <udp4>
              <address>192.168.10.13</address>
              <port>7412</port>
            </udp4>
          </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
<i>STATIC EDP</i>	It activates the STATIC discovery protocol.
<i>STATIC EDP XML Configuration Specification</i>	Specifies an XML content with a description of the remote DataWriters and DataReaders.
<i>Initial Announcements</i>	It defines the behavior of the DomainParticipant initial announcements (PDP phase).

STATIC EDP

To activate the STATIC EDP, the SEDP must be disabled on the `WireProtocolConfigQos`. This can be done either by code or using an XML configuration file:

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol =
    false;
pqos.wire_protocol().builtin.discovery_config.use_STATIC_EndpointDiscoveryProtocol =
    true;
```

XML

```
<participant profile_name="participant_profile_static_edp">
  <rtps>
    <builtin>
      <discovery_config>
        <EDP>STATIC</EDP>
      </discovery_config>
    </builtin>
  </rtps>
</participant>
```

Currently two different formats of exchanging information in the Participant Discovery Phase (PDP) are supported: the default one and another that reduces the network bandwidth used. *Static Discovery's Exchange Format* explains how to change this.

STATIC EDP XML Configuration Specification

Since activating STATIC EDP suppresses all EDP meta traffic, the information about the remote entities (DataWriters and DataReaders) must be statically specified, which is done using dedicated XML files. A *DomainParticipant* may load several of such configuration files so that the information about different entities can be contained in one file, or split into different files to keep it more organized. *Fast DDS* provides a *Static Discovery example* that implements this EDP discovery protocol.

The following table describes all the possible elements of a STATIC EDP XML configuration file. A full example of such file can be found in *STATIC EDP XML Example*.

Name	Description	Values	Default
<userId>	Mandatory. Uniquely identifies the DataReader/DataWriter.	uint16_t	0
<entityId>	EntityId of the DataReader/DataWriter.	uint16_t	0
<expectsInline>	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
<partitionNumber>	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 .		
<reliability>	See the ReliabilityQosPolicy section.	BEST_EFFORT_RELIABILITY_QOS RELIABLE_RELIABILITY_QOS	BEST_EFFORT_RELIABILITY_QOS
<durability>	See the DurabilityQosPolicy section.	VOLATILE_DURABILITY_QOS TRANSIENT_LOCAL_DURABILITY_QOS TRANSIENT_DURABILITY_QOS	VOLATILE_DURABILITY_QOS
<ownership>	See Ownership QoS .		
<liveliness>	Defines the liveliness of the remote peer. See Liveliness QoS .		
<disablePositiveAcks>	See DisablePositiveACKsQosPolicy .	See DisablePositiveAcks	

Locators definition

Locators for remote peers are configured using <unicastLocator> and <multicastLocator> tags. These take no value, and the locators are defined using tag elements. Locators defined with <unicastLocator> and <multicastLocator> are accumulative, so they can be repeated to assign several remote endpoints locators to the same peer.

- **address**: a mandatory string representing the locator address.
- **port**: an optional uint16_t representing a port on that address.

Ownership QoS

The ownership of the topic can be configured using <ownershipQos> tag. It takes no value, and the configuration is done using tag elements:

- **kind**: can be one of SHARED_OWNERSHIP_QOS or EXCLUSIVE_OWNERSHIP_QOS. This element is mandatory withing the tag.
- **strength**: an optional uint32_t specifying how strongly the remote DomainParticipant owns the *Topic*. This QoS can be set on DataWriters **only**. If not specified, default value is zero.

Liveliness QoS

The *LivelinessQosPolicy* of the remote peer is configured using <livelinessQos> tag. It takes no value, and the configuration is done using tag elements:

- **kind**: can be any of `AUTOMATIC_LIVELINESS_QOS`, `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS` or `MANUAL_BY_TOPIC_LIVELINESS_QOS`. This element is mandatory withing the tag.
- **leaseDuration_ms**: an optional `uint32` specifying the lease duration for the remote peer. The special value `INF` can be used to indicate infinite lease duration. If not specified, default value is `INF`

Checking STATIC EDP XML Files

Before loading a static EDP XML file, it would be useful to check its validity and make sure the file will be successfully loaded. This verification can be performed on *DomainParticipantFactory* using `DomainParticipantFactory::check_xml_static_discovery()`, using either XML files or the configuration directly, as in the examples below.

File

```
// The (file://) flag is optional.
std::string file = "file://static_Discovery.xml";
DomainParticipantFactory* factory = DomainParticipantFactory::get_instance();
if (RETCODE_OK != factory->check_xml_static_discovery(file))
{
    std::cout << "Error parsing xml file " << file << std::endl;
}
```

Data

```
// The (data://) flag is required to load the configuration directly.
std::string fileData = "data://<?xml version=\"1.0\" encoding=\"utf-8\"?>\" \
    "<staticdiscovery>" \
    "<participant>" \
    "<name>HelloWorldPublisher</name>" \
    "<writer>" \
    "<userId>1</userId>" \
    "<entityID>2</entityID>" \
    "<topicName>HelloWorldTopic</topicName>" \
    "<topicDataType>HelloWorld</topicDataType>" \
    "</writer>" \
    "</participant>" \
    "</staticdiscovery>";
if (RETCODE_OK != factory->check_xml_static_discovery(fileData))
{
    std::cout << "Error parsing xml file data:" << std::endl << fileData << std::endl;
}
```

STATIC EDP XML Example

The following is a complete example of a configuration XML file for two remote DomainParticipant, a DataWriter and a DataReader. This configuration **must** agree with the configuration used to create the remote DataReader/DataWriter. Otherwise, communication between DataReaders and DataWriters may be affected. If any non-mandatory element is missing, it will take the default value. As a rule of thumb, all the elements that were specified on the remote DataReader/DataWriter creation should be configured.

XML

```

<staticdiscovery>
  <participant>
    <name>HelloWorldSubscriber</name>
    <reader>
      <userId>3</userId>
      <entityID>4</entityID>
      <expectsInlineQos>true</expectsInlineQos>
      <topicName>HelloWorldTopic</topicName>
      <topicDataType>HelloWorld</topicDataType>
      <topicKind>WITH_KEY</topicKind>
      <partitionQos>HelloPartition</partitionQos>
      <partitionQos>WorldPartition</partitionQos>
      <unicastLocator address="192.168.0.128" port="5000"/>
      <unicastLocator address="10.47.8.30" port="6000"/>
      <multicastLocator address="239.255.1.1" port="7000"/>
      <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>
      <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
      <ownershipQos kind="SHARED_OWNERSHIP_QOS"/>
      <livelinessQos kind="AUTOMATIC_LIVELINESS_QOS" leaseDuration_ms="1000"/>
      <disablePositiveAcks>
        <enabled>true</enabled>
      </disablePositiveAcks>
    </reader>
  </participant>
  <participant>
    <name>HelloWorldPublisher</name>
    <writer>
      <unicastLocator address="192.168.0.120" port="9000"/>
      <unicastLocator address="10.47.8.31" port="8000"/>
      <multicastLocator address="239.255.1.1" port="7000"/>
      <userId>5</userId>
      <entityID>6</entityID>
      <topicName>HelloWorldTopic</topicName>
      <topicDataType>HelloWorld</topicDataType>
      <topicKind>WITH_KEY</topicKind>
      <partitionQos>HelloPartition</partitionQos>
      <partitionQos>WorldPartition</partitionQos>
      <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>
      <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
      <ownershipQos kind="SHARED_OWNERSHIP_QOS" strength="50"/>
      <livelinessQos kind="AUTOMATIC_LIVELINESS_QOS" leaseDuration_ms="1000"/>
      <disablePositiveAcks>
        <enabled>true</enabled>
        <duration>
          <sec>300</sec>
        </duration>
      </disablePositiveAcks>
    </writer>
  </participant>
</staticdiscovery>

```

Loading STATIC EDP XML Files

Statically discovered remote DataReaders/DataWriters **must** define a unique *userID* on their profile, whose value **must** agree with the one specified in the discovery configuration XML. This is done by setting the user ID on the *DataReaderQos/DataWriterQos*:

C++

```
// Configure the DataWriter
DataWriterQos wqos;
wqos.endpoint().user_defined_id = 1;

// Configure the DataReader
DataReaderQos rqos;
rqos.endpoint().user_defined_id = 3;
```

XML

```
<data_writer profile_name="writer_xml_conf_static_discovery">
  <userDefinedID>3</userDefinedID>
</data_writer>

<data_reader profile_name="reader_xml_conf_static_discovery">
  <userDefinedID>5</userDefinedID>
</data_reader>
```

On the local DomainParticipant, you can load STATIC EDP configuration content specifying the file containing it.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.static_edp_xml_config("file://
↪RemotePublisher.xml");
pqos.wire_protocol().builtin.discovery_config.static_edp_xml_config("file://
↪RemoteSubscriber.xml");
```

XML

```
<participant profile_name="participant_profile_static_load_xml">
  <rtps>
    <builtin>
      <discovery_config>
        <static_edp_xml_config>file://RemotePublisher.xml</static_edp_xml_
↪config>
        <static_edp_xml_config>file://RemoteSubscriber.xml</static_edp_xml_
↪config>
      </discovery_config>
    </builtin>
  </rtps>
</participant>
```

Or you can specify the STATIC EDP configuration content directly.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.static_edp_xml_config(
    "data://<?xml version=\"1.0\" encoding=\"utf-8\"?>\" \
    "<staticdiscovery><participant><name>RTPSParticipant</name></participant></\
    ↪staticdiscovery>");
```

Discovery Server Settings

This mechanism is based on a client-server discovery paradigm, i.e. the metatraffic (message exchange among *Domain-Participants* to identify each other) is managed by one or several server DomainParticipants (left figure), as opposed to simple discovery (right figure), where metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol. A *Discovery-Server* tool is available to ease Discovery Server setup and testing.

Note: *DDS Domain* concept does not apply when enabling the Discovery Server mechanism.

- *Key concepts*
- *Choosing between Client and Server*
- *The GuidPrefix as the server unique identifier*
- *The server locator list*
- *Fine tuning discovery server handshake*
- *Modifying remote servers list at run time*
- *Configure Discovery Server locators using names*
- *Full example*
- *Security*

Fig. 9: Comparison of Discovery Server and Simple discovery mechanisms

Key concepts

In this architecture there are several key concepts to understand:

- The Discovery Server mechanism reuses the RTPS discovery messages structure, as well as the standard DDS *DataWriters* and *DataReaders*.
- Discovery Server DomainParticipants may be *clients* or *servers*. The only difference between them is how they handle discovery traffic. The user traffic, that is, the traffic among the DataWriters and DataReaders they create, is role-independent.

- All *server* and *client* discovery information will be shared with linked *clients*. Note that a *server* may act as a *client* for other *servers*.
- A **SERVER** is a participant to which the *clients* (and maybe other *servers*) send their discovery information. The role of the *server* is to re-distribute the *clients* (and *servers*) discovery information to their known *clients* and *servers*. A *server* may connect to other *servers* to receive information about their *clients*. Known *servers* will receive all the information known by the *server*. Known *clients* will only receive the information they need to establish communication, i.e. the information about the DomainParticipants, DataWriters, and DataReaders to which they match. This means that the *server* runs a “matching” algorithm to sort out which information is required by which *client*.
- A **BACKUP server** is a *server* that persists its discovery database into a file. This type of *server* can load the network graph from a file on start-up without the need of receiving any *client*'s information. It can be used to persist the *server* knowledge about the network between runs, thus securing the *server*'s information in case of unexpected shutdowns. It is important to note that the discovery times will be negatively affected when using this type of *server*, since periodically writing to a file is an expensive operation.
- A **CLIENT** is a participant that connects to one or more *servers* from which it receives only the discovery information they require to establish communication with matching endpoints.
- *Clients* require prior knowledge of the *servers* to which they want to link. Basically it is reduced to the *servers* identity (henceforth called `GuidPrefix_t`) and a list of locators where the *servers* are listening. These locators also define the transport protocol (UDP or TCP) the client will use to contact the *server*.
 - The `GuidPrefix_t` is the RTPS standard RTPSParticipant unique identifier, a 12-byte chain. This identifier allows *clients* to assess whether they are receiving messages from the right *server*, as each standard RTPS message contains this piece of information.

The `GuidPrefix_t` is used because the *server*'s IP address may not be a reliable enough server identifier, since several *servers* can be hosted in the same machine, thus having the same IP, and also because multicast addresses are acceptable addresses.
- A **SUPER_CLIENT** is a *client* that receives the discovery information known by the *server*, in opposition to *clients*, which only receive the information they need.

Note: A **SUPER_CLIENT** does not behave as a *Server* as it only receives the discovery information through the *Server* to which it is connected. Any DomainParticipant discovered by the *Server* with no endpoints will not be known by the **SUPER_CLIENT**.

- *Servers* do not require any prior knowledge of their *clients*, but their `GuidPrefix_t` and locator list (where they are listening) must match the one provided to the *clients*. *Clients* send discovery messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement. From then on, the *server* knows about the *client* and will inform it of the relevant discovery information. The same principle applies to a *server* connecting to another *server*.

Choosing between Client and Server

It is set by the *Discovery Protocol* general setting. A participant can only play one role (despite the fact that a *server* may connect to other *servers*). It is mandatory to fill this value because it defaults to **SIMPLE**. The examples below shows how to set this parameter both programmatically and using XML.

C++

```
DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    DiscoveryProtocol_t::CLIENT;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    DiscoveryProtocol_t::SUPER_CLIENT;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    DiscoveryProtocol_t::SERVER;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    DiscoveryProtocol_t::BACKUP;
```

XML

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

```
DomainParticipantQos serverQos;
std::stringstream("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">
  <participant profile_name="participant_server_guidprefix" >
    <rtps>
      <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
    </rtps>
  </participant>
</profiles>
```

Note that a *server* can connect to other *servers*. Thus, the following section may also apply.

Important: When selecting a GUID prefix for the *server*, it is important to take into account that Fast DDS also uses this parameter to identify participants in the same process and enable intra-process communications. Setting two DomainParticipant GUID prefixes as intra-process compatible will result in no communication if the DomainParticipants run in separate processes. For more information, please refer to [GUID Prefix considerations for intra-process delivery](#).

Warning: Launching more than one server using the same GUID prefix is undefined behavior.

Client side setup

Each *client* must keep a list of the *servers* to which it wants to link. Each single element represents an individual server, and a `GuidPrefix_t` must be provided. The *server* list must be populated with `RemoteServerAttributes` objects with a valid `GuidPrefix_t` data member. In XML the server list and its elements are simultaneously specified. Note that `prefix` is an element of the `RemoteServer` tag.

C++

```
RemoteServerAttributes server;
server.ReadguidPrefix("44.53.00.5f.45.50.52.4f.53.49.4d.41");

DomainParticipantQos clientQos;
clientQos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
    back(server);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="participant_profile_discovery_client_prefix">
    <rtps>
      <builtin>
        <discovery_config>
          <discoveryServersList>
            <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
              <!-- Metatraffic locators -->
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

The server locator list

Each *server* must specify valid locators where it can be reached. Any *client* must be given proper locators to reach each of its *servers*. As in the [above section](#), here there is a *server* and a *client* side setup.

Server side setup

The examples below show how to setup the server locator list and XML tag.

C++

```
Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 1, 133);
locator.port = 64863;

DomainParticipantQos serverQos;
serverQos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="participant_profile_discovery_server_server_metatraffic
  <--"
    <rtps>
      <builtin>
        <metatrafficUnicastLocatorList>
          <locator>
            <udp4>
              <!-- placeholder server UDP address -->
              <address>192.168.1.113</address>
              <port>64863</port>
            </udp4>
          </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">
  <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>
                  <udp4>
                    <!-- placeholder server UDP address -->
                    <address>192.168.1.113</address>
                    <port>64863</port>
                  </udp4>
                </locator>
              </metatrafficUnicastLocatorList>
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

Fine tuning discovery server handshake

As explained [above](#) the *clients* send discovery messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement. Mind that this period also applies for those *servers* which connect to other *servers*. The default value for this period is 450 ms.

C++

```
DomainParticipantQos participant_qos;
participant_qos.wire_protocol().builtin.discovery_config.discoveryServer_client_
    syncperiod =
        Duration_t(0, 2500000000);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="participant_profile_ping" >
    <rtps>
      <builtin>
        <discovery_config>
          <clientAnnouncementPeriod>
            <!-- change default to 250 ms -->
            <nanosec>2500000000</nanosec>
          </clientAnnouncementPeriod>
        </discovery_config>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

Modifying remote servers list at run time

Once a *server* or *client* is running, it is possible to programmatically modify the participant's list of remote *servers* to which the running *server* or *client* should connect. This is done by calling `DomainParticipant::set_qos()` with a `DomainParticipantQos` which has a modified `WireProtocolConfigQos` (see [WireProtocolConfigQos](#)). This feature allows to include a new remote server into the Discovery Server network or modify the remote server locator in case that the remote server is relaunched with a different listening locator.

Important: The list of remote *servers* can only be modified to either add more *servers*, or modify the remote server locator, but not to remove any of the existing ones. This means that the new list passed to `DomainParticipant::set_qos()` must be a superset of the existing one.

Note: The remote server list can also be modified using the `ROS_DISCOVERY_SERVER` environment variable. Please refer to [FASTDDS_ENVIRONMENT_FILE](#) for more information.

Warning: It is strongly advised to use either the API or the environment file. Using both at the same time may cause undefined behavior.

C++

```
// Get existing QoS for the server or client
DomainParticipantQos client_or_server_qos;
client_or_server->get_qos(client_or_server_qos);

/* Create a new server entry to which the client or server should connect */
RemoteServerAttributes remote_server_att;

// Set server's GUID prefix
remote_server_att.ReadguidPrefix("44.53.00.5f.45.50.52.4f.53.49.4d.42");

// Set server's listening locator for PDP
Locator_t locator;
IPLocator::setIPv4(locator, 127, 0, 0, 1);
locator.port = 11812;
remote_server_att.metatrafficUnicastLocatorList.push_back(locator);

/* Update list of remote servers for this client or server */
client_or_server_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
->back(remote_server_att);
if (RETCODE_OK != client_or_server->set_qos(client_or_server_qos))
{
    // Error
    return;
}
```

Configure Discovery Server locators using names

All the examples provided in *Discovery Server Settings* use IPv4 addresses to specify the servers' listening locators. However, *Fast DDS* also allows to *specify locator addresses using names*.

Full example

The following constitutes a full example on how to configure *server* and *client* both programmatically and using XML. You may also have a look at the *eProsima Fast DDS* Github repository, which contains [an example](#) similar to the one discussed in this section, as well as multiple other examples for different use cases.

Server side setup

C++

```
// Get default participant QoS
DomainParticipantQos server_qos = PARTICIPANT_QOS_DEFAULT;

// Set participant as SERVER
server_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    DiscoveryProtocol_t::SERVER;

// Set SERVER's GUID prefix
std::istringstream("44.53.00.5f.45.50.52.4f.53.49.4d.41") >> server_qos.wire_
    protocol().prefix;

// Set SERVER's listening locator for PDP
Locator_t locator;
IPLocator::setIPv4(locator, 127, 0, 0, 1);
locator.port = 11811;
server_qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);

/* Add a remote serve to which this server will connect */
// Set remote SERVER's GUID prefix
RemoteServerAttributes remote_server_att;
remote_server_att.ReadguidPrefix("44.53.01.5f.45.50.52.4f.53.49.4d.41");

// Set remote SERVER's listening locator for PDP
Locator_t remote_locator;
IPLocator::setIPv4(remote_locator, 127, 0, 0, 1);
remote_locator.port = 11812;
remote_server_att.metatrafficUnicastLocatorList.push_back(remote_locator);

// Add remote SERVER to SERVER's list of SERVERs
server_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
    back(remote_server_att);

// Create SERVER
DomainParticipant* server =
    DomainParticipantFactory::get_instance()->create_participant(0, server_qos);
if (nullptr == server)
{
    // Error
    return;
}
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="participant_profile_server_full_example">
    <rtsp>
      <!-- Set SERVER's GUID prefix -->
      <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
      <builtin>
```

300

```
<!-- Set participant as SERVER -->
```

```
<discovery_config>
  <discoveryProtocol>SERVER</discoveryProtocol>
  <!--
```

```
    Set a list of remote servers to which this server connects.
```


Client side setup

C++

```
// Get default participant QoS
DomainParticipantQos client_qos = PARTICIPANT_QOS_DEFAULT;

// Set participant as CLIENT
client_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    DiscoveryProtocol_t::CLIENT;

// Set SERVER's GUID prefix
RemoteServerAttributes remote_server_att;
remote_server_att.ReadguidPrefix("44.53.00.5f.45.50.52.4f.53.49.4d.41");

// Set SERVER's listening locator for PDP
Locator_t locator;
IPLocator::setIPv4(locator, 127, 0, 0, 1);
locator.port = 11811;
remote_server_att.metatrafficUnicastLocatorList.push_back(locator);

// Add remote SERVER to CLIENT's list of SERVERs
client_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
    back(remote_server_att);

// Set ping period to 250 ms
client_qos.wire_protocol().builtin.discovery_config.discoveryServer_client_syncperiod =
    Duration_t(0, 250000000);

// Create CLIENT
DomainParticipant* client =
    DomainParticipantFactory::get_instance()->create_participant(0, client_qos);
if (nullptr == client)
{
    // Error
    return;
}
```

XML

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

Security

Configuring *Security* on *servers* and *clients* is done the same way as for any other participant. This section depicts the limitations imposed by the security enforcement on the communication between *clients* and *servers*, and which discovery information is propagated by a *server* depending on the security configuration of the *clients* and *servers* to which it is connected.

It is important to note that for enabling a secure discovery when using Discovery Server, *Fast DDS* must be compiled with security support (see *CMake options*), and the *Domain Governance Document* must explicitly encrypt the discovery.

As in SDP, when using this feature, the Domain Governance Document of all *clients* and *servers* connecting to a *server* must match that of the *server*, which implies that all *DomainParticipants* belonging to the same Discovery Server network must configure the discovery protection in the same manner.

Although the *server* mediates the discovery process and creates connections between *clients*, the *clients* themselves still go through the PKI (Public Key Infrastructure) exchange in order to have a secure communication between them.

Important: In order to keep the behavior consistent with the QoS Policies, the *server* does not check the *DomainParticipant Permissions Document* of the *DomainParticipants* that it is connecting.

Important: Security support for Discovery Server is only supported from Fast DDS v2.10.0 onward.

DomainParticipantListener Discovery Callbacks

As stated in *DomainParticipantListener*, the *DomainParticipantListener* is an abstract class defining the callbacks that will be triggered in response to state changes on the *DomainParticipant*. Fast DDS defines three callbacks attached to events that may occur during discovery: *on_participant_discovery()*, *on_data_reader_discovery()*, *on_data_writer_discovery()*. Further information about the *DomainParticipantListener* is provided in the *DomainParticipantListener* section. The following is an example of the implementation of *DomainParticipantListener* discovery callbacks.

```
class DiscoveryDomainParticipantListener : public DomainParticipantListener
{
    /* Custom Callback on_participant_discovery */
    void on_participant_discovery(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::ParticipantDiscoveryInfo&& info,
        bool& should_be_ignored) override
    {
        should_be_ignored = false;
        static_cast<void>(participant);
        switch (info.status){
            case eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOVERED_
↳PARTICIPANT:
            {
                /* Process the case when a new DomainParticipant was found in the domain_
↳*/

                std::cout << "New DomainParticipant '" << info.info.m_participantName <<
                    "' with ID '" << info.info.m_guid.entityId << "' and GuidPrefix '" <<
                    info.info.m_guid.guidPrefix << "' discovered." << std::endl;
```

(continues on next page)

(continued from previous page)

```

        /* The following line can be substituted to evaluate whether the_
↪ discovered participant should be ignored */
        bool ignoring_condition = false;
        if (ignoring_condition)
        {
            should_be_ignored = true;    // Request the ignoring of the_
↪ discovered participant
        }
    }
    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 << "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_data_reader_discovery */
void on_data_reader_discovery(
    DomainParticipant* participant,
    eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info,
    bool& should_be_ignored) override
{
    should_be_ignored = false;
    static_cast<void>(participant);
    switch (info.status){
        case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERED_READER:
        {
            /* Process the case when a new datareader was found in the domain */
            std::cout << "New DataReader subscribed to topic '" << info.info.
↪ topicName() <<
                "' of type '" << info.info.typeName() << "' discovered";
            /* The following line can be substituted to evaluate whether the_
↪ discovered datareader should be ignored */
            bool ignoring_condition = false;
            if (ignoring_condition)
            {
                should_be_ignored = true;    // Request the ignoring of the_
↪ discovered datareader
            }
        }
        break;
        case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::CHANGED_QOS_READER:
            /* Process the case when a datareader changed its QOS */
            break;
    }
}

```

(continues on next page)

(continued from previous page)

```

        case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::REMOVED_READER:
            /* Process the case when a datareader was removed from the domain */
            std::cout << "DataReader subscribed to topic '" << info.info.topicName()
<<
                "' of type '" << info.info.typeName() << "' left the domain.";
            break;
        }
    }

    /* Custom Callback on_data_writer_discovery */
    void on_data_writer_discovery(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info,
        bool& should_be_ignored) override
    {
        should_be_ignored = false;
        static_cast<void>(participant);
        switch (info.status){
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERED_WRITER:
            {
                /* Process the case when a new datawriter was found in the domain */
                std::cout << "New DataWriter publishing under topic '" << info.info.
<<topicName() <<
                    "' of type '" << info.info.typeName() << "' discovered";
                /* The following line can be substituted to evaluate whether the
<<discovered datawriter should be ignored */
                bool ignoring_condition = false;
                if (ignoring_condition)
                {
                    should_be_ignored = true;    // Request the ignoring of the
<<discovered datawriter
                }
            }
            break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::CHANGED_QOS_WRITER:
                /* Process the case when a datawriter changed its QOS */
                break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::REMOVED_WRITER:
                /* Process the case when a datawriter was removed from the domain */
                std::cout << "DataWriter publishing under topic '" << info.info.
<<topicName() <<
                    "' of type '" << info.info.typeName() << "' left the domain.";
                break;
            }
        }
    }
};

```

To use the previously implemented discovery callbacks in `DiscoveryDomainParticipantListener` class, which inherits from the `DomainParticipantListener`, an object of this class is created and registered as a listener of the `DomainParticipant`.

```
// Create the participant QoS and configure values
DomainParticipantQos pqos;

// Create a custom user DomainParticipantListener
DiscoveryDomainParticipantListener* plistener = new DiscoveryDomainParticipantListener();
// Pass the listener on DomainParticipant creation.
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(
        0, pqos, plistener);
```

Important: Read more about callbacks and its hierarchy [here](#)

7.21 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*.

7.21.1 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.

- *TransportDescriptorInterface*
- *TransportInterface*
- *Locator*
- *Chaining of transports*

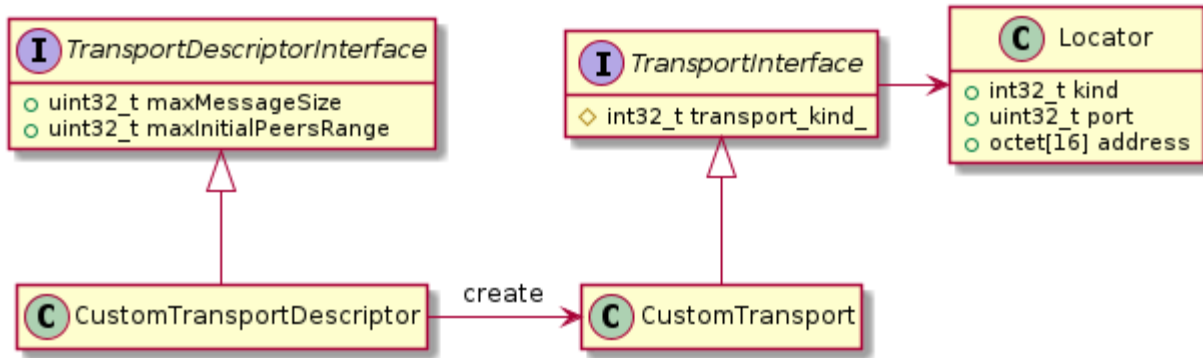


Fig. 10: Transport API diagram

TransportDescriptorInterface

Any class that implements the `TransportDescriptorInterface` is known as a `TransportDescriptor`. It acts as a *builder* for a given transport, meaning that it allows to configure the transport, and then a new *Transport* can be built according to this configuration using its `create_transport` factory member function.

Data members

The `TransportDescriptorInterface` defines the following data members:

Member	Data type	Description
<code>maxMessageSize</code>	<code>uint32_t</code>	Maximum size of a single message in the transport.
<code>maxInitialPeersRange</code>	<code>uint32_t</code>	Number of channels opened with each initial remote peer.

Any implementation of *TransportDescriptorInterface* should add as many data members as required to fully configure the transport it describes.

TransportInterface

A `Transport` is any class that implements the `TransportInterface`. It is the object that actually performs the message distribution over a physical transport.

Each `Transport` class defines its own `kind`, a unique identifier that is used to check the compatibility of a *Locator* with a `Transport`, i.e., determine whether a `Locator` refers to a `Transport` or not.

Applications do not create the `Transport` instance themselves. Instead, applications use a `TransportDescriptor` instance to configure the desired transport, and add this configured instance to the list of user-defined transports of the *DomainParticipant*. The `DomainParticipant` will use the factory function on the `TransportDescriptor` to create the `Transport` when required.

```

DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto udp_transport = std::make_shared<UDIPv4TransportDescriptor>();
udp_transport->sendBufferSize = 9216;
udp_transport->receiveBufferSize = 9216;
  
```

(continues on next page)

(continued from previous page)

```

udp_transport->non_blocking_send = true;

// [OPTIONAL] ThreadSettings configuration
udp_transport->default_reception_threads(eprosima::fastdds::rtps::ThreadSettings{2, 2, 2,
↪ 2});
udp_transport->set_thread_config_for_port(12345, eprosima::fastdds::rtps::ThreadSettings
↪ {3, 3, 3, 3});

// 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
<code>transport_kind_</code>	<code>int32_t</code>	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
<code>LOCATOR_KIND_RESERVED</code>	0	None. Reserved value for internal use.
<code>LOCATOR_KIND_UDPv4</code>	1	<i>UDP Transport</i> over IPv4.
<code>LOCATOR_KIND_UDPv6</code>	2	<i>UDP Transport</i> over IPv6.
<code>LOCATOR_KIND_TCPv4</code>	4	<i>TCP Transport</i> over IPv4.
<code>LOCATOR_KIND_TCPv6</code>	8	<i>TCP Transport</i> over IPv6.
<code>LOCATOR_KIND_SHM</code>	16	<i>Shared Memory Transport</i> .

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.

Configuring IP locators with IPLocator

IPLocator is an auxiliary static class that offers methods to manipulate IP based locators. It is convenient when setting up a new *UDP Transport* or *TCP Transport*, as it simplifies setting IPv4 and IPv6 addresses, or manipulating ports.

For example, normally users configure the physical port and do not need to worry about logical ports. However, IPLocator allows to manage them if needed.

```
// We will configure a TCP locator with IPLocator
Locator_t locator;

// Get & set the physical port
uint16_t physical_port = IPLocator::getPhysicalPort(locator);
IPLocator::setPhysicalPort(locator, 5555);

// On TCP locators, we can get & set the logical port
uint16_t logical_port = IPLocator::getLogicalPort(locator);
IPLocator::setLogicalPort(locator, 7400);

// Set WAN address
IPLocator::setWan(locator, "80.88.75.55");
```

Fast DDS also allows to specify locator addresses using names. When an address is specified by a name, *Fast DDS* will query the known hosts and available DNS servers to try to resolve the IP address. This address will in turn be used to create the listening locator in the case of *server*, or as the address of the remote *server* in the case of *clients* (and *servers* that connect to other *servers*).

C++

```
Locator_t locator;
auto response = eprosima::fastrtps::rtps::IPLocator::resolveNameDNS("localhost");
// Get the first returned IPv4
if (response.first.size() > 0)
{
    IPLocator::setIPv4(locator, response.first.begin()->data());
    locator.port = 11811;
}
// Use the locator to create server or client
```

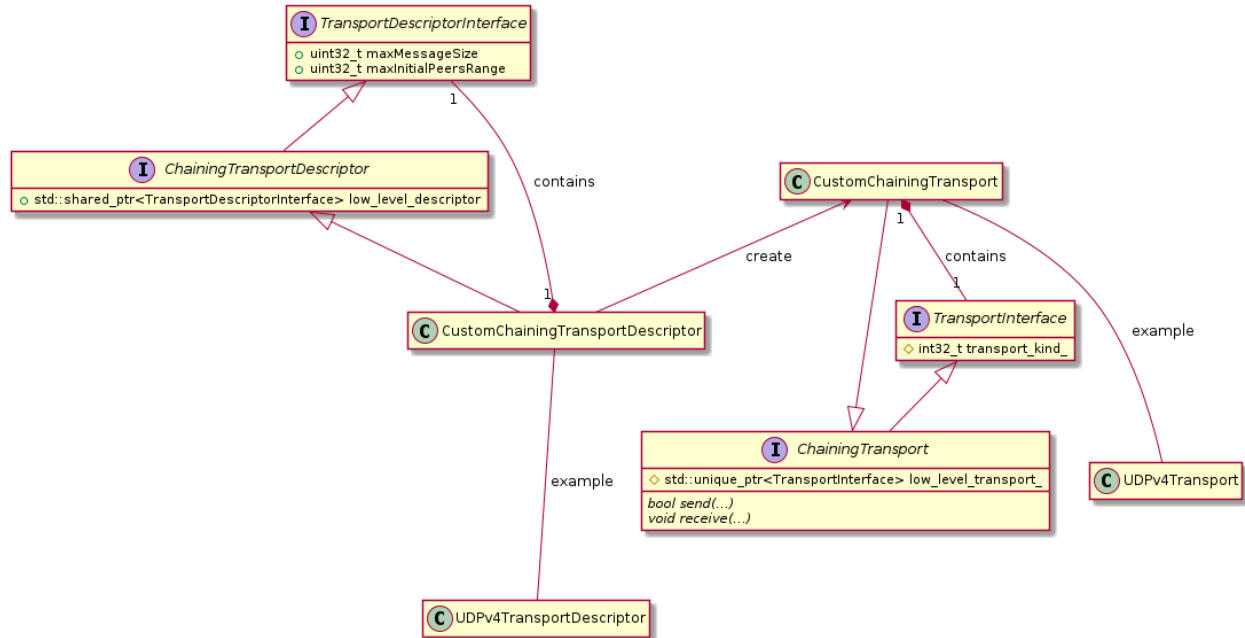
XML

```
<locator>
  <udp4>
    <port>11811</port>
    <address>localhost</address>
  </udp4>
</locator>
```

Warning: Currently, XML only supports loading IP addresses by name for UDP transport.

Chaining of transports

There are use cases where the user needs to pre-process out-coming information before being sent to network and also the incoming information after being received. *Transport API* offers two interfaces for implementing this kind of functionality: `ChainingTransportDescriptor` and `ChainingTransport`.



These extensions allow to implement a new Transport which depends on another one (called here as `low_level_transport_`). The user can override the `send()` function, pre-processing the out-coming buffer before calling the associated `low_level_transport_`. Also, when a incoming buffer arrives to the `low_level_transport_`, this one calls the overridden `receive()` function to allow to pre-process the buffer.

ChainingTransportDescriptor

Implementing `ChainingTransportDescriptor` allows to configure the new Transport and set the `low_level_transport_` on which it depends. The associated `low_level_transport_` can be any transport which inherits from `TransportInterface` (including another `ChainingTransport`).

The `ChainingTransportDescriptor` defines the following data members:

Member	Data type	Description
<code>low_level_descriptor</code>	<code>std::shared_ptr<TransportDescriptorInterface></code>	Transport descriptor of the <code>low_level_transport_</code> .

User has to specify the `low_level_transport_` in the definition of its new custom transport.

```

DomainParticipantQos qos;

auto udp_transport = std::make_shared<UDpv4TransportDescriptor>();

// Create a descriptor for the new transport.
  
```

(continues on next page)

(continued from previous page)

```

// The low level transport will be a UDPv4Transport.
auto custom_transport = std::make_shared<CustomChainingTransportDescriptor>(udp_
↳transport);

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(custom_transport);

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

```

ChainingTransport

This interface forces the user to implement `send()` and `receive()` functions. The idea is to pre-process the buffer and after, call to the next level.

```

class CustomChainingTransport : public eprosima::fastdds::rtps::ChainingTransport
{
public:

    CustomChainingTransport(
        const CustomChainingTransportDescriptor& descriptor)
        : ChainingTransport(descriptor)
        , descriptor_(descriptor)
    {
    }

    eprosima::fastdds::rtps::TransportDescriptorInterface* get_configuration()
    {
        return &descriptor_;
    }

    bool send(
        eprosima::fastrtps::rtps::SenderResource* low_sender_resource,
        const eprosima::fastrtps::rtps::octet* send_buffer,
        uint32_t send_buffer_size,
        eprosima::fastrtps::rtps::LocatorsIterator* destination_locators_begin,
        eprosima::fastrtps::rtps::LocatorsIterator* destination_locators_end,
        const std::chrono::steady_clock::time_point& timeout) override
    {
        //
        // Preprocess outgoing buffer.
        //

        // Call low level transport
        return low_sender_resource->send(send_buffer, send_buffer_size, destination_
↳locators_begin,
            destination_locators_end, timeout);
    }

    void receive(

```

(continues on next page)

(continued from previous page)

```

        eprosima::fastdds::rtps::TransportReceiverInterface* next_receiver,
        const eprosima::fastrtps::rtps::octet* receive_buffer,
        uint32_t receive_buffer_size,
        const eprosima::fastrtps::rtps::Locator_t& local_locator,
        const eprosima::fastrtps::rtps::Locator_t& remote_locator) override
    {
        //
        // Preprocess incoming buffer.
        //

        // Call upper level
        next_receiver->OnDataReceived(receive_buffer, receive_buffer_size, local_locator,
        ↪ remote_locator);
    }

private:

    CustomChainingTransportDescriptor descriptor_;
};

```

7.21.2 UDP Transport

UDP is a connectionless transport, where the receiving *DomainParticipant* must open a UDP port listening for incoming messages, and the sending *DomainParticipant* sends messages to this port.

Warning: This documentation assumes the reader has basic knowledge of UDP/IP concepts, since terms like Time To Live (TTL), socket buffers, and port numbering are not explained in detail. However, it is possible to configure a basic UDP transport on *Fast DDS* without this knowledge.

UDPTransportDescriptor

eProsima Fast DDS implements UDP transport for both UDPv4 and UDPv6. Each of these transports is independent from the other, and has its own *TransportDescriptorInterface*. However, all their *TransportDescriptorInterface* data members are common.

The following table describes the common data members for both UDPv4 and UDPv6.

Member	Data type	Default	Description
sendBufferSize	uint32_t	0	Size of the sending buffer of the socket (octets).
receiveBufferSize	uint32_t	0	Size of the receiving buffer of the socket (octets).
netmask_filter	NetmaskFilterKind	AUTO	See Netmask filtering .
allowlist	vector<pair<string, NetmaskFilterKind>>	Empty vector	List of allowed interfaces with netmask filter configuration. See Allowlist .
blocklist	vector<string>	Empty vector	List of blocked interfaces. See Blocklist .
interfaceWhitelist	vector<string>	Empty vector	List of allowed interfaces. See Interface Whitelist .
TTL	uint8_t	1	Time to live, in number of hops.
m_output_udp_socket	uint16_t	0	Port number for the outgoing messages.
non_blocking_send	bool	false	Do not block on send operations (*).
default_reception_thread_settings	ThreadSettings		Default ThreadSettings for the reception threads.
reception_threads	std::map<uint32_t, ThreadSettings>		ThreadSettings for the reception threads on specific ports.

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.

UDIPv4TransportDescriptor

UDIPv4TransportDescriptor has no additional data members from the common ones described in [UDIPvTransportDescriptor](#).

Note: The kind value for a UDIPv4TransportDescriptor is given by the value `LOCATOR_KIND_UDIPv4`.

UDIPv6TransportDescriptor

UDIPv6TransportDescriptor has no additional data members from the common ones described in [UDIPvTransportDescriptor](#).

Note: The kind value for a UDIPv6TransportDescriptor is given by the value `LOCATOR_KIND_UDIPv6`.

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 *UD Pv4TransportDescriptor* (for UDPv4) or *UD Pv6TransportDescriptor* (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<UD Pv4TransportDescriptor>();
udp_transport->sendBufferSize = 9216;
udp_transport->receiveBufferSize = 9216;
udp_transport->non_blocking_send = true;

// [OPTIONAL] ThreadSettings configuration
udp_transport->default_reception_threads(eprosima::fastdds::rtps::ThreadSettings{2, 2, 2,
↪ 2});
udp_transport->set_thread_config_for_port(12345, eprosima::fastdds::rtps::ThreadSettings
↪ {3, 3, 3, 3});

// 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" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com">
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>udp_transport</transport_id>
        <type>UD Pv4</type>
        <sendBufferSize>9216</sendBufferSize>
        <receiveBufferSize>9216</receiveBufferSize>
        <non_blocking_send>true</non_blocking_send>
        <default_reception_threads>
          <scheduling_policy>2</scheduling_policy>
          <priority>2</priority>
          <affinity>2</affinity>
          <stack_size>2</stack_size>
        </default_reception_threads>
        <reception_threads>
          <reception_thread port="12345">
            <scheduling_policy>3</scheduling_policy>
            <priority>3</priority>
            <affinity>3</affinity>
            <stack_size>3</stack_size>
          </reception_thread>
        </reception_threads>
      </transport_descriptor>
    </transport_descriptors>
  </profiles>
</dds>
```

(continues on next page)

(continued from previous page)

```

        </reception_threads>
      </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="UDPParticipant">
      <rtps>
        <userTransports>
          <transport_id>udp_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
      </rtps>
    </participant>
  </profiles>
</dds>

```

7.21.3 TCP Transport

TCP is a connection oriented transport, so the *DomainParticipant* must establish a TCP connection to the remote peer before sending data messages. Therefore, one of the communicating DomainParticipants (the one acting as *server*) must open a TCP port listening for incoming connections, and the other one (the one acting as *client*) must connect to this port.

Note: The *server* and *client* concepts are independent from the DDS concepts of *Publisher*, *Subscriber*, *DataWriter*, and *DataReader*. Also, these concepts are independent from the *eProsima Discovery Server* servers and clients (*Discovery Server Settings*). Any of them can act as a *TCP Server* or *TCP Client* when establishing the connection, and the DDS communication will work over this connection.

Warning: This documentation assumes the reader has basic knowledge of TCP/IP concepts, since terms like Time To Live (TTL), Cyclic Redundancy Check (CRC), Transport Layer Security (TLS), socket buffers, and port numbering are not explained in detail. However, it is possible to configure a basic TCP transport on *Fast DDS* without this knowledge.

TCPTransportDescriptor

eProsima Fast DDS implements TCP transport for both TCPv4 and TCPv6. Each of these transports is independent from the other, and has its own `TransportDescriptorInterface`. However, they share many of their features, and most of the `TransportDescriptorInterface` data members are common.

The following table describes the common data members for both TCPv4 and TCPv6.

Member	Data type	Default	Description
<code>sendBuf:</code>	<code>uint32_t</code>	0	Size of the sending buffer of the socket (octets).
<code>receive:</code>	<code>uint32_t</code>	0	Size of the receiving buffer of the socket (octets).
<code>netmask:</code>	<code>NetmaskFilter</code>	AUTO	See Netmask filtering .
<code>allowlist:</code>	<code>vector<pair<NetmaskFilter, NetmaskFilter>></code>	Empty vector	List of allowed interfaces with netmask filter configuration. See Allowlist .
<code>blocklist:</code>	<code>vector<string></code>	Empty vector	List of blocked interfaces. See Blocklist .
<code>interface:</code>	<code>vector<string></code>	Empty vector	List of allowed interfaces See Interface Whitelist .
<code>TTL:</code>	<code>uint8_t</code>	1	Time to live, in number of hops.
<code>listenlist:</code>	<code>vector<uint></code>	Empty vector	List of ports to listen as <i>server</i> . If a port is set to 0, an available port will be automatically assigned.
<code>keep_alive:</code>	<code>uint32_t</code>	5000	Frequency of RTCP keep alive requests (in ms).
<code>keep_alive:</code>	<code>uint32_t</code>	1500	Time since sending the last keep alive request to consider a connection as broken (in ms).
<code>max_logical:</code>	<code>uint16_t</code>	100	Maximum number of logical ports to try during RTCP negotiation.
<code>logical:</code>	<code>uint16_t</code>	20	Maximum number of logical ports per request to try during RTCP negotiation.
<code>logical:</code>	<code>uint16_t</code>	2	Increment between logical ports to try during RTCP negotiation.
<code>enable_:</code>	<code>bool</code>	false	Enables the TCP_NODELAY socket option.
<code>non_block:</code>	<code>bool</code>	false	Do not block on send operations (*).
<code>calculate:</code>	<code>bool</code>	true	True to calculate and send CRC on message headers.
<code>check_crc:</code>	<code>bool</code>	true	True to check the CRC of incoming message headers.
<code>apply_ssl:</code>	<code>bool</code>	false	True to use TLS. See TLS over TCP .
<code>tls_config:</code>	<code>TLSConfig</code>		Configuration for TLS. See TLS over TCP .
<code>default:</code>	<code>ThreadSettings</code>		Default ThreadSettings for the reception threads.
<code>reception:</code>	<code>std::map<uint, ThreadSettings></code>		ThreadSettings for the reception threads on specific ports.
<code>keep_alive:</code>	<code>ThreadSettings</code>		ThreadSettings for the thread keeping alive TCP connections.
<code>accept:</code>	<code>ThreadSettings</code>		ThreadSettings for the threads processing incoming TCP connection requests.
<code>tcp_negotiation:</code>	<code>uint32_t</code>	0	Time to wait for logical port negotiation (in ms). If a logical port is under negotiation, it waits for the negotiation to finish up to this timeout before trying to send a message to that port. Setting this option to non-zero values increases the discovery time. Setting it to zero means no wait but could lead to loss of first messages.

Warning: Although the member `listening_ports` accepts multiple ports, only the first listening port will be effectively used. The rest of the ports will be ignored.

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.

Note: When `non_blocking_send` is set to `true`, send operations will return immediately if the send buffer might get full, but no error will be returned to the upper layer. This means that the application will behave as if the packet is sent and lost.

When set to `false`, send operations will block until the network buffer has space for the packet.

TCPv4TransportDescriptor

The following table describes the data members that are exclusive for `TCPv4TransportDescriptor`.

Member	Data type	Default	Description
<code>wan_addr</code>	<code>octet[4]</code>	<code>[0, 0, 0, 0]</code>	Configuration for WAN. See <i>WAN or Internet Communication over TCPv4</i> .

Note: The kind value for a `TCPv4TransportDescriptor` is given by the value `LOCATOR_KIND_TCPv4`.

TCPv6TransportDescriptor

`TCPv6TransportDescriptor` has no additional data members from the common ones described in *TCPTransportDescriptor*.

Note: The kind value for a `TCPv6TransportDescriptor` is given by the value `LOCATOR_KIND_TCPv6`.

Enabling TCP Transport

There are several ways of enabling TCP transport in *eprosima Fast DDS*. According to the facet of each scenario, one method might suit better than the others.

Configuration of Builtin Transports

The first option is to modify the builtin transports that are responsible of the creation of the DomainParticipant transports. The existing configuration that enables TCP transports is `LARGE_DATA`. This option instantiates a UDPv4, a TCPv4 and a SHM transport, respectively. UDP protocol will be used for multicast announcements during the participant discovery phase (see *Discovery phases*) while the participant liveliness and the application data delivery occurs over TCP or SHM. This configuration enables auto discovery and does not require to manually set up each participant IP and listening port. Hence, avoiding the typical Server-Client configuration.

Builtin Transports can be configured using the `FASTDDS_BUILTIN_TRANSPORTS` environment variable (see *FASTDDS_BUILTIN_TRANSPORTS*), XML profiles (see *RTPS element type*) or via code.

Environment Variable

```
export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <!--
      UDP transport for PDP and SHM/TCPv4 transport for both EDP and application_
    -->
    <participant profile_name="large_data_builtin_transports" is_default_profile=
      "true">
      <rtps>
        <builtinTransports>LARGE_DATA</builtinTransports>
      </rtps>
    </participant>
  </profiles>
</dds>
```

C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;
qos.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_DATA);
```

Note: Note that `LARGE_DATA` configuration of the builtin transports will also create a SHM transport along the UDP and TCP transports. Shared Memory will be used whenever it is possible. Manual configuration will be required if a TCP communication is required when SHM is feasible. (See *Large Data Mode*).

Server-Client Configuration

To set up a Server-Client configuration you need to create an instance of *TCPv4TransportDescriptor* (for TCPv4) or *TCPv6TransportDescriptor* (for TCPv6), and add it to the user transport list of the DomainParticipant.

Depending on the TCP transport descriptor settings and network locators defined, the DomainParticipant can act as a TCP Server or TCP Client.

- **TCP Server:** If you provide `listening_ports` on the descriptor, the DomainParticipant will act as *TCP server*, listening for incoming remote connections on the given ports. The examples below show this procedure in both C++ code and XML file.

C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared
    <eprosima::fastdds::rtps::TCPv4TransportDescriptor>();
tcp_transport->add_listener_port(5100);

// [OPTIONAL] ThreadSettings configuration
tcp_transport->default_reception_threads(eprosima::fastdds::rtps::ThreadSettings{-1,
    <- 0, 0, -1});
tcp_transport->set_thread_config_for_port(12345,
    <- eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
tcp_transport->keep_alive_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0,
    <- -1};
tcp_transport->accept_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1}
    <-;

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tcp_transport);

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

// [OPTIONAL] Set unicast locators
eprosima::fastrtps::rtps::Locator_t locator;
locator.kind = LOCATOR_KIND_TCPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, "192.168.1.10");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(locator, 5100);
// [OPTIONAL] Logical port default value is 0, automatically assigned.
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(locator, 5100);

qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
qos.wire_protocol().default_unicast_locator_list.push_back(locator);
```

XML

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

(continues on next page)

(continued from previous page)

```

    <transport_id>tcp_server_transport</transport_id>
    <type>TCPv4</type>
    <listening_ports>
      <port>5100</port>
    </listening_ports>
    <!-- Optional thread configuration -->
    <default_reception_threads>
      <scheduling_policy>-1</scheduling_policy>
      <priority>0</priority>
      <affinity>0</affinity>
      <stack_size>-1</stack_size>
    </default_reception_threads>
    <reception_threads>
      <reception_thread port="12345">
        <scheduling_policy>-1</scheduling_policy>
        <priority>0</priority>
        <affinity>0</affinity>
        <stack_size>-1</stack_size>
      </reception_thread>
    </reception_threads>
    <keep_alive_thread>
      <scheduling_policy>-1</scheduling_policy>
      <priority>0</priority>
      <affinity>0</affinity>
      <stack_size>-1</stack_size>
    </keep_alive_thread>
    <accept_thread>
      <scheduling_policy>-1</scheduling_policy>
      <priority>0</priority>
      <affinity>0</affinity>
      <stack_size>-1</stack_size>
    </accept_thread>
  </transport_descriptor>
</transport_descriptors>

<participant profile_name="tcp_server_participant">
  <rtps>
    <userTransports>
      <transport_id>tcp_server_transport</transport_id>
    </userTransports>
    <useBuiltinTransports>false</useBuiltinTransports>
    <!-- Optional unicast locator set -->
    <defaultUnicastLocatorList>
      <locator>
        <tcpv4>
          <address>192.168.1.10</address>
          <physical_port>5100</physical_port>
          <!-- Optional logical port set -->
          <port>5100</port>
        </tcpv4>
      </locator>
    </defaultUnicastLocatorList>
  </rtps>
</participant>

```

(continues on next page)

(continued from previous page)

```

        <!-- Optional metatraffic unicast locator set -->
        <builtin>
            <metatrafficUnicastLocatorList>
                <locator>
                    <tcpv4>
                        <address>192.168.1.10</address>
                        <physical_port>5100</physical_port>
                        <!-- Optional logical port set -->
                        <port>5100</port>
                    </tcpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>
</profiles>
</dds>

```

- **TCP Client:** If you provide `initialPeersList` to the `DomainParticipant`, it will act as *TCP client*, trying to connect to the remote *servers* at the given addresses and ports. The examples below show this procedure in both C++ code and XML file. See *Initial peers* for more information about their configuration.

C++

```

eprosima::fastdds::dds::DomainParticipantQos qos;

// Disable the built-in Transport Layer.
qos.transport().use_builtin_transports = false;

// Create a descriptor for the new transport.
// Do not configure any listener port
auto tcp_transport = std::make_shared
    <eprosima::fastdds::rtps::TCpv4TransportDescriptor>();
qos.transport().user_transports.push_back(tcp_transport);

// [OPTIONAL] ThreadSettings configuration
tcp_transport->default_reception_threads(eprosima::fastdds::rtps::ThreadSettings{-1,
    < 0, 0, -1});
tcp_transport->set_thread_config_for_port(12345,
    < eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
tcp_transport->keep_alive_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0,
    < -1};
tcp_transport->accept_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1}
    < ;

// Set initial peers.
eprosima::fastrtps::rtps::Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(initial_peer_locator, "192.168.1.10");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(initial_peer_locator, 5100);
// If the logical port is set in the server side, it must be also set here with the
    < same value.
// If not set in the server side in a unicast locator, do not set it here.

```

(continues on next page)

(continued from previous page)

```
eprosima::fastrtps::rtps::IPlocator::setLogicalPort(initial_peer_locator, 5100);
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer_locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com">
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>tcp_client_transport</transport_id>
        <type>TCPv4</type>
        <!-- Optional thread configuration -->
        <default_reception_threads>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </default_reception_threads>
        <reception_threads>
          <reception_thread port="12345">
            <scheduling_policy>-1</scheduling_policy>
            <priority>0</priority>
            <affinity>0</affinity>
            <stack_size>-1</stack_size>
          </reception_thread>
        </reception_threads>
        <keep_alive_thread>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </keep_alive_thread>
        <accept_thread>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </accept_thread>
      </transport_descriptor>
    </transport_descriptors>

    <participant profile_name="tcp_client_participant">
      <rtps>
        <userTransports>
          <transport_id>tcp_client_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
        <builtin>
          <initialPeersList>
            <locator>
```

(continues on next page)

(continued from previous page)

```

        <tcpv4>
        <address>192.168.1.10</address>
        <physical_port>5100</physical_port>
        <!-- To be set if set in server -->
        <port>5100</port>
    </tcpv4>
</locator>
</initialPeersList>
</builtin>
</rtps>
</participant>
</profiles>
</dds>

```

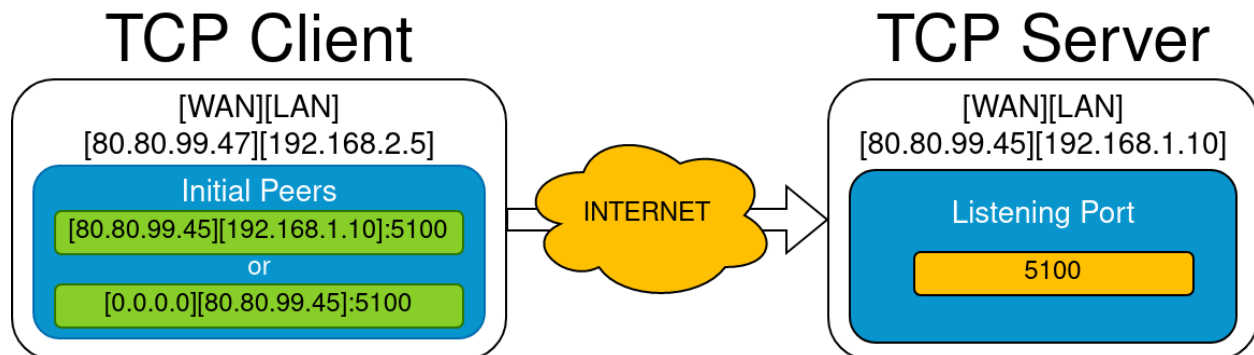
Note: Manually setting unicast locators is optional. If not setting them or setting them with a logical port of 0, the client's initial peer shouldn't set its logical port (or set it to 0). Otherwise, initial peer's logical port must match server's unicast logical port.

HelloWorldExampleTCP shows how to use and configure a TCP transport.

WAN or Internet Communication over TCPv4

Fast DDS is able to connect through the Internet or other WAN networks when configured properly. To achieve this kind of scenarios, the involved network devices such as routers and firewalls must add the rules to allow the communication.

For example, imagine we have the scenario represented on the following figure:



- A DomainParticipant acts as a *TCP server* listening on port 5100 and is connected to the WAN through a router with public IP 80.80.99.45.
- Another DomainParticipant acts as a *TCP client* and has configured the server's IP address and port in its *Initial peers* list.

By using `set_WAN_address(wan_ip)`, the WAN IP address is set on the participant's locators that are communicated during the discovery phase.

Like in the LAN case, manually setting unicast locators is optional. However, in this case, there are some considerations to take into account when setting its IP addresses:

- Setting the WAN IP address using the `setWAN()` method in unicast locators is ineffective because it gets overridden by the `set_WAN_address()` call.

- For assigning IP addresses to unicast locators, use only the `setIPv4()` or `setIPv6()` methods, which are LAN IP setters. There are some configurations which allow using these setters with a WAN IP address.

Depending on whether the server has manually set its metatraffic unicast locators and default unicast locators, the client needs to adjust its initial peer list accordingly:

- If the server's unicast locators are configured with the LAN IP address:
 - The initial peer can be set up with only the server's WAN IP using the LAN IP setter.
 - Alternatively, it can be configured with both the server's LAN and WAN IP addresses using the LAN setter for the LAN IP and the WAN setter for the WAN IP.
- If the server's unicast locators are configured with the WAN IP address:
 - The initial peer must be set up with only the server's WAN IP using the LAN setter.
 - Alternatively, it can be configured with the WAN IP address using both the LAN and WAN setters.
- If the server has not set any unicast locators:
 - The initial peer can be configured with only the server's WAN IP using the LAN setter.
 - Alternatively, it can be configured with both the server's LAN and WAN IP addresses using the LAN setter for the LAN IP and the WAN setter for the WAN IP.

Note: Manually setting unicast locators is optional. If not setting them or setting them with a logical port of 0, the client's initial peer shouldn't set its logical port (or set it to 0). Otherwise, initial peer's logical port must match server's unicast logical port.

On the server side, the router must be configured to forward to the *TCP server* all traffic incoming to port 5100. Typically, a NAT routing of port 5100 to our machine is enough. Any existing firewall should be configured as well.

In addition, to allow incoming connections through a WAN, the *TCPv4TransportDescriptor* must indicate its **public** IP address in the `wan_addr` data member. The following examples show how to configure the *DomainParticipant* both in C++ and XML.

C++

```
eprosima::fastdds::dds::DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>();
tcp_transport->add_listener_port(5100);
tcp_transport->set_WAN_address("80.80.99.45");

// [OPTIONAL] ThreadSettings configuration
tcp_transport->default_reception_threads(eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
tcp_transport->set_thread_config_for_port(12345, eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
tcp_transport->keep_alive_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1};
tcp_transport->accept_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1};

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tcp_transport);
```

(continues on next page)

(continued from previous page)

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

// [OPTIONAL] Set unicast locators (do not use setWAN(), set_WAN_address() overwrites it)
eprosima::fastrtps::rtps::Locator_t locator;
locator.kind = LOCATOR_KIND_TCPv4;
// [RECOMMENDED] Use the LAN address of the server
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, "192.168.1.10");
// [ALTERNATIVE] Use server's WAN address. In that case, initial peers must be configured
// only with server's WAN address.
// eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, "80.80.99.45");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(locator, 5100);
// [OPTIONAL] Logical port default value is 0, automatically assigned.
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(locator, 5100);

qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
qos.wire_protocol().default_unicast_locator_list.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>tcp_server_wan_transport</transport_id>
        <type>TCPv4</type>
        <listening_ports>
          <port>5100</port>
        </listening_ports>
        <wan_addr>80.80.99.45</wan_addr>
        <!-- Optional thread configuration -->
        <default_reception_threads>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </default_reception_threads>
        <reception_threads>
          <reception_thread port="12345">
            <scheduling_policy>-1</scheduling_policy>
            <priority>0</priority>
            <affinity>0</affinity>
            <stack_size>-1</stack_size>
          </reception_thread>
        </reception_threads>
        <keep_alive_thread>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </keep_alive_thread>
```

(continues on next page)

(continued from previous page)

```

        <accept_thread>
            <scheduling_policy>-1</scheduling_policy>
            <priority>0</priority>
            <affinity>0</affinity>
            <stack_size>-1</stack_size>
        </accept_thread>
    </transport_descriptor>
</transport_descriptors>

<participant profile_name="tcp_server_wan_participant">
    <rtps>
        <userTransports>
            <transport_id>tcp_server_wan_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
        <!-- Optional unicast locator set -->
        <defaultUnicastLocatorList>
            <locator>
                <tcpv4>
                    <address>192.168.1.10</address>
                    <!-- Alternatively use WAN address -->
                    <!-- <address>80.80.99.45</address> -->
                    <physical_port>5100</physical_port>
                    <!-- Optional logical port set -->
                    <port>5100</port>
                </tcpv4>
            </locator>
        </defaultUnicastLocatorList>
        <!-- Optional metatraffic unicast locator set -->
        <builtin>
            <metatrafficUnicastLocatorList>
                <locator>
                    <tcpv4>
                        <address>192.168.1.10</address>
                        <!-- Alternatively use WAN address -->
                        <!-- <address>80.80.99.45</address> -->
                        <physical_port>5100</physical_port>
                        <!-- Optional logical port set -->
                        <port>5100</port>
                    </tcpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>
</profiles>
</dds>

```

On the client side, the DomainParticipant must be configured with the **public** IP address and listening_ports of the *TCP server* as *Initial peers*.

C++

```

eprosima::fastdds::dds::DomainParticipantQos qos;

// Disable the built-in Transport Layer.
qos.transport().use_builtin_transports = false;

// Create a descriptor for the new transport.
// Do not configure any listener port
auto tcp_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
    ();
// [RECOMMENDED] Use client's WAN address if there are more clients in other local
    networks.
tcp_transport->set_WAN_address("80.80.99.47");
qos.transport().user_transports.push_back(tcp_transport);

// [OPTIONAL] ThreadSettings configuration
tcp_transport->default_reception_threads(eprosima::fastdds::rtps::ThreadSettings{-1, 0,
    0, -1});
tcp_transport->set_thread_config_for_port(12345, eprosima::fastdds::rtps::ThreadSettings
    {-1, 0, 0, -1});
tcp_transport->keep_alive_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1};
tcp_transport->accept_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1};

// Set initial peers.
eprosima::fastrtps::rtps::Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
// [RECOMMENDED] Use both WAN and LAN server addresses
eprosima::fastrtps::rtps::IPLocator::setIPv4(initial_peer_locator, "192.168.1.10");
eprosima::fastrtps::rtps::IPLocator::setWan(initial_peer_locator, "80.80.99.45");
// [ALTERNATIVE] Use server's WAN address only. Valid if server specified its unicast
    locators
// with its LAN or WAN address.
// eprosima::fastrtps::rtps::IPLocator::setIPv4(initial_peer_locator, "80.80.99.45");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(initial_peer_locator, 5100);
// If the logical port is set in the server side, it must be also set here with the same
    value.
// If not set in the server side in a unicast locator, do not set it here.
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(initial_peer_locator, 5100);

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

```

XML

```

<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>tcp_client_wan_transport</transport_id>
        <type>TCPv4</type>
        <!-- Recommended client's WAN set -->
        <wan_addr>80.80.99.47</wan_addr>
        <!-- Optional thread configuration -->
        <default_reception_threads>

```

(continues on next page)

(continued from previous page)

```

        <scheduling_policy>-1</scheduling_policy>
        <priority>0</priority>
        <affinity>0</affinity>
        <stack_size>-1</stack_size>
    </default_reception_threads>
    <reception_threads>
        <reception_thread port="12345">
            <scheduling_policy>-1</scheduling_policy>
            <priority>0</priority>
            <affinity>0</affinity>
            <stack_size>-1</stack_size>
        </reception_thread>
    </reception_threads>
    <keep_alive_thread>
        <scheduling_policy>-1</scheduling_policy>
        <priority>0</priority>
        <affinity>0</affinity>
        <stack_size>-1</stack_size>
    </keep_alive_thread>
    <accept_thread>
        <scheduling_policy>-1</scheduling_policy>
        <priority>0</priority>
        <affinity>0</affinity>
        <stack_size>-1</stack_size>
    </accept_thread>
</transport_descriptor>
</transport_descriptors>

<participant profile_name="tcp_client_wan_participant">
    <rtps>
        <userTransports>
            <transport_id>tcp_client_wan_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>>false</useBuiltinTransports>
        <builtin>
            <initialPeersList>
                <locator>
                    <tcpv4>
                        <!-- Recommended use of both WAN and LAN server -->
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.10</address>
                        <!-- Alternatively use server's WAN addresses only -->
                        <!-- <address>80.80.99.45</address> -->
                        <physical_port>5100</physical_port>
                        <!-- To be set if set in server -->
                        <port>5100</port>
                    </tcpv4>
                </locator>
            </initialPeersList>
        </builtin>
    </rtps>

```

(continues on next page)

(continued from previous page)

```
</participant>
</profiles>
</dds>
```

HelloWorldExampleTCP

A TCP version of helloworld example can be found in the [HelloWorldExampleTCP folder](#). It shows a publisher and a subscriber that communicate through TCP. The publisher is configured as *TCP server* while the Subscriber is acting as *TCP client*.

7.21.4 Shared Memory Transport

The shared memory (SHM) transport enables fast communications between entities running in the same processing unit/machine, relying on the shared memory mechanisms provided by the host operating system.

Note: Fast DDS utilizes the *DomainParticipant*'s `GuidPrefix_t` to identify peers running in the same host. Two participants with identical 4 first bytes on the `GuidPrefix_t` are considered to be running in the same host. `is_on_same_host_as()` API is provided to check this condition. Please, take also into account the caveats included in *GUID Prefix considerations for intra-process delivery*.

SHM transport provides better performance than other network transports like UDP / TCP, even when these transports use loopback interface. This is mainly due to the following reasons:

- Large message support: Network protocols need to fragment data in order to comply with the specific protocol and network stacks requirements, increasing communication overhead. SHM transport allows the copy of full messages where the only size limit is the machine's memory capacity.
- Reduce the number of memory copies: When sending the same message to different endpoints, SHM transport can directly share the same memory buffer with all the destination endpoints. Other protocols require to perform one copy of the message per endpoint.
- Less operating system overhead: Once initial setup is completed, shared memory transfers require much less system calls than the other protocols. Therefore, there is a performance/time consume gain by using SHM.

Definition of Concepts

This section describes basic concepts to help explain how the Shared Memory Transport works in order to deliver the data messages to the appropriate *DomainParticipant*. The purpose is not to be an exhaustive reference of the implementation, but to be a comprehensive explanation of each concept, so that users can configure the transport to their needs.

Many of the descriptions in this section will be made following the example use case depicted in the following figure, where *Participant 1* sends a data message to *Participant 2*. Please, refer to the figure when following the definitions.

Fig. 11: Sequence diagram for Shared Memory Transport

Segment

A *Segment* is a block of shared memory that can be accessed from different processes. Every DomainParticipant that has been configured with Shared Memory Transport creates a segment of shared memory. The DomainParticipant writes to this segment any data it needs to deliver to other DomainParticipants, and the remote DomainParticipants are able to read it directly using the shared memory mechanisms.

Note: Launching any of the processes with a higher privileged user (for instance, *root*) can lead to communication problems, as processes run by non-privileged users may not be able to write into the memory segment.

Every segment has a *segmentId*, a 16-character UUID that uniquely identifies each shared memory segment. These *segmentIds* are used to identify and access the segment of each DomainParticipant.

Segment Buffer

A buffer allocated in the shared memory Segment. It works as a container for a DDS message that is placed in the Segment. In other words, each message that the DomainParticipant writes on the Segment will be placed in a different buffer.

Buffer Descriptor

It acts as a pointer to a specific Segment Buffer in a specific Segment. It contains the *segmentId* and the offset of the Segment Buffer from the base of the Segment. When communicating a message to other DomainParticipants, Shared Memory Transport only distributes the Buffer Descriptor, avoiding the copy of the message from a DomainParticipant to another. With this descriptor, the receiving DomainParticipant can access the message written in the buffer, as is uniquely identifies the Segment (through the *segmentId*) and the Segment Buffer (through its offset).

Port

Represents a channel to communicate Buffer Descriptors. It is implemented as a ring-buffer in shared memory, so that any DomainParticipant can potentially read or write information on it. Each port has a unique identifier, a 32 bit number that can be used to refer to the port. Every DomainParticipant that has been configured with Shared Memory Transport creates a port to receive Buffer Descriptors. The identifier of this port is shared during the *Discovery*, so that remote peers know which port to use when they want to communicate with each DomainParticipant.

DomainParticipants create a listener to their receiving port, so that they can be notified when a new Buffer Descriptor is pushed to the port.

Port Health Check

Every time a DomainParticipant opens a Port (for reading or writing), a health check is performed to assess its correctness. The reason is that if one of the processes involved crashes while using a Port, that port can be left inoperative. If the attached listeners do not respond in a given timeout, the Port is considered damaged, and it is destroyed and created again.

SharedMemTransportDescriptor

In addition to the data members defined in the *TransportDescriptorInterface*, the *TransportDescriptor* for Shared Memory defines the following ones:

Member	Data type	De- fault	Accessor / Muta- tor	Description
segment_size_	uint32_t	512*10	segment_size()	Size of the shared memory segment (in octets).
port_queue_capa	uint32_t	512	port_queue_capac	The size of the listening port (in messages).
healthy_check_t	uint32_t	1000	healthy_check_ti	Timeout for the health check of ports (in milliseconds).
rtps_dump_file_	string	""	rtps_dump_file()	Full path of the protocol dump file.
default_receptio	<i>ThreadSettings</i>		default_receptio	Default <i>ThreadSettings</i> for the reception threads.
reception_thread	std::map<uint32_t, ThreadSettings>		reception_thread	<i>ThreadSettings</i> for the reception threads on specific ports.
dump_thread()	<i>ThreadSettings</i>		dump_thread()	<i>ThreadSettings</i> for the SHM dump thread.

If `rtps_dump_file_` is not empty, all the shared memory traffic on the *DomainParticipant* (sent and received) is traced to a file. The output file format is *tcpdump* hexadecimal text, and can be processed with protocol analyzer applications such as Wireshark. Specifically, to open the file using Wireshark, use the “Import from Hex Dump” option using the “Raw IPv4” encapsulation type.

Note: The kind value for a *SharedMemTransportDescriptor* is given by the value `LOCATOR_KIND_SHM`.

Warning: Setting a `segment_size()` close to or smaller than the data size poses a high risk of data loss, since the write operation will overwrite the buffer during a single send operation.

Enabling Shared Memory Transport

Fast DDS enables a SHM transport by default. Nevertheless, the application can enable other SHM transports if needed. To enable a new SHM transport in a *DomainParticipant*, first create an instance of *SharedMemTransportDescriptor*, and add it to the user transport list of the *DomainParticipant*.

The examples below show this procedure in both C++ code and XML file.

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
std::shared_ptr<SharedMemTransportDescriptor> shm_transport =
    std::make_shared<SharedMemTransportDescriptor>();

// [OPTIONAL] ThreadSettings configuration
```

(continues on next page)

(continued from previous page)

```
shm_transport->default_reception_threads(eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
shm_transport->set_thread_config_for_port(12345, eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
shm_transport->dump_thread(eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(shm_transport);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com">
    <transport_descriptors>
      <!-- Create a descriptor for the new transport -->
      <transport_descriptor>
        <transport_id>shm_transport</transport_id>
        <type>SHM</type>
        <default_reception_threads> <!-- OPTIONAL -->
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </default_reception_threads>
        <reception_threads> <!-- OPTIONAL -->
          <reception_thread port="12345">
            <scheduling_policy>-1</scheduling_policy>
            <priority>0</priority>
            <affinity>0</affinity>
            <stack_size>-1</stack_size>
          </reception_thread>
        </reception_threads>
        <dump_thread> <!-- OPTIONAL -->
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </dump_thread>
      </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>
</dds>
```

Note: In case that several transports are enabled, the discovery traffic is always performed using the UDP/TCP transport, even if the SHM transport is enabled in both participants running in the same machine. This may cause discovery issues if one or several of the participants only has SHM enabled and other participants use some other transport at the same time. Also, when two participants on the same machine have SHM transport enabled, the user data communication between them is automatically performed by SHM transport only. The rest of the enabled transports are not used between those two participants.

Hint: To configure discovery traffic through Shared Memory, the default builtin transports must be disabled. In that way, communication is performed completely using Shared Memory. The snippet examples below show this procedure in both C++ code and XML file. See [HelloWorldExampleSharedMem](#) for a complete example.

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
std::shared_ptr<SharedMemTransportDescriptor> shm_transport =
    std::make_shared<SharedMemTransportDescriptor>();

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(shm_transport);

// Explicit configuration of SharedMem transport
qos.transport().use_builtin_transports = false;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <transport_descriptors>
    <!-- Create a descriptor for the new transport -->
    <transport_descriptor>
      <transport_id>shm_transport_only</transport_id>
      <type>SHM</type>
    </transport_descriptor>
  </transport_descriptors>

  <participant profile_name="DisableBuiltinTransportsParticipant">
    <rtps>
      <!-- Link the Transport Layer to the Participant -->
      <userTransports>
        <transport_id>shm_transport_only</transport_id>
      </userTransports>
      <useBuiltinTransports>false</useBuiltinTransports>
    </rtps>
  </participant>
</profiles>
```

HelloWorldExampleSharedMem

A Shared Memory version of helloworld example can be found in the [HelloWorldExampleSharedMem](#) folder. It shows a publisher and a subscriber that communicate through Shared Memory.

7.21.5 Data-sharing delivery

Fast DDS allows to speed up communications between entities within the same machine by sharing the history of the *DataWriter* with the *DataReader* through shared memory. This prevents any of the overhead involved in the transport layer, effectively avoiding any data copy between *DataWriter* and *DataReader*.

Note: *Fast DDS* utilizes the *DomainParticipant*'s `GuidPrefix_t` to identify peers running in the same host. Two participants with identical 4 first bytes on the `GuidPrefix_t` are considered to be running in the same host. `is_on_same_host_as()` API is provided to check this condition. Please, take also into account the caveats included in *GUID Prefix considerations for intra-process delivery*.

Use of Data-sharing delivery does not prevent data copies between the application and the *DataReader* and *DataWriter*. These can be avoided in some cases using *Zero-Copy communication*.

Note: Although Data-sharing delivery uses shared memory, it differs from *Shared Memory Transport* in that Shared Memory is a full-compliant transport. That means that with Shared Memory Transport the data being transmitted must be copied from the *DataWriter* history to the transport and from the transport to the *DataReader*. With Data-sharing these copies can be avoided.

- *Overview*
- *Constraints*
- *Data-sharing delivery configuration*
- *DataReader and DataWriter history coupling*

Overview

When the *DataWriter* is created, *Fast DDS* will pre-allocate a pool of `max_samples + extra_samples` samples that reside in a shared memory mapped file. When publishing new data, the *DataWriter* will take a sample from this pool and add it to its history, and notify the *DataReader* which sample from the pool has the new data.

The *DataReader* will have access to the same shared memory mapped file, and will be able to access the data published by the *DataWriter*.

Constraints

This feature is available only if the following requirements are met:

- The *DataWriter* and *DataReader* have access to the same shared memory.
- The *Topic* has a bounded *TopicDataType*, i.e., its `is_bounded()` member function returns true.
- The Topic *is not keyed*.
- The DataWriter is configured with `PREALLOCATED_MEMORY_MODE` or `PREALLOCATED_WITH_REALLOC_MEMORY_MODE`.

There is also a limitation with the DataReader's HistoryQos. Using Data-sharing mechanism, the DataWriter's history is shared with the DataReaders. This means that the effective HistoryQos depth on the DataReader is, at most, the DataWriter's HistoryQos depth. To avoid confusions, set the DataReaders' history depth to a value equal or less than the DataWriter's.

Data-sharing delivery configuration

Data-sharing delivery can be configured in the *DataWriter* and the *DataReader* using *DataSharingQosPolicy*. Four attributes can be configured:

- The data-sharing delivery kind
- The shared memory directory
- The data-sharing domain identifiers.
- The maximum number of data-sharing domain identifiers.

Data-Sharing delivery kind

Can be set to one of three modes:

- **AUTO**: If both a DataWriter and DataReader meet the requirements, data-sharing delivery will be used between them. This is the default value.
- **ON**: Like **AUTO**, but the creation of the entity will fail if the requirements are not met.
- **OFF**: No data-sharing delivery will be used on this entity.

The following matrix shows when two entities are data-sharing compatible according to their configuration (given that the entity creation does not fail and that both entities have access to a shared memory):

		Reader		
		ON	OFF	AUTO
Writer	ON	Only if they have common domain IDs	No	Only if they have common domain IDs
	OFF	No	No	No
	AUTO	Only if they have common domain IDs	No	Only if the TopicDataType is bounded and they have common domain IDs

Data-sharing domain identifiers

Each entity defines a set of identifiers that represent a *domain* to which the entity belongs. Two entities will be able to use data-sharing delivery between them only if both have at least a common domain.

Users can define the domains of a *DataWriter* or *DataReader* with the *DataSharingQosPolicy*. If no domain identifier is provided by the user, the system will create one automatically. This automatic data-sharing domain will be unique for the machine where the entity is running. That is, all entities running on the same machine, and for which the user has configured no user-specific domains, will be able to use data-sharing delivery (given that the rest of requirements are met).

During the discovery phase, entities will exchange their domain identifiers and check if they can use Data-sharing to communicate.

Note: Even though a data-sharing domain identifier is a 64 bit integer, user-defined identifiers are restricted to 16 bit integers.

Maximum number of Data-sharing domain identifiers

The maximum number of domain identifiers that are expected to be received from a remote entity during discovery. If the remote entity defines (and sends) more than this number of domain identifiers, the discovery will fail.

By default there is no limit to the number of identifiers. The default value can be changed with the `max_domains()` function. Defining a finite number allows to preallocate the required memory to receive the list of identifiers during the entity creation, avoiding dynamic memory allocations afterwards. Note that a value of 0 means no limit.

Shared memory directory

If a user-defined directory is given for the shared memory files, this directory will be used for the memory-mapped files used for data-sharing delivery. If none is given, the default directory configured for the current system is used.

Configuring a user-defined directory may be useful in some scenarios:

- To select a file system with Huge TLB enabled for the memory-mapped files.
- To allow data-sharing delivery between containers that mount the same container.

DataReader and DataWriter history coupling

With traditional *Transport Layer* delivery, the DataReader and DataWriter keep separate and independent histories, each one with their own copy of the sample. Once the sample is sent through the transport and received by the DataReader, the DataWriter is free to remove the sample from its history without affecting the DataReader.

With data-sharing delivery, the DataReader directly accesses the data instance created by the DataWriter. This means that the samples in both the history of the DataReader and the DataWriter refer to the same object in the shared memory. Therefore, there is a strong coupling in the behavior of the DataReader and DataWriter histories.

Important: If the DataWriter reuses the same sample to publish new data, the DataReader loses access to the old data sample.

Note: The DataWriter can remove the sample from its history, and it will still be available on the DataReader, unless the same sample from the pool is reused to publish a new one.

Data acknowledgement

With data-sharing delivery, sample acknowledgment from the DataReader occurs the first time a sample is retrieved by the application (using `DataReader::read_next_sample()`, `DataReader::take_next_sample()`, or any of their variations). Once the data has been accessed by the application, the DataWriter is free to reuse that sample to publish new data. The DataReader detects when a sample has been reused and automatically removes it from its history.

This means that subsequent attempts to access the same sample from the DataReader may return no sample at all.

Blocking reuse of samples until acknowledged

With `KEEP_LAST_HISTORY_QOS` or `BEST_EFFORT_RELIABILITY_QOS` configurations, the DataWriter can remove samples from its history to add new ones, even if they were not acknowledged by the DataReader. In situations where the publishing rate is consistently faster than the rate at which the DataReader can process the samples, this can lead to every sample being reused before the application has a chance to process it, thus blocking the communication at application level.

In order to avoid this situation, the samples in the preallocated pool are never reused unless they have been acknowledged, i.e., they have been processed by the application at least once. If there is no reusable sample in the pool, the writing operation in the DataWriter will be blocked until one is available or until `max_blocking_time` is reached.

Note that the DataWriter history is not affected by this behavior, samples will be removed from the history by standard rules. Only the reuse of pool samples is affected. This means that the DataWriter history can be empty and the write operation be still blocked because all samples in the pool are unacknowledged.

The chance of the DataWriter blocking on a write operation can be reduced using `extra_samples`. This will make the pool to allocate more samples than the history size, so that the DataWriter has more chances to get a free sample, while the DataReader can still access samples that have been removed from the DataWriter history.

7.21.6 Intra-process delivery

eProsima Fast DDS allows to speed up communications between entities within the same process by avoiding any of the overhead involved in the transport layer. Instead, the *Publisher* directly calls the reception functions of the *Subscriber*. This not only avoids the copy or send operations of the transport, but also ensures the message is received by the Subscriber, avoiding the acknowledgement mechanism.

This feature is enabled by default, and can be configured using *XML profiles* (see *Intra-process delivery profiles*). Currently the following options are available:

- **INTRAPROCESS_OFF**: The feature is disabled.
- **INTRAPROCESS_USER_DATA_ONLY**: Discovery metadata keeps using ordinary transport.
- **INTRAPROCESS_FULL**: Default value. Both user data and discovery metadata using Intra-process delivery.

XML

```
<library_settings>
  <intraprocess_delivery>FULL</intraprocess_delivery> <!-- OFF | USER_DATA_ONLY |   

  ↪ FULL -->
</library_settings>
```

GUID Prefix considerations for intra-process delivery

Fast DDS utilizes the *DomainParticipant*'s `GuidPrefix_t` to identify peers running in the same process. Two participants with identical 8 first bytes on the `GuidPrefix_t` are considered to be running in the same process, and therefore intra-process delivery is used. `is_on_same_process_as()` API is provided to check this condition. This mechanism works out-of-the-box when letting Fast DDS set the GUID prefixes for the created *DomainParticipants*. However, special consideration is required when setting the `GuidPrefix_t` manually, either programmatically or when using XML.

Important: Fast DDS assigns GUID prefixes considering several host parameters. Among them, the network interfaces enabled. Thus, if at runtime, the network interfaces change, any new *DomainParticipant* will have a different GUID prefix and will be considered to be running on another host.

C++ - ASCII format.

```
eprosima::fastrtps::rtps::GuidPrefix_t guid_prefix;
guid_prefix.value[0] = eprosima::fastrtps::rtps::octet(0x77);
guid_prefix.value[1] = eprosima::fastrtps::rtps::octet(0x73);
guid_prefix.value[2] = eprosima::fastrtps::rtps::octet(0x71);
guid_prefix.value[3] = eprosima::fastrtps::rtps::octet(0x85);
guid_prefix.value[4] = eprosima::fastrtps::rtps::octet(0x69);
guid_prefix.value[5] = eprosima::fastrtps::rtps::octet(0x76);
guid_prefix.value[6] = eprosima::fastrtps::rtps::octet(0x95);
guid_prefix.value[7] = eprosima::fastrtps::rtps::octet(0x66);
guid_prefix.value[8] = eprosima::fastrtps::rtps::octet(0x65);
guid_prefix.value[9] = eprosima::fastrtps::rtps::octet(0x82);
guid_prefix.value[10] = eprosima::fastrtps::rtps::octet(0x82);
guid_prefix.value[11] = eprosima::fastrtps::rtps::octet(0x79);
```

```
DomainParticipantQos participant_qos;
participant_qos.wire_protocol().prefix = guid_prefix;
```

C++ - Extraction operator.

```
DomainParticipantQos participant_qos;
std::istringstream("77.73.71.85.69.76.95.66.65.82.82.79") >> participant_qos.wire_  

↪ protocol().prefix;
```

XML

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

(continues on next page)

(continued from previous page)

```

    <rtps>
      <prefix>77.73.71.85.69.76.95.66.65.82.82.79</prefix>
    </rtps>
  </participant>
</profiles>

```

7.21.7 TLS over TCP

Warning: This documentation assumes the reader has basic knowledge of TLS concepts since terms like Certificate Authority (CA), Private Key, *Rivest–Shamir–Adleman* (RSA) cryptosystem, and Diffie–Hellman encryption protocol are not explained in detail.

Fast DDS allows configuring TCP Transports to use TLS (Transport Layer Security). In order to set up TLS, the *TCPTransportDescriptor* must have its `apply_security` data member set to `true`, and its `tls_config` data member filled with the desired configuration on the *TCPTransportDescriptor*. The following is an example of configuration of TLS on the *TCP server*.

C++

```

DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tls_transport = std::make_shared<TCPv4TransportDescriptor>();
tls_transport->sendBufferSize = 9216;
tls_transport->receiveBufferSize = 9216;
tls_transport->add_listener_port(5100);

// Create the TLS configuration
using TLSOptions =
    eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSOptions;
tls_transport->apply_security = true;
tls_transport->tls_config.password = "test";
tls_transport->tls_config.cert_chain_file = "server.pem";
tls_transport->tls_config.private_key_file = "serverkey.pem";
tls_transport->tls_config.tmp_dh_file = "dh2048.pem";
tls_transport->tls_config.add_option(TLSOptions::DEFAULT_WORKAROUNDS);
tls_transport->tls_config.add_option(TLSOptions::SINGLE_DH_USE);
tls_transport->tls_config.add_option(TLSOptions::NO_SSLV2);

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tls_transport);

```

XML

```

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

```

(continues on next page)

(continued from previous page)

```

<tls>
  <password>test</password>
  <private_key_file>serverkey.pem</private_key_file>
  <cert_chain_file>server.pem</cert_chain_file>
  <tmp_dh_file>dh2048.pem</tmp_dh_file>
  <options>
    <option>DEFAULT_WORKAROUNDS</option>
    <option>SINGLE_DH_USE</option>
    <option>NO_SSLV2</option>
  </options>
</tls>
<sendBufferSize>9216</sendBufferSize>
<receiveBufferSize>9216</receiveBufferSize>
<listening_ports>
  <port>5100</port>
</listening_ports>
</transport_descriptor>
</transport_descriptors>

<participant profile_name="TLSServerParticipant">
  <rtps>
    <userTransports>
      <transport_id>tls_transport_server</transport_id>
    </userTransports>
  </rtps>
</participant>
</profiles>

```

The corresponding configuration on the *TCP client* is shown in the following example.

C++

```

DomainParticipantQos qos;

// Set initial peers.
Locator_t initial_peer_locator;
initial_peer_locator.kind = LOCATOR_KIND_TCPv4;
IPLocator::setIPv4(initial_peer_locator, "192.168.1.10");
initial_peer_locator.port = 5100;
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer_locator);

// Create a descriptor for the new transport.
auto tls_transport = std::make_shared<TCPv4TransportDescriptor>();

// Create the TLS configuration
using TLSOptions =
    ↪ eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSOptions;
using TLSVerifyMode =
    ↪ eprosima::fastdds::rtps::TCPTransportDescriptor::TLSConfig::TLSVerifyMode;
tls_transport->apply_security = true;
tls_transport->tls_config.verify_file = "ca.pem";
tls_transport->tls_config.add_verify_mode(TLSVerifyMode::VERIFY_PEER);
tls_transport->tls_config.add_verify_mode(TLSVerifyMode::VERIFY_FAIL_IF_NO_PEER_CERT);

```

(continues on next page)

(continued from previous page)

```

tls_transport->tls_config.add_option(TLSOptions::DEFAULT_WORKAROUNDS);
tls_transport->tls_config.add_option(TLSOptions::SINGLE_DH_USE);
tls_transport->tls_config.add_option(TLSOptions::NO_SSLV2);
tls_transport->tls_config.server_name = "my_server.com";

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tls_transport);

```

XML

```

<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <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>
        <server_name>my_server.com</server_name>
      </tls>
    </transport_descriptor>
  </transport_descriptors>

  <participant profile_name="TLSClientParticipant">
    <rtps>
      <userTransports>
        <transport_id>tls_transport_client</transport_id>
      </userTransports>
      <builtin>
        <initialPeersList>
          <locator>
            <tcpv4>
              <address>192.168.1.10</address>
              <physical_port>5100</physical_port>
            </tcpv4>
          </locator>
        </initialPeersList>
      </builtin>
    </rtps>
  </participant>
</profiles>

```

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

Member	Data type	Default	Description
password	string	""	Password of the private_key_file or rsa_private_key_file.
private_key_file	string	""	Path to the private key certificate file.
rsa_private_key_file	string	""	Path to the private key RSA certificate file.
cert_chain_file	string	""	Path to the public certificate chain file.
tmp_dh_file	string	""	Path to the Diffie-Hellman parameters file.
verify_file	string	""	Path to the CA (Certification- Authority) file.
verify_mode	TLSVerifyMode	TLSVerifyMode::NONE	Establishes the verification mode mask. See TLS Verification Mode .
options	TLSOptions	TLSOptions::NONE	Establishes the SSL Context options mask. See TLS Options .
verify_paths	vector<string>	Empty vector	Paths where the system will look for verification files.
verify_depth	int32_t	-1	Maximum allowed depth for verifying intermediate certificates.
default_verify	bool	false	Look for verification files on the default paths.
handshake_role	TLSHandShakeRole	TLSHandShakeRole::SERVER	Role that the transport will take on handshaking. See TLS Handshake Role .
server_name	string	""	Server name or host name required in case Server Name Indication (SNI) is used.

Note: *Fast DDS* uses the [Boost.Asio](#) library to handle TLS secure connections. These data members are used to build the asio library context, and most of them are mapped directly into this context without further manipulation. You can find more information about the implications of each member on the [Boost.Asio context](#) documentation.

TLS Verification Mode

The verification mode defines how the peer node will be verified. The following table describes the available verification options. Several verification options can be combined in the same `TCPTransportDescriptor` using the `add_verify_mode()` member function.

Value	Description
<code>TLSVerifyMode::VERIFY_NONE</code>	Perform no verification.
<code>TLSVerifyMode::VERIFY_PEER</code>	Perform verification of the peer.
<code>TLSVerifyMode::VERIFY_FAIL_IF_NO_CERT</code>	Fail verification if the peer has no certificate. Ignored unless <code>TLSVerifyMode::VERIFY_PEER</code> is also set.
<code>TLSVerifyMode::VERIFY_CLIENT_ONCE</code>	Do not request client certificate on renegotiation. Ignored unless <code>TLSVerifyMode::VERIFY_PEER</code> 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
<code>TLSOptions::DEFAULT_WORKAROUNDS</code>	Implement various bug workarounds. See Boost.Asio context .
<code>TLSOptions::NO_COMPRESSION</code>	Disable compression.
<code>TLSOptions::NO_SSLV2</code>	Disable SSL v2.
<code>TLSOptions::NO_SSLV3</code>	Disable SSL v3.
<code>TLSOptions::NO_TLSV1</code>	Disable TLS v1.
<code>TLSOptions::NO_TLSV1_1</code>	Disable TLS v1.1.
<code>TLSOptions::NO_TLSV1_2</code>	Disable TLS v1.2.
<code>TLSOptions::NO_TLSV1_3</code>	Disable TLS v1.3.
<code>TLSOptions::SINGLE_DH_USE</code>	Always create a new key when using <i>Diffie-Hellman</i> parameters.

TLS Handshake Role

The role can take the following values:

Value	Description
<code>TLShandShakeRole::DEFAULT</code>	Configured as client if connector, and as server if acceptor
<code>TLShandShakeRole::CLIENT</code>	Configured as client.
<code>TLShandShakeRole::SERVER</code>	Configured as server.

7.21.8 Listening Locators

Listening *Locators* are used to receive incoming traffic on the *DomainParticipant*. These Locators can be classified according to the communication type and to the nature of the data.

According to the communication type we have:

- **Multicast locators:** Listen to multicast communications.
- **Unicast locators:** Listen to unicast communications.

According to the nature of the data we have:

- **Metatraffic locators:** Used to receive metatraffic information, usually used by built-in endpoints to perform discovery.
- **User locators:** Used by the endpoints created by the user to receive user *Topic* data changes.

Applications can *provide their own Listening Locators*, or use the *Default Listening Locators* provided by *eProsima Fast DDS*.

Adding Listening Locators

Users can add custom Listening Locators to the DomainParticipant using the *DomainParticipantQos*. Depending on the field where the Locator is added, it will be treated as a *multicast*, *unicast*, *user* or *metatraffic* Locator.

Note: Both UDP and TCP unicast Locators support to have a null address. In that case, *Fast DDS* automatically gets and uses local network addresses.

Note: Both UDP and TCP Locators support to have a zero port. In that case, *Fast DDS* automatically calculates and uses well-known ports for that type of traffic. See *Well Known Ports* for details about the well-known ports.

Warning: TCP does not support multicast scenarios, so the network architecture must be carefully planned.

Metatraffic Multicast Locators

Users can set their own metatraffic multicast locators within the *WireProtocolConfigQos*: `builtin.metatrafficMulticastLocatorList`.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22222 over multicast address 239.255.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 239, 255, 0, 1);
locator.port = 22222;

// Add the locator to the DomainParticipantQos
qos.wire_protocol().builtin.metatrafficMulticastLocatorList.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="CustomMetatrafficMulticastParticipant">
    <rtps>
      <builtin>
        <metatrafficMulticastLocatorList>
          <!-- LOCATOR_LIST -->
          <locator>
            <udp4>
              <address>239.255.0.1</address>
              <port>22222</port>
            </udp4>
          </locator>
        </metatrafficMulticastLocatorList>
      </builtin>
    </rtps>
```

(continues on next page)

(continued from previous page)

```
</participant>
</profiles>
```

Metatraffic Unicast Locators

Users can set their own metatraffic unicast locators within the *WireProtocolConfigQos*: `builtin.metatrafficUnicastLocatorList`.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22223 over address 192.168.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 0, 1);
locator.port = 22223;

// Add the locator to the DomainParticipantQos
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <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 within the *WireProtocolConfigQos::default_multicast_locator_list*.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22224 over multicast address 239.255.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 239, 255, 0, 1);
locator.port = 22224;

// Add the locator to the DomainParticipantQos
qos.wire_protocol().default_multicast_locator_list.push_back(locator);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="CustomUsertrafficMulticastParticipant">
    <rtps>
      <defaultMulticastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
          <udp4>
            <address>239.255.0.1</address>
            <port>22224</port>
          </udp4>
        </locator>
      </defaultMulticastLocatorList>
    </rtps>
  </participant>
</profiles>
```

User-traffic Unicast Locators

Users can set their own user-traffic unicast locators within the *WireProtocolConfigQos::default_unicast_locator_list*.

C++

```
DomainParticipantQos qos;

// This locator will open a socket to listen network messages
// on UDPv4 port 22225 over address 192.168.0.1
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 0, 1);
locator.port = 22225;
```

(continues on next page)

(continued from previous page)

```
// 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">
  <participant profile_name="CustomUsertrafficUnicastParticipant">
    <rtps>
      <defaultUnicastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
          <udp4>
            <address>192.168.0.1</address>
            <port>22225</port>
          </udp4>
        </locator>
      </defaultUnicastLocatorList>
    </rtps>
  </participant>
</profiles>
```

Default Listening Locators

If the application does not define any Listening Locators, *eProsima Fast DDS* automatically enables a set of listening UDPv4 locators by default. This allows out-of-the-box communication in most cases, without the need of further configuring the *Transport Layer*.

- If the application does not define any *metatraffic* Locator (neither *unicast* nor *multicast*), *Fast DDS* enables one *multicast* Locator that will be used during *Discovery*, and one *unicast* Locator that will be used for peer-to-peer communication with already discovered DomainParticipants.
- If the application does not define any *user-traffic* Locator (neither *unicast* nor *multicast*), *Fast DDS* enables one *unicast* Locator that will be used for peer-to-peer communication of *Topic* data.

For example, it is possible to prevent *multicast* traffic adding a single *metatraffic unicast* Locator as described in *Disabling all Multicast Traffic*.

Default Listening Locators always use *Well Known Ports*.

Well Known Ports

The *DDSI-RTPS V2.2* standard (Section 9.6.1.1) defines a set of rules to calculate well-known ports for default Locators, so that DomainParticipants can communicate with these default Locators. Well-known ports are also selected automatically by *Fast DDS* when a Locator is configured with port number 0.

Well-known ports are calculated using the following predefined rules:

Table 10: Rules to calculate ports on default listening locators

Traffic type	Well-known port expression
Metatraffic multicast	$PB + DG * domainId + offsetd0$
Metatraffic unicast	$PB + DG * domainId + offsetd1 + PG * participantId$
User multicast	$PB + DG * domainId + offsetd2$
User unicast	$PB + DG * domainId + offsetd3 + PG * participantId$

The values used in these rules are explained on the following table. The default values can be modified using the `port` member of the *WireProtocolConfigQos* on the *DomainParticipantQos*.

Table 11: Values used in the rules to calculate well-known ports

Symbol	Meaning	Default value	QoS field
DG	DomainID gain	250	<code>wire_protocol().port.domainIDGain</code>
PG	ParticipantId gain	2	<code>wire_protocol().port.participantIDGain</code>
PB	Port Base number	7400	<code>wire_protocol().port.portBase</code>
<code>offsetd0</code>	Additional offset	0	<code>wire_protocol().port.offsetd0</code>
<code>offsetd1</code>	Additional offset	10	<code>wire_protocol().port.offsetd1</code>
<code>offsetd2</code>	Additional offset	1	<code>wire_protocol().port.offsetd2</code>
<code>offsetd3</code>	Additional offset	11	<code>wire_protocol().port.offsetd3</code>

7.21.9 Announced Locators

In order for communication to take place, DDS entities need to exchange the list of addresses and ports where they can be reached. Apart from the *default announced locators*, which correspond to addresses of the interfaces in the host where the application is running, the user can configure *additional locators* with addresses and ports on other networks, when routing rules have been correspondingly set up.

Default Announced Locators

The default list of announced locators will be constructed from the *listening locators*, as follows:

- If the address field of the locator is a null address (i.e. 0.0.0.0 for UDPv4), a locator of the same kind and port will be announced for each of the addresses of the network interfaces of the host.
- If the address field of the locator is not a null address, a single locator with that address will be announced.

External Locators

The user can configure a set of external locators for each of the lists of unicast locators:

- `builtin.metatraffic_external_unicast_locators` on *WireProtocolConfigQos*
- `default_external_unicast_locators` on *WireProtocolConfigQos*
- `external_unicast_locators` on *RTPSEndpointQos*

An external locator is made up of the standard locator fields (kind, address, and port), plus the following attributes:

- An *externality* that indicates the number of hops from the host where the application is running to the LAN represented by the external locator.
- A *cost* indicating the communication cost relative to other locators on the same externality level.

- A *mask* with the number of significant bits on the LAN represented by the external locator.

Externality levels

The main purpose of the external locators is to enable communication across different levels of interconnected LANs. Communication will be performed using the locators of the innermost LAN available.

As an example, consider a network topology where the application is running on a host connected to a LAN of an office, which in turn connects to a LAN for all the offices in the same floor, which in turn connects to a LAN for the building.

With the default configuration, communication will only occur between hosts on the LAN for the office. This is considered the externality level 0, which is reserved for the LANs directly connected to the network interfaces of the host where the application is running. This is the externality level that will be used on the *matching algorithm* for the *default announced locators*. The floor LAN will be configured as externality level 1, whereas the building LAN will be configured as externality level 2.

Note that in order for the communication to be successful, routing rules should most probably need to be added to the different network routers.

Important: Externality level 0 is automatically populated by Fast DDS and cannot be configured by the application.

Matching algorithm

When a remote entity is discovered, its list of announced locators is processed to select the ones on the innermost externality level where the communication can be established. The highest externality level is checked first.

If the discovered addresses for one level are equal to the ones announced by the local entity, it means they are on the same host at that level, and the algorithm proceeds to an inner level. If the discovered addresses are not equal to the ones announced by the local entity, processing stops at the current level.

When the externality level on which the communication will be established has been decided, the algorithm will:

- Remove locators that match with addresses on any other externality level.
- Keep locators that match with the selected externality level.
- For the locators with an address that does not match with any of the locators announced by the local entity:
 - Keep them when `ignore_non_matching_locators` is `false` (default behavior)
 - Remove them when `ignore_non_matching_locators` is `true`

Additional considerations

Since using external locators increases the number of locators announced, the *allocation limits for locators discovery* would need to be adjusted for your application.

Participants running on the same host, but using different addresses on their `builtin.metatraffic_external_unicast_locators` will discard shared memory transport locators. Data sharing communication is not affected by this limitation.

Warning: Interface whitelist will be deprecated in future versions of Fast DDS. The use of the new *interfaces allowlist* is recommended.

7.21.10 Interface Whitelist

Using *Fast DDS*, it is possible to limit the network interfaces used by *TCP Transport* and *UDP Transport*. This is achieved by adding the interfaces to the `interfaceWhiteList` field in the *TCPTransportDescriptor* or *UDPTransportDescriptor*. Thus, the communication interfaces used by the *DomainParticipants* whose *TransportDescriptorInterface* defines an `interfaceWhiteList` is limited to the interfaces' addresses defined in that list, therefore avoiding the use of the rest of the network interfaces available in the system. The interfaces in `interfaceWhiteList` can be specified both by IP address or interface name. For example:

- Interface whitelist filled with IP address:

C++

```
DomainParticipantQos qos;

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

// Add loopback to the whitelist by IP address
tcp_transport->interfaceWhiteList.emplace_back("127.0.0.1");

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tcp_transport);

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

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <transport_descriptors>
    <transport_descriptor>
      <transport_id>CustomTcpTransportWhitelistAddress</transport_id>
      <type>TCPv4</type>
      <interfaceWhiteList>
        <address>127.0.0.1</address>
      </interfaceWhiteList>
    </transport_descriptor>
  </transport_descriptors>

  <participant profile_name="CustomTcpTransportWhitelistAddressParticipant">
    <rtps>
      <useBuiltinTransports>>false</useBuiltinTransports>
      <userTransports>
        <transport_id>CustomTcpTransportWhitelistAddress</transport_id>
      </userTransports>
    </rtps>
  </participant>
</profiles>
```

- Interface whitelist filled with interface names:

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<TCpV4TransportDescriptor>();

// Add loopback to the whitelist by interface name
tcp_transport->interfaceWhiteList.emplace_back("lo");

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(tcp_transport);

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

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <transport_descriptors>
    <transport_descriptor>
      <transport_id>CustomTcpTransportWhitelistName</transport_id>
      <type>TCpV4</type>
      <interfaceWhiteList>
        <interface>lo</interface>
      </interfaceWhiteList>
    </transport_descriptor>
  </transport_descriptors>

  <participant profile_name="CustomTcpTransportWhitelistNameParticipant">
    <rtps>
      <useBuiltinTransports>false</useBuiltinTransports>
      <userTransports>
        <transport_id>CustomTcpTransportWhitelistName</transport_id>
      </userTransports>
    </rtps>
  </participant>
</profiles>
```

Important: If none of the values in the transport descriptor's whitelist match the interfaces on the host, then all the interfaces in the whitelist are filtered out and therefore no communication will be established through that transport.

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

Warning: The configuration options described in this section apply to network interfaces. Therefore, it is only available on *TCP Transport* and *UDP Transport*.

7.21.11 Interfaces configuration

By default, *Fast DDS* makes use of all active network interfaces found in the system to communicate (note: applies to *UDP Transport* and *TCP Transport*). However, it is possible for a user to indicate a specific set of network interfaces to be used by the library, and/or configure some of them in a specific manner.

Netmask filtering

The standard behaviour in *Fast-DDS* is to attempt data transmission to any remote locator for which a compatible transport (based on *kind*) is registered. This may result in a non-optimum resource utilization, as messages could be sent from an interface to an unreachable destination given a particular network architecture. In this situation, a user may decide to enable the netmask filtering feature, which would prevent this behavior by only sending data from a network interface to remote locators within the same subnetwork.

This configuration option can be set at participant, transport and interface levels, and its possible values are:

Value	Description
ON	Enable netmask filtering.
OFF	Disable netmask filtering.
AUTO	Use container's netmask filter configuration.

An AUTO netmask filter configuration means that its effective value will be given by that of its “container”, which in the case of an *allowlist* entry would be the *transport descriptor* where it is included, and in the case of a transport descriptor would be the participant where it is registered.

However not all configurations are valid; this is, for example, a transport's netmask filter configuration cannot be OFF if it is ON for the participant where this transport is registered. Likewise, the netmask filter configuration for an allowlist entry cannot be ON if it is OFF for the transport descriptor where this allowlist is defined.

Note: Due to implementation details, it is required to set *ignore_non_matching_locators* to *true* (see *Matching algorithm*) both in *participants* and *endpoints* when enabling the netmask filtering feature at participant or transport level without defining an *allowlist* or *blocklist*.

Additional considerations need to be taken into account when using netmask filtering in combination with *external locators*. In particular, it is not possible to enable netmask filtering in all entries of an allowlist when a set of local external locators (with an *externality* greater than 0) is defined for a *participant* or *endpoint*. The reason for this is that a matching remote external locator would then (most likely) be effectively ignored, as no network interface would be able to reach it according to its network mask.

Netmask filtering can be enabled at participant level both via C++ API or XML configuration:

C++

```
// Configure netmask filtering at participant level
qos.transport().netmask_filter = NetmaskFilterKind::AUTO;
qos.wire_protocol().ignore_non_matching_locators = true; // Required if not defining an
↳ allowlist or blocklist
```

XML

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

(continues on next page)

(continued from previous page)

```

<participant profile_name="CustomTcpParticipantNetmaskFilterParticipant">
  <rtps>
    <ignore_non_matching_locators>true</ignore_non_matching_locators>
    <netmask_filter>ON</netmask_filter>
  </rtps>
</participant>
</profiles>

```

For socket (UDP/TCP) transport descriptors:

C++

```

// Create a descriptor for the new transport.
auto udp_transport = std::make_shared<UD Pv4TransportDescriptor>();

// Configure netmask filtering at transport level
udp_transport->netmask_filter = NetmaskFilterKind::AUTO;
qos.wire_protocol().ignore_non_matching_locators = true; // Required if not defining an
↳ allowlist or blocklist

```

XML

```

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

  <participant profile_name="CustomTcpTransportNetmaskFilterParticipant">
    <rtps>
      <useBuiltinTransports>>false</useBuiltinTransports>
      <userTransports>
        <transport_id>CustomTcpTransportNetmaskFilter</transport_id>
      </userTransports>
      <ignore_non_matching_locators>true</ignore_non_matching_locators>
    </rtps>
  </participant>
</profiles>

```

See *Allowlist* to learn how to configure netmask filtering for specific network devices.

Allowlist

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 to the allowlist field in the *TCPTransportDescriptor* or *UDPTransportDescriptor*. Thus, the communication interfaces used by the *DomainParticipants* whose *TransportDescriptorInterface* defines an allowlist is limited to the interfaces defined in that list, therefore avoiding the use of the rest of the network interfaces available in the system. The interfaces in allowlist can be specified both by IP address or interface name. Additionally, each entry added to the allowlist may specify a *netmask filter* configuration value (AUTO by default).

For example:

C++

```
DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto udp_transport = std::make_shared<UDPV4TransportDescriptor>();

// Add allowed interface by device name
udp_transport->interface_allowlist.emplace_back("eth0", NetmaskFilterKind::OFF);

// Add allowed interface by IP address (using default netmask filter AUTO)
udp_transport->interface_allowlist.emplace_back("127.0.0.1");

// Add allowed interface with explicit AllowedNetworkInterface construction
AllowedNetworkInterface another_allowed_interface("docker0", NetmaskFilterKind::OFF);
udp_transport->interface_allowlist.emplace_back(another_allowed_interface);

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(udp_transport);

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

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
  <transport_descriptors>
    <transport_descriptor>
      <transport_id>CustomTcpTransportAllowlist</transport_id>
      <type>TCPv4</type>
      <interfaces>
        <allowlist>
          <interface name="wlp59s0" netmask_filter="ON"/>
          <interface name="192.168.1.10" netmask_filter="OFF"/>
        </allowlist>
      </interfaces>
    </transport_descriptor>
  </transport_descriptors>

  <participant profile_name="CustomTcpTransportAllowlistParticipant">
    <rtps>
      <useBuiltinTransports>false</useBuiltinTransports>
      <userTransports>
```

(continues on next page)

(continued from previous page)

```

        <transport_id>CustomTcpTransportAllowlist</transport_id>
    </userTransports>
</rtps>
</participant>
</profiles>

```

Important: If none of the values in the transport descriptor's allowlist match the interfaces on the host, then all the interfaces in the allowlist are filtered out and therefore no communication will be established through that transport.

Blocklist

Apart from defining a list of *allowed interfaces*, it is also possible to define a list of interfaces that should be blocked. This is accomplished through the `blocklist` field present in the *TCPTransportDescriptor* or *UDPTransportDescriptor*.

Note: This list takes precedence over the allowlist, so if a network interface is in both lists, it will be blocked.

The interfaces in `blocklist` can be specified both by IP address or interface name.

For example:

C++

```

DomainParticipantQos qos;

// Create a descriptor for the new transport.
auto udp_transport = std::make_shared<UDPV4TransportDescriptor>();

// Add blocked interface by device name
udp_transport->interface_blocklist.emplace_back("docker0");

// Add blocked interface by IP address
udp_transport->interface_blocklist.emplace_back("127.0.0.1");

// Add blocked interface with explicit BlockedNetworkInterface construction
BlockedNetworkInterface another_blocked_interface("eth0");
udp_transport->interface_blocklist.emplace_back(another_blocked_interface);

// Link the Transport Layer to the Participant.
qos.transport().user_transports.push_back(udp_transport);

// Avoid using the builtin transports
qos.transport().use_builtin_transports = false;

```

XML

```

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

```

(continues on next page)

(continued from previous page)

```

    <transport_id>CustomTcpTransportBlocklist</transport_id>
    <type>TCPv4</type>
    <interfaces>
        <blocklist>
            <interface name="127.0.0.1"/>
            <interface name="docker0"/>
        </blocklist>
    </interfaces>
</transport_descriptor>
</transport_descriptors>

<participant profile_name="CustomTcpTransportBlocklistParticipant">
    <rtps>
        <useBuiltinTransports>false</useBuiltinTransports>
        <userTransports>
            <transport_id>CustomTcpTransportBlocklist</transport_id>
        </userTransports>
    </rtps>
</participant>
</profiles>

```

7.21.12 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;

```

(continues on next page)

(continued from previous page)

```
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">
  <participant profile_name="disable_multicast" is_default_profile="true">
    <rtps>
      <builtin>
        <metatrafficUnicastLocatorList>
          <locator>
            <udp4>
              <address>192.168.0.1</address>
            </udp4>
          </locator>
        </metatrafficUnicastLocatorList>
        <initialPeersList>
          <locator>
            <udp4>
              <address>192.168.0.1</address>
            </udp4>
          </locator>
        </initialPeersList>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

7.22 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.

7.22.1 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*):

7.22.2 PERSISTENCE:SQLITE3 built-in plugin

This plugin provides persistence through a local database file using *SQLite3* API. To activate the plugin, `dds.persistence.plugin` property must be added to the `PropertyPolicyQos` of the `DomainParticipant`, `DataWriter`, or `DataReader` with value `builtin.SQLITE3`. Furthermore, `dds.persistence.sqlite3.filename` property must be added to the entities `PropertyPolicyQos`, specifying the database file name. These properties are summarized in the following table:

Table 12: **Persistence::SQLITE3 configuration properties**

Property name	Property value
<code>dds.persistence.plugin</code>	<code>builtin.SQLITE3</code>
<code>dds.persistence.sqlite3.filename</code>	Name of the file used for persistent storage. Default value: <code>persistence.db</code>

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 not exist or is invalid.

7.22.3 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";

TypeDescriptor::_ref_type struct_type_descriptor {traits<TypeDescriptor>::make_shared()};
struct_type_descriptor->kind(TK_STRUCTURE);
struct_type_descriptor->name(topic_type_name);
DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_instance()->
    create_type(struct_type_
    descriptor)};

// The type consists of two members, and index and a message. Add members to the struct.
MemberDescriptor::_ref_type index_member_descriptor {traits<MemberDescriptor>::make_
    shared()};
index_member_descriptor->name("index");
```

(continues on next page)

(continued from previous page)

```

index_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
    get_primitive_type(TK_UINT32));
struct_builder->add_member(index_member_descriptor);

MemberDescriptor::_ref_type message_member_descriptor {traits<MemberDescriptor>::make_
    ↪shared()};
message_member_descriptor->name("message");
message_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
    create_string_type(static_cast<uint32_t>(LENGTH_UNLIMITED))->build());
struct_builder->add_member(message_member_descriptor);

// Build the type
DynamicType::_ref_type struct_type {struct_builder->build()};

// Create type support and register the type
TypeSupport type_support(new DynamicPubSubType(struct_type));
type_support.register_type(participant);

// Create data sample a populate data. This is to be used when calling `writer->write()`
DynamicData::_ref_type dyn_helloworld {DynamicDataFactory::get_instance()->create_
    ↪data(struct_type)};

dyn_helloworld->set_uint32_value(0, 0);
dyn_helloworld->set_string_value(1, "HelloWorld");
/
↪ *****
* END CREATE TYPE AND TYPE SUPPORT
↪ *****

// Create a topic
Topic* topic = participant->create_topic("persistence_topic_name", topic_type_name,
    ↪TOPIC_QOS_DEFAULT);

// Create a publisher and a subscriber with default QoS
Publisher* publisher = participant->create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);
Subscriber* subscriber = participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT, nullptr);

// Configure DataWriter's durability and persistence GUID so it can use the persistence_
    ↪service
DataWriterQos dwqos = DATAWRITER_QOS_DEFAULT;
dwqos.durability().kind = TRANSIENT_DURABILITY_QOS;
dwqos.properties().properties().emplace_back("dds.persistence.guid",
    "77.72.69.74.65.72.5f.70.65.72.73.5f|67.75.69.64");
DataWriter* writer = publisher->create_datawriter(topic, dwqos);

// Configure DataReaders's durability and persistence GUID so it can use the persistence_
    ↪service
DataReaderQos drqos = DATAREADER_QOS_DEFAULT;
drqos.durability().kind = TRANSIENT_DURABILITY_QOS;
drqos.properties().properties().emplace_back("dds.persistence.guid",
    "72.65.61.64.65.72.5f.70.65.72.73.5f|67.75.69.64");

```

(continues on next page)

(continued from previous page)

```
DataReader* reader = subscriber->create_datareader(topic, drqos);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <!-- DomainParticipant configuration -->
    <participant profile_name="persistence_service_participant">
      <rtps>
        <propertiesPolicy>
          <properties>
            <!-- Select persistence plugin -->
            <property>
              <name>dds.persistence.plugin</name>
              <value>builtin.SQLITE3</value>
            </property>
            <!-- Database file name -->
            <property>
              <name>dds.persistence.sqlite3.filename</name>
              <value>persistence_service.db</value>
            </property>
          </properties>
        </propertiesPolicy>
      </rtps>
    </participant>

    <!-- DataWriter configuration -->
    <data_writer profile_name="persistence_service_data_writer">
      <qos>
        <!-- Set durability to TRANSIENT_DURABILITY_QOS -->
        <durability>
          <kind>TRANSIENT</kind>
        </durability>
      </qos>
      <propertiesPolicy>
        <properties>
          <!-- Persistence GUID -->
          <property>
            <name>dds.persistence.guid</name>
            <value>77.72.69.74.65.72.5f.70.65.72.73.5f|67.75.69.64</value>
          </property>
        </properties>
      </propertiesPolicy>
    </data_writer>

    <data_reader profile_name="persistence_service_data_reader">
      <qos>
        <!-- Set durability to TRANSIENT_DURABILITY_QOS -->
        <durability>
          <kind>TRANSIENT</kind>
        </durability>
      </qos>
    </data_reader>
  </profiles>
</dds>
```

(continues on next page)

(continued from previous page)

```

    </qos>
    <propertiesPolicy>
      <properties>
        <!-- Persistence GUID -->
        <property>
          <name>dds.persistence.guid</name>
          <value>72.65.61.64.65.72.5f.70.65.72.73.5f|67.75.69.64</value>
        </property>
      </properties>
    </propertiesPolicy>
  </data_reader>
</profiles>
</dds>

```

Note: For instructions on how to create DomainParticipants, DataReaders, and DataWriters, please refer to *Profile based creation of a DomainParticipant*, *Profile based creation of a DataWriter*, and *Profile based creation of a DataReader* respectively.

7.23 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.

7.23.1 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
<code>identity_ca</code>	URI to the X.509 v3 certificate of the Identity CA in PEM format. Supported URI schemes: file.
<code>identity_certificate</code>	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.
<code>identity_crl</code> (optional)	URI to a X.509 Certificate Revocation List (CRL). Supported URI schemes: file.
<code>private_key</code>	URI to access the private Private Key for the Participant. Supported URI schemes: file, <i>PKCS#11</i> .
<code>password</code> (optional)	A password used to decrypt the <i>private_key</i> . If the <i>password</i> property is not present, then the value supplied in the <i>private_key</i> property must contain the decrypted private key. The <i>password</i> property is ignored if the <i>private_key</i> is given in PKCS#11 scheme.

Note: All listed properties have “dds.sec.auth.builtin.PKI-DH.” prefix. For example: `dds.sec.auth.builtin.`

PKI-DH.identity_ca.

The following is an example of how to set the properties of DomainParticipantQoS for the DDS:Auth:PKI-DH plugin configuration.

C++

```

DomainParticipantQos pqos;

// Activate DDS:Auth:PKI-DH plugin
pqos.properties().properties().emplace_back("dds.sec.auth.plugin",
    "builtin.PKI-DH");

// Configure DDS:Auth:PKI-DH plugin
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_ca",
    "file://maincacert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_certificate",
    "file://partcert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_crl",
    "file://crl.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.private_key",
    "file://partkey.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.password",
    "domainParticipantPassword");

```

XML

```

<participant profile_name="secure_domainparticipant_conf_auth_plugin_xml_profile">
  <rtps>
    <propertiesPolicy>
      <properties>
        <!-- Activate DDS:Auth:PKI-DH plugin -->
        <property>
          <name>dds.sec.auth.plugin</name>
          <value>builtin.PKI-DH</value>
        </property>
        <!-- Configure DDS:Auth:PKI-DH plugin -->
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.identity_ca</name>
          <value>file://maincacert.pem</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.identity_certificate</name>
          <value>file://partcert.pem</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.identity_crl</name>
          <value>file://crl.pem</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.private_key</name>
          <value>file://partkey.pem</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.password</name>
          <value>domainParticipantPassword</value>
        </property>
      </properties>
    </propertiesPolicy>

```

Generation of X.509 certificates

An X.509 digital certificate is a document that has been encrypted and/or digitally signed according to [RFC 5280](#). The X.509 certificate refers to the Public Key Infrastructure (PKI) certificate of the [IETF](#), and specifies the standard formats for public-key certificates and a certification route validation algorithm. A simple way to generate these certificates for a proprietary PKI structure is through the [OpenSSL](#) toolkit. This section explains how to build a certificate infrastructure from the trusted CA certificate to the end-entity certificate, i.e. the DomainParticipant.

Generating the CA certificate for self-signing

First, since multiple certificates will need to be issued, one for each of the DomainParticipants, a dedicated CA is set up, and the CA's certificate is installed as the root key of all DomainParticipants. Thus, the DomainParticipants will accept all certificates issued by our own CA. To create a proprietary CA certificate, a configuration file must first be written with the CA information. An example of the CA configuration file is shown below. The OpenSSL commands shown in this example are compatible with both Linux and Windows Operating Systems (OS). However, all other commands are only compatible with Linux OS.

```
# File: maincaconf.cnf
# OpenSSL example Certificate Authority configuration file

#####
[ ca ]
default_ca = CA_default # The default ca section

#####
[ CA_default ]

dir = . # Where everything is kept
certs = $dir/certs # Where the issued certs are kept
crl_dir = $dir/crl # Where the issued crl are kept
database = $dir/index.txt # database index file.
unique_subject = no # Set to 'no' to allow creation of
                    # several certificates with same subject.
new_certs_dir = $dir

certificate = $dir/maincacert.pem # The CA certificate
serial = $dir/serial # The current serial number
crlnumber = $dir/crlnumber # the current crl number
                    # must be commented out to leave a V1 CRL
crl = $dir/crl.pem # The current CRL
private_key = $dir/maincakey.pem # The private key
RANDFILE = $dir/private/.rand # private random number file

name_opt = ca_default # Subject Name options
cert_opt = ca_default # Certificate field options

default_days= 1825 # how long to certify for
default_crl_days = 30 # how long before next CRL
default_md = sha256 # which md to use.
preserve = no # keep passed DN ordering

policy = policy_match
```

(continues on next page)

(continued from previous page)

```

# For the CA policy
[ policy_match ]
countryName = match
stateOrProvinceName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

[ req ]
prompt = no
#default_bits = 1024
#default_keyfile = privkey.pem
distinguished_name= req_distinguished_name
#attributes = req_attributes
#x509_extensions = v3_ca # The extensions to add to the self signed cert
string_mask = utf8only

[ req_distinguished_name ]
countryName = ES
stateOrProvinceName = MA
localityName = Tres Cantos
#.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 = eProxima
emailAddress = example@eprosima.com
commonName = DomainParticipantName
```

After writing the DomainParticipant certificate configuration file, next commands generate the X.509 certificate, using ECDSA, for a DomainParticipant.

```
openssl ecparam -name prime256v1 > ecdsaparam

openssl req -nodes -new \
  -newkey ec:ecdsaparam \
  -config partconf.cnf \
  -keyout partkey.pem \
  -out partreq.pem

openssl ca -batch -create_serial \
  -config maincaconf.cnf \
  -days 3650 \
  -in partreq.pem \
  -out partcert.pem
```

Generating the Certificate Revocation List (CRL)

Finally, the CRL is created. This is a list of the X.509 certificates revoked by the certificate issuing CA before they reach their expiration date. Any certificate that is on this list will no longer be trusted. To create a CRL using OpenSSL just run the following commands.

```
echo -ne '\00' > crlnumber

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

As an example, below is shown how to add the X.509 certificate of a DomainParticipant to the CRL.

```

openssl ca \
  -config maincaconf.cnf \
  -cert maincacert.pem \
  -keyfile maincakey.pem \
  -revoke partcert.pem

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

```

7.23.2 Access control plugin: DDS:Access:Permissions

The access control plugin provides the mechanisms and operations required for validating the *DomainParticipant* permissions. If the security module was activated at Fast DDS compilation, after a remote DomainParticipant is authenticated, its permissions need to be validated and enforced.

Access rights that each DomainParticipant has over a resource are defined using the access control plugin. For the proper functioning of a DomainParticipant in a DDS Domain, the DomainParticipant must be authorized to operate in that specific domain. The DomainParticipant is responsible for creating the *DataWriters* and *DataReaders* that communicate over a certain *Topic*. Hence, a DomainParticipant must have the permissions needed to create a Topic, to publish through its DataWriters under defined Topics, and to subscribe via its DataReaders to other Topics. Access control plugin can configure the Cryptographic plugin as its usage is based on the DomainParticipant's permissions.

The authentication plugin implemented in Fast DDS is referred to as “DDS:Access:Permissions”, in compliance with the *DDS Security* specification. This plugin is explained in detail below.

This builtin plugin provides access control using a permissions document signed by a trusted CA. The DDS:Access:Permissions plugin requires three documents for its configuration which contents are explained in detail below.

1. The Permissions CA certificate.
2. The Domain governance signed by the Permissions CA.
3. The DomainParticipant permissions signed by the Permissions CA.

The DDS:Access:Permissions authentication plugin, can be activated setting the *DomainParticipantQos* properties() `dds.sec.access.plugin` with the value `builtin.Access-Permissions`. The following table outlines the properties used for the DDS:Access:Permissions plugin configuration.

Property name	Property value
<code>permissions_ca</code>	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.
<code>governance</code>	URI to shared Governance Document signed by the Permissions CA in S/MIME format. Supported URI schemes: file.
<code>permissions</code>	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.

```

1 <dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:noNamespaceSchemaLocation="omg_shared_ca_domain_governance.xsd">
3   <domain_access_rules>
4     <domain_rule>
5       <domains>
6         <id_range>
7           <min>0</min>
8           <max>230</max>
9         </id_range>
10        </domains>
11        <allow_unauthenticated_participants>>false</allow_unauthenticated_
12↪participants>
13        <enable_join_access_control>>true</enable_join_access_control>
14        <discovery_protection_kind>ENCRYPT</discovery_protection_kind>
15        <liveliness_protection_kind>ENCRYPT</liveliness_protection_kind>
16        <rtps_protection_kind>ENCRYPT</rtps_protection_kind>
17        <topic_access_rules>
18          <topic_rule>
19            <topic_expression>HelloWorldTopic</topic_expression>
20            <enable_discovery_protection>>true</enable_discovery_protection>
21            <enable_liveliness_protection>>false</enable_liveliness_protection>
22            <enable_read_access_control>>true</enable_read_access_control>
23            <enable_write_access_control>>true</enable_write_access_control>
24            <metadata_protection_kind>ENCRYPT</metadata_protection_kind>
25            <data_protection_kind>ENCRYPT</data_protection_kind>
26          </topic_rule>
27        </topic_access_rules>
28      </domain_rule>
29    </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	<i>Allow Unauthenticated Participants</i>	<code><allow_unauthenticated_participants></code>	false
	<i>Enable Join Access Control</i>	<code><enable_join_access_control></code>	true
	<i>Discovery Protection Kind</i>	<code><discovery_protection_kind></code>	false
			true
			SIGN
			ENCRYPT
			NONE
	<i>Liveliness Protection Kind</i>	<code><liveliness_protection_kind></code>	SIGN
			ENCRYPT
			NONE
Section	<i>RTPS Protection Kind</i>	<code><rtps_protection_kind></code>	SIGN
			ENCRYPT
			NONE
	<i>Domains</i>	<code><domains></code>	<code><domains></code>
	<i>Topic Access Rules</i>	<code><topic_access_rules></code>	<code><topic_rule></code>

The following describes the possible configurations of each of the elements and sections listed above that are contained in the domain rules.

Domains

This element is delimited by the `<domains>` XML element tag. The value in this element identifies the collection of DDS Domains to which the rule applies. The `<domains>` element can contain:

- A single domain identifier:

```
<domains>
  <id>1</id>
</domains>
```

- A range of domain identifiers:

```
<domains>
  <id_range>
    <min>1</min>
    <max>10</max>
  </id_range>
</domains>
```

Or a combination of both, a list of domain identifiers and ranges of domain identifiers.

Allow Unauthenticated Participants

This element is delimited by the `<allow_unauthenticated_participants>` XML element tag. It indicates whether the matching of a DomainParticipant with a remote DomainParticipant requires authentication. The possible values for this element are:

- `false`: the DomainParticipant shall enforce the authentication of remote *DomainParticipants* and disallow matching those that cannot be successfully authenticated.
- `true`: the DomainParticipant shall allow matching other DomainParticipants (event if the remote DomainParticipant cannot authenticate) as long as there is not an already valid authentication with the same DomainParticipant's GUID.

In accordance with the [DDS Security specification](#), the following premises should be considered:

- Topics protected with `enable_read_access_control` or `enable_write_access_control` will not communicate regardless of the `allow_unauthenticated_participants` flag value.
- If *RTPS Protection Kind* is not `NONE` and *Allow Unauthenticated Participants* is enabled, the entity creation will fail with an error.
- Authentication is always attempted first regardless of `<allow_unauthenticated_participants>` configuration.

Enable Join Access Control

This element is delimited by the `<enable_join_access_control>` XML element tag. Indicates whether the matching of the participant with a remote DomainParticipant 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
<i>Topic expression</i>	<code><topic_expression></code>	Topic name
<i>Enable Discovery Protection</i>	<code><enable_discovery_protection></code>	false true
<i>Enable Liveliness Protection</i>	<code><enable_liveliness_protection></code>	false true
<i>Enable Read Access Control</i>	<code><enable_read_access_control></code>	false true
<i>Enable Write Access Control</i>	<code><enable_write_access_control></code>	false true
<i>Metadata protection Kind</i>	<code><metadata_protection_kind></code>	SIGN ENCRYPT NONE
<i>Data protection Kind</i>	<code><data_protection_kind></code>	SIGN ENCRYPT NONE

The topic expression within the rules selects a set of Topic names. The rule applies to any *DataReader* or *DataWriter* associated with a *Topic* whose name matches the Topic expression name. The topic access rules are evaluated in the same order as they appear within the `<topic_access_rules>` section. If multiple rules match, the first rule that matches is the only one that applies. If no matching `<topic_rule>` is found, the entity creation will fail.

Topic expression

This element is delimited by the `<topic_expression>` XML element tag. The value in this element identifies the set of Topic names to which the rule applies. The rule applies to any *DataReader* or *DataWriter* associated with a *Topic* whose name matches the value.

The Topic name expression syntax and matching shall use the syntax and rules of the POSIX `fnmatch()` function as specified in [IEEE 1003.1-2017](#).

Enable Discovery Protection

This element is delimited by the `<enable_discovery_protection>` XML element tag. Indicates whether the entity related discovery information shall go through the secure channel of endpoint discovery phase.

- `false`: the entity discovery information shall be sent by an unsecured channel of discovery.
- `true`: the information shall be sent by the secure channel.

Enable Liveliness Protection

This element is delimited by the `<enable_liveliness_protection>` XML element tag. Indicates whether the entity related liveliness information shall go through the secure channel of liveliness mechanism.

- `false`: the entity liveliness information shall be sent by an unsecured channel of liveliness.
- `true`: the information shall be sent by the secure channel.

Enable Read Access Control

This element is delimited by the `<enable_read_access_control>` XML element tag. Indicates whether read access to the Topic is protected.

- `false`: then local Subscriber creation and remote Subscriber matching can proceed without further access-control mechanisms imposed.
- `true`: they shall be checked using Access control plugin.

Enable Write Access Control

This element is delimited by the `<enable_write_access_control>` XML element tag. Indicates whether write access to the Topic is protected.

- `false`: then local Publisher creation and remote Publisher matching can proceed without further access-control mechanisms imposed.
- `true`: they shall be checked using Access control plugin.

Metadata Protection Kind

This element is delimited by the `<metadata_protection_kind>` XML element tag. Indicates whether the entity's RTPS submessages shall be encrypted by the Cryptographic plugin.

- `NONE`: shall not be protected.
- `SIGN`: shall be protected by MAC.
- `ENCRYPT`: shall be encrypted.

Data Protection Kind

This element is delimited by the `<data_protection_kind>` XML element tag. Indicates whether the data payload shall be encrypted by the Cryptographic plugin.

- `NONE`: shall not be protected.
- `SIGN`: shall be protected by MAC.
- `ENCRYPT`: shall be encrypted.

Domain Governance XSD

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3   elementFormDefault="qualified" attributeFormDefault="unqualified">
4   <xs:element name="dds" type="DomainAccessRulesNode" />
5   <xs:complexType name="DomainAccessRulesNode">
6     <xs:sequence minOccurs="1" maxOccurs="1">
7       <xs:element name="domain_access_rules"
8         type="DomainAccessRules" />
9     </xs:sequence>
10  </xs:complexType>
```

(continues on next page)

(continued from previous page)

```

11 <xs:complexType name="DomainAccessRules">
12   <xs:sequence minOccurs="1" maxOccurs="unbounded">
13     <xs:element name="domain_rule" type="DomainRule" />
14   </xs:sequence>
15 </xs:complexType>
16 <xs:complexType name="DomainRule">
17   <xs:sequence minOccurs="1" maxOccurs="1">
18     <xs:element name="domains" type="DomainIdSet" />
19     <xs:element name="allow_unauthenticated_participants"
20       type="xs:boolean" />
21     <xs:element name="enable_join_access_control"
22       type="xs:boolean" />
23     <xs:element name="discovery_protection_kind"
24       type="ProtectionKind" />
25     <xs:element name="liveliness_protection_kind"
26       type="ProtectionKind" />
27     <xs:element name="rtps_protection_kind"
28       type="ProtectionKind" />
29     <xs:element name="topic_access_rules"
30       type="TopicAccessRules" />
31   </xs:sequence>
32 </xs:complexType>
33 <xs:complexType name="DomainIdSet">
34   <xs:choice minOccurs="1" maxOccurs="unbounded">
35     <xs:element name="id" type="DomainId" />
36     <xs:element name="id_range" type="DomainIdRange" />
37   </xs:choice>
38 </xs:complexType>
39 <xs:simpleType name="DomainId">
40   <xs:restriction base="xs:nonNegativeInteger" />
41 </xs:simpleType>
42 <xs:complexType name="DomainIdRange">
43   <xs:choice>
44     <xs:sequence>
45       <xs:element name="min" type="DomainId" />
46       <xs:element name="max" type="DomainId" minOccurs="0" />
47     </xs:sequence>
48     <xs:element name="max" type="DomainId" />
49   </xs:choice>
50 </xs:complexType>
51 <xs:simpleType name="ProtectionKind">
52   <xs:restriction base="xs:string">
53     <xs:enumeration value="ENCRYPT_WITH_ORIGIN_AUTHENTICATION" />
54     <xs:enumeration value="SIGN_WITH_ORIGIN_AUTHENTICATION" />
55     <xs:enumeration value="ENCRYPT" />
56     <xs:enumeration value="SIGN" />
57     <xs:enumeration value="NONE" />
58   </xs:restriction>
59 </xs:simpleType>
60 <xs:simpleType name="BasicProtectionKind">
61   <xs:restriction base="ProtectionKind">
62     <xs:enumeration value="ENCRYPT" />

```

(continues on next page)

(continued from previous page)

```

63         <xs:enumeration value="SIGN" />
64         <xs:enumeration value="NONE" />
65     </xs:restriction>
66 </xs:simpleType>
67 <xs:complexType name="TopicAccessRules">
68     <xs:sequence minOccurs="1" maxOccurs="unbounded">
69         <xs:element name="topic_rule" type="TopicRule" />
70     </xs:sequence>
71 </xs:complexType>
72 <xs:complexType name="TopicRule">
73     <xs:sequence minOccurs="1" maxOccurs="1">
74         <xs:element name="topic_expression" type="TopicExpression" />
75         <xs:element name="enable_discovery_protection"
76             type="xs:boolean" />
77         <xs:element name="enable_liveliness_protection"
78             type="xs:boolean" />
79         <xs:element name="enable_read_access_control"
80             type="xs:boolean" />
81         <xs:element name="enable_write_access_control"
82             type="xs:boolean" />
83         <xs:element name="metadata_protection_kind"
84             type="ProtectionKind" />
85         <xs:element name="data_protection_kind"
86             type="BasicProtectionKind" />
87     </xs:sequence>
88 </xs:complexType>
89 <xs:simpleType name="TopicExpression">
90     <xs:restriction base="xs:string" />
91 </xs:simpleType>
92 </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.

```

1 <dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:noNamespaceSchemaLocation="http://www.omg.org/spec/DDS-Security/20170801/omg_
   ↳ shared_ca_permissions.xsd">
3   <permissions>
4       <grant name="ParticipantPermissions">
5           <subject_name>emailAddress=example@eprosima.com, CN=DomainParticipantName,
   ↳ 0=eProsima, ST=MA, C=ES</subject_name>
6           <validity>
7               <not_before>2013-06-01T13:00:00</not_before>
8               <not_after>2038-06-01T13:00:00</not_after>
9           </validity>

```

(continues on next page)

(continued from previous page)

```

10      <allow_rule>
11        <domains>
12          <id_range>
13            <min>0</min>
14            <max>230</max>
15          </id_range>
16        </domains>
17        <publish>
18          <topics>
19            <topic>HelloWorldTopic</topic>
20          </topics>
21        </publish>
22      </allow_rule>
23      <default>DENY</default>
24    </grant>
25
26    <!-- Fill the subject name with the information specified in the Participant_
↪ certificate -->
27    <grant name="OtherParticipantPermissions">
28      <subject_name> emailAddress=xxx@eprosima.com, CN=xxx, OU=x, O=x, ST=XX, C=XX
↪ </subject_name>
29      <validity>
30        <not_before>2013-06-01T13:00:00</not_before>
31        <not_after>2038-06-01T13:00:00</not_after>
32      </validity>
33      <allow_rule>
34        <domains>
35          <id_range>
36            <min>0</min>
37            <max>230</max>
38          </id_range>
39        </domains>
40        <subscribe>
41          <topics>
42            <topic>HelloWorldTopic</topic>
43          </topics>
44        </subscribe>
45      </allow_rule>
46      <default>DENY</default>
47    </grant>
48  </permissions>
49</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	<code><publish></code>
Allow/Deny Subscribe	<code><subscribe></code>
Allow/Deny Relay	<code><relay></code>

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

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3   elementFormDefault="qualified" attributeFormDefault="unqualified">
4   <xs:element name="dds" type="PermissionsNode" />
5   <xs:complexType name="PermissionsNode">
6     <xs:sequence minOccurs="1" maxOccurs="1">
7       <xs:element name="permissions" type="Permissions" />
8     </xs:sequence>
9   </xs:complexType>
10  <xs:complexType name="Permissions">
11    <xs:sequence minOccurs="1" maxOccurs="unbounded">
12      <xs:element name="grant" type="Grant" />
13    </xs:sequence>
14  </xs:complexType>
15  <xs:complexType name="Grant">
16    <xs:sequence minOccurs="1" maxOccurs="1">
17      <xs:element name="subject_name" type="xs:string" />
```

(continues on next page)

(continued from previous page)

```

18     <xs:element name="validity" type="Validity" />
19     <xs:sequence minOccurs="1" maxOccurs="unbounded">
20         <xs:choice minOccurs="1" maxOccurs="1">
21             <xs:element name="allow_rule" minOccurs="0" type="Rule" />
22             <xs:element name="deny_rule" minOccurs="0" type="Rule" />
23         </xs:choice>
24     </xs:sequence>
25     <xs:element name="default" type="DefaultAction" />
26 </xs:sequence>
27 <xs:attribute name="name" type="xs:string" use="required" />
28 </xs:complexType>
29 <xs:complexType name="Validity">
30     <xs:sequence minOccurs="1" maxOccurs="1">
31         <xs:element name="not_before" type="xs:dateTime" />
32         <xs:element name="not_after" type="xs:dateTime" />
33     </xs:sequence>
34 </xs:complexType>
35 <xs:complexType name="Rule">
36     <xs:sequence minOccurs="1" maxOccurs="1">
37         <xs:element name="domains" type="DomainIdSet" />
38         <xs:sequence minOccurs="0" maxOccurs="unbounded">
39             <xs:element name="publish" type="Criteria" />
40         </xs:sequence>
41         <xs:sequence minOccurs="0" maxOccurs="unbounded">
42             <xs:element name="subscribe" type="Criteria" />
43         </xs:sequence>
44         <xs:sequence minOccurs="0" maxOccurs="unbounded">
45             <xs:element name="relay" type="Criteria" />
46         </xs:sequence>
47     </xs:sequence>
48 </xs:complexType>
49 <xs:complexType name="DomainIdSet">
50     <xs:choice minOccurs="1" maxOccurs="unbounded">
51         <xs:element name="id" type="DomainId" />
52         <xs:element name="id_range" type="DomainIdRange" />
53     </xs:choice>
54 </xs:complexType>
55 <xs:simpleType name="DomainId">
56     <xs:restriction base="xs:nonNegativeInteger" />
57 </xs:simpleType>
58 <xs:complexType name="DomainIdRange">
59     <xs:choice>
60         <xs:sequence>
61             <xs:element name="min" type="DomainId" />
62             <xs:element name="max" type="DomainId" minOccurs="0" />
63         </xs:sequence>
64         <xs:element name="max" type="DomainId" />
65     </xs:choice>
66 </xs:complexType>
67 <xs:complexType name="Criteria">
68     <xs:all minOccurs="1">
69         <xs:element name="topics" minOccurs="1"

```

(continues on next page)

(continued from previous page)

```

70         type="TopicExpressionList" />
71         <xs:element name="partitions" minOccurs="0"
72             type="PartitionExpressionList" />
73         <xs:element name="data_tags" minOccurs="0" type="DataTags" />
74     </xs:all>
75 </xs:complexType>
76 <xs:complexType name="TopicExpressionList">
77     <xs:sequence minOccurs="1" maxOccurs="unbounded">
78         <xs:element name="topic" type="TopicExpression" />
79     </xs:sequence>
80 </xs:complexType>
81 <xs:complexType name="PartitionExpressionList">
82     <xs:sequence minOccurs="1" maxOccurs="unbounded">
83         <xs:element name="partition" type="PartitionExpression" />
84     </xs:sequence>
85 </xs:complexType>
86 <xs:simpleType name="TopicExpression">
87     <xs:restriction base="xs:string" />
88 </xs:simpleType>
89 <xs:simpleType name="PartitionExpression">
90     <xs:restriction base="xs:string" />
91 </xs:simpleType>
92 <xs:complexType name="DataTags">
93     <xs:sequence minOccurs="1" maxOccurs="unbounded">
94         <xs:element name="tag" type="TagNameValuePair" />
95     </xs:sequence>
96 </xs:complexType>
97 <xs:complexType name="TagNameValuePair">
98     <xs:sequence minOccurs="1" maxOccurs="unbounded">
99         <xs:element name="name" type="xs:string" />
100        <xs:element name="value" type="xs:string" />
101    </xs:sequence>
102 </xs:complexType>
103 <xs:simpleType name="DefaultAction">
104     <xs:restriction base="xs:string">
105         <xs:enumeration value="ALLOW" />
106         <xs:enumeration value="DENY" />
107     </xs:restriction>
108 </xs:simpleType>
109 </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
```

7.23.3 Cryptographic plugin: DDS:Crypto:AES-GCM-GMAC

The cryptographic plugin provides the tools and operations required to support encryption and decryption, digests computation, message authentication codes computation and verification, key generation, and key exchange for DomainParticipants, *DataWriters* and *DataReaders*. Encryption can be applied over three different levels of DDS protocol:

- The whole RTPS messages.
- The RTPS submessages of a specific DDS Entity (DataWriter or DataReader).
- The payload (user data) of a particular DataWriter.

The authentication plugin implemented in Fast DDS is referred to as “DDS:Crypto:AES-GCM-GMAC”, in compliance with the [DDS Security](#) specification. This plugin is explained in detail below.

The DDS:Crypto:AES-GCM-GMAC plugin provides authentication encryption using Advanced Encryption Standard (AES) in Galois Counter Mode ([AES-GCM](#)). It supports 128 bits and 256 bits AES key sizes. It may also provide additional DataReader-specific Message Authentication Codes (MACs) using Galois MAC ([AES-GMAC](#)).

The DDS:Crypto:AES-GCM-GMAC authentication plugin, can be activated setting the *DomainParticipantQos* properties() `dds.sec.crypto.plugin` with the value `builtin.AES-GCM-GMAC`. Moreover, this plugin needs the activation of the *Authentication plugin: DDS:Auth:PKI-DH*. The DDS:Crypto:AES-GCM-GMAC plugin is configured using the *Access control plugin: DDS:Access:Permissions*, i.e the cryptography plugin is configured through the properties and configuration files of the access control plugin.

The following is an example of how to set the properties of *DomainParticipantQoS* for the DDS:Crypto:AES-GCM-GMAC configuration.

C++

```
DomainParticipantQos pqos;

// Activate DDS:Crypto:AES-GCM-GMAC plugin
pqos.properties().properties().emplace_back("dds.sec.crypto.plugin",
    "builtin.AES-GCM-GMAC");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_crypto_plugin_xml_profile">
  <rtps>
    <propertiesPolicy>
      <properties>
```

(continues on next page)

(continued from previous page)

```

        <!-- Activate DDS:Crypto:AES-GCM-GMAC plugin -->
        <property>
            <name>dds.sec.crypto.plugin</name>
            <value>builtin.AES-GCM-GMAC</value>
        </property>
    </properties>
</propertiesPolicy>
</rtps>
</participant>

```

7.23.4 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 not 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>
```

7.23.5 PKCS#11 support

The *private key* property used for the DDS:Auth:PKI-DH plugin configuration can be specified using a PKCS#11 compliant URI that represents a key stored in a HSM (Hardware Security Module). When a PKCS#11 URI is given, the *private key* is never taken out of the HSM, providing a more secure setup.

Support for PKCS#11 URIs is provided by the [libp11](#) library. This library provides a PKCS#11 engine for OpenSSL that acts as a proxy between OpenSSL and the HSM driver provided by the manufacturer. To make OpenSSL aware of the new engine, the OpenSSL configuration file might need to be updated. For details on how to set up the PKCS#11 engine in different platforms follow the dedicated documentation:

- *Libp11 and SoftHSM libraries* on Windows.
- *Libp11 and SoftHSM libraries* on Linux distributions

Also, the following section contains the list of *PropertyPolicyQos* that can be set within *Fast DDS Security: Security Property QoS Settings*.

7.24 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.

7.24.1 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*.

7.24.2 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.

7.24.3 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 awoken, 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()`. In addition, it is possible to configure certain settings of this logging thread via the member function `Log::SetThreadConfig()`.

```
// Block current thread until the log queue is empty.  
Log::Flush();  
  
// Stop the login thread and free its resources.  
Log::KillThread();  
  
// Configure ThreadSettings for the logging thread  
Log::SetThreadConfig(eprosima::fastdds::rtps::ThreadSettings{-1, 0, 0, -1});
```

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()`.

7.24.4 Logging Messages

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

- `EPROSIMA_LOG_INFO`: Logs messages with `Log::Kind::Info` verbosity.
- `EPROSIMA_LOG_WARNING`: Logs messages with `Log::Kind::Warning` verbosity.
- `EPROSIMA_LOG_ERROR`: Logs messages with `Log::Kind::Error` verbosity.

Said macros take exactly two arguments, a category and a message, and produce a log entry showing the message itself plus some meta information depending on the module's configuration (see [Log Entry Specification](#) and [Log Entry](#)).

```
EPROSIMA_LOG_INFO(DOCUMENTATION_CATEGORY, "This is an info message");
EPROSIMA_LOG_WARNING(DOCUMENTATION_CATEGORY, "This is an warning message");
EPROSIMA_LOG_ERROR(DOCUMENTATION_CATEGORY, "This is an error message");
```

There exist some old log macros used in previous versions: `logInfo`, `logWarning` and `logError`. These macros are still available as long as user does not manually disable them by `ENABLE_OLD_LOG_MACROS` CMake option or in-site macro `ENABLE_OLD_LOG_MACROS_` before including *Log* module. See section [Old Log macros disable](#) for more information.

Warning: Note that each message level is deactivated when CMake options `LOG_NO_INFO`, `LOG_NO_WARNING` or `LOG_NO_ERROR` are set to ON respectively. For more information about how to enable and disable each individual logging macro, please refer to [Disable Logging Module](#).

7.24.5 Module Configuration

The logging module offers a variety of configuration options. The different components of a log entry (see [Log Entry Specification](#)) can be configured as explained in [Log Entry](#). Furthermore, the logging module allows for registering several log consumer, allowing applications to direct the logging output to different destinations (see [Register Consumers](#)). In addition, some of the logging features can be configured using *eProsima Fast DDS* XML configuration files (see [XML Configuration](#)).

- [Log Entry](#)
- [Register Consumers](#)
- [Reset Configuration](#)
- [XML Configuration](#)

Log Entry

All the different components of a log entry are summarized in the following table (please refer to each component's section for further explanation):

Component	Optional	Default
<i>Timestamp</i>	NO	ENABLED
<i>Category</i>	NO	ENABLED
<i>Verbosity Level</i>	NO	ENABLED
<i>Message</i>	NO	ENABLED
<i>File Context</i>	YES	DISABLED
<i>Function Name</i>	YES	ENABLED

Timestamp

The log timestamp follows the [ISO 8601 standard](#) for local timestamps, i.e. *YYYY-MM-DD hh:mm:ss.sss*. This component cannot be further configured or disabled.

Category

Log entries have a category assigned when producing the log via the macros presented in [Logging Messages](#). The category component can be used to filter log entries so that only those categories specified in the filter are consumed (see [Filters](#)). This component cannot be further configured or disabled.

Verbosity Level

eProsima Fast DDS logging module provides three verbosity levels defined by the `Log::Kind` enumeration, those are:

- `Log::Kind::Error`: Used to log error messages.
- `Log::Kind::Warning`: Used to log error and warning messages.
- `Log::Kind::Info`: Used to log error, warning, and info messages.

The logging module's verbosity level defaults to `Log::Kind::Error`, which means that only messages logged with `EPROSIMA_LOG_ERROR` would be consumed. The verbosity level can be set and retrieved using member functions `Log::SetVerbosity()` and `Log::GetVerbosity()` respectively.

```
// Set log verbosity level to Log::Kind::Info
Log::SetVerbosity(Log::Kind::Info);

// Get log verbosity level
Log::Kind verbosity_level = Log::GetVerbosity();
```

Warning: Setting any of the CMake options `LOG_NO_INFO`, `LOG_NO_WARNING` or `LOG_NO_ERROR` to `ON` will completely disable the corresponding verbosity level. `LOG_NO_INFO` is set to `ON` for Single-Config generators as default value if not in Debug mode.

Message

This component constitutes the body of the log entry. It is specified when producing the log via the macros presented in [Logging Messages](#). The message component can be used to filter log entries so that only those entries whose message pattern-matches the filter are consumed (see [Filters](#)). This component cannot be further configured or disabled.

File Context

This component specifies the origin of the log entry in terms of file name and line number (see *Logging Messages* for a log entry example featuring this component). This is useful when tracing code flow for debugging purposes. The file context component can be enabled/disabled using the member function `Log::ReportFileNames()`.

```
// Enable file name and line number reporting
Log::ReportFileNames(true);

// Disable file name and line number reporting
Log::ReportFileNames(false);
```

Function Name

This component specifies the origin of the log entry in terms of the function name (see *Logging Messages* for a log entry example featuring this component). This is useful when tracing code flow for debugging purposes. The function name component can be enabled/disabled using the member function `Log::ReportFunctions()`.

```
// Enable function name reporting
Log::ReportFunctions(true);

// Disable function name reporting
Log::ReportFunctions(false);
```

Register Consumers

eProsima Fast DDS logging module supports zero or more *consumers* logging the entries registered in the logging queue with the methods described in *Logging Messages*. To register a consumer, the `Log` class exposes member function `Log::RegisterConsumer()`

```
// Create a FileConsumer consumer that logs entries in "archive.log"
std::unique_ptr<FileConsumer> file_consumer(new FileConsumer("archive.log"));
// Register the consumer. Log entries will be logged to STDOUT and "archive.log"
Log::RegisterConsumer(std::move(file_consumer));
```

The consumers list can be emptied with member function `Log::ClearConsumers()`.

```
// Clear all the consumers. Log entries are discarded upon consumption.
Log::ClearConsumers();
```

Note: Registering and configuring consumers can also be done using *Fast DDS XML* configuration files. Please refer to *XML Configuration* for details.

Warning: `Log::ClearConsumers()` empties the consumers lists. All log entries are discarded until a new consumer is register via `Log::RegisterConsumer()`, or until `Log::Reset()` is called.

Reset Configuration

The logging module's configuration can be reset to default settings with member function `Log::Reset()`.

Warning: Resetting the module's configuration entails:

- Setting *Verbosity Level* to `Log::Kind::Error`.
- Disabling *File Context* component.
- Enabling *Function Name* component.
- Clear all *Filters*.
- Clear all consumers and reset the default consumer according to CMake option `LOG_CONSUMER_DEFAULT`.

XML Configuration

eProsima Fast DDS allows for registering and configuring log consumers using XML configuration files. Please refer to *Log profiles* for details.

7.24.6 Filters

eProsima Fast DDS logging module allows for log entry filtering when consuming the logs, so that an application execution output can be limited to specific areas of interest. Beside the *Verbosity Level*, *Fast DDS* provides three different filtering possibilities.

- *Category Filtering*
- *File Name Filtering*
- *Content Filtering*
- *Reset Logging Filters*

It is worth mentioning that filters are applied in the specific order presented above, meaning that file name filtering is only applied to the entries that pattern-match the category filter, and content filtering is only applied to the entries that pattern-match both category and file name filters.

Category Filtering

Log entries can be filtered upon consumption according to their *Category* component using regular expressions. Each time an entry is ready to be consumed, the category filter is applied using `std::regex_search()`. To set a category filter, member function `Log::SetCategoryFilter()` is used:

```
// Set filter using regular expression
Log::SetCategoryFilter(std::regex("(CATEGORY_1)|(CATEGORY_2)"));

// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY_1, "First log entry");
// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY_2, "Second log entry");
```

(continues on next page)

(continued from previous page)

```
// Would NOT be consumed
EPROSIMA_LOG_ERROR(CATEGORY_3, "Third log entry");
```

The previous example would produce the following output:

```
2020-05-27 15:07:05.771 [CATEGORY_FILTER_1 Error] First log entry -> Function main
2020-05-27 15:07:05.771 [CATEGORY_FILTER_2 Error] Second log entry -> Function main
```

File Name Filtering

Log entries can be filtered upon consumption according to their *File Context* component using regular expressions. Each time an entry is ready to be consumed, the file name filter is applied using `std::regex_search()`. To set a file name filter, member function `Log::SetFilenameFilter()` is used:

```
// Filename: example.cpp

// Enable file name and line number reporting
Log::ReportFilenames(true);

// Set filter using regular expression so filename must match "example"
Log::SetFilenameFilter(std::regex("example"));
// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY, "First log entry");

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

The previous example would produce the following output:

```
2020-05-27 15:07:05.771 [CATEGORY Error] First log entry (example.cpp:50) -> Function_
↳main
```

Note: File name filters are applied even when the *File Context* entry component is disabled.

Content Filtering

Log entries can be filtered upon consumption according to their *Message* component using regular expressions. Each time an entry is ready to be consumed, the content filter is applied using `std::regex_search()`. To set a content filter, member function `Log::SetErrorStringFilter()` is used:

```
// Set filter using regular expression so message component must match "First"
Log::SetErrorStringFilter(std::regex("First"));
// Would be consumed
EPROSIMA_LOG_ERROR(CATEGORY, "First log entry");
// Would NOT be consumed
EPROSIMA_LOG_ERROR(CATEGORY, "Second log entry");
```

The previous example would produce the following output:

```
2020-05-27 15:07:05.771 [CATEGORY Error] First log entry -> Function main
```

Reset Logging Filters

The logging module's filters can be reset with member function `Log::Reset()`.

Warning: Resetting the module's filters entails:

- Setting *Verbosity Level* to `Log::Kind::Error`.
- Disabling *File Context* component.
- Enabling *Function Name* component.
- Clear all *Filters*.
- Clear all consumers and reset the default consumer according to CMake option `LOG_CONSUMER_DEFAULT`.

7.24.7 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));
```

7.24.8 Disable Logging Module

Setting the *Verbosity Level*, translates into entries not being added to the log queue if the entry’s level has lower importance than the set one. This check is performed when calling the macros defined in *Logging Messages*. However, it is possible to fully disable each macro (and therefore each verbosity level individually) at build time.

- EPROSIMA_LOG_INFO is fully disabled by either:
 - Setting CMake option LOG_NO_INFO to ON (default for Single-Config generators if CMAKE_BUILD_TYPE is other than Debug).
 - Defining macro HAVE_LOG_NO_INFO to 1.
- EPROSIMA_LOG_WARNING is fully disabled by either:
 - Setting CMake option LOG_NO_WARNING to ON.
 - Defining macro HAVE_LOG_NO_WARNING to 1.
- EPROSIMA_LOG_ERROR is fully disabled by either:
 - Setting CMake option LOG_NO_ERROR to ON.
 - Defining macro HAVE_LOG_NO_ERROR to 1.

Applying either of the previously described methods will set the macro to be empty at configuration time, thus allowing the compiler to optimize the call out. This is done so that all the debugging messages present on the library are optimized out at build time if not building for debugging purposes, thus preventing them to impact performance.

INTERNAL_DEBUG CMake option activates log macros compilation, so the arguments of the macros are compiled. However:

- It does not activate the log Warning and Error messages, i.e. the messages are not written in the log queue.
- EPROSIMA_LOG_INFO has a special behaviour to simplify working with Multi-Config capability IDEs. If CMake option LOG_NO_INFO is OFF, or the C++ definition HAVE_LOG_NO_INFO is 0, then logging is enabled only for Debug configuration. In this scenario, setting FASTDDS_ENFORCE_LOG_INFO to ON will enable EPROSIMA_LOG_INFO even on non Debug configurations. This is specially useful when using the *Fast DDS*' logging module in an external application which links with *Fast DDS* compiled in Release. In that case, applications wanting to use all three levels of logging can simply add the following code prior to including any Fast DDS header:

```
#define HAVE_LOG_NO_INFO 0
#define FASTDDS_ENFORCE_LOG_INFO 1
```

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

7.24.9 Old Log macros disable

Before version 2.8.2, Fast DDS project used log macros: `logInfo`, `logWarning` and `logError`, which may collide with other libraries. These log macros have been replaced by new ones with a more specific format: (e.g. `EPROSIMA_LOG_INFO`). In order to disable old macros compilation, use CMake option `ENABLE_OLD_LOG_MACROS = ON` or define `ENABLE_OLD_LOG_MACROS_ 0` before including the log module `#include <fastdds/dds/log/Log.hpp>`.

Warning: These macros will be deprecated in future versions of Fast DDS. The use of the new format ones is encouraged.

7.25 XML profiles

eProxima 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_FASTDDS_PROFILES.xml* located in the current execution path.
- Load an XML file which location is defined using the environment variable `FASTDDS_DEFAULT_PROFILES_FILE` (see [FASTDDS_DEFAULT_PROFILES_FILE](#)).
- Load the configuration parameters directly from the classes' definitions without looking for the *DEFAULT_FASTDDS_PROFILES.xml* in the working directory (see [SKIP_DEFAULT_XML](#)).
- Load directly the XML as a string data buffer.

An XML profile is defined by a unique name that is used to reference the XML profile during the creation of an *Entity*, the *Transport* configuration, or the *DynamicTypes* definition.

Both options can be complemented, i.e. it is possible to load multiple XML files but these must not have XML profiles with the same name. This section explains how to configure DDS entities using XML profiles. This includes the description of all the configuration values available for each of the XML profiles, as well as how to create complete XML files.

7.25.1 Creating an XML profiles file

An XML file can contain several XML profiles. These XML profiles are defined within the `<dds>` element, and in turn, within the `<profiles>` XML elements. The possible topologies for the definition of XML profiles are specified in *Routed vs Standalone profiles definition*.

It is worth mentioning that the first element of the xml profile must have the `xmlns` attribute with the link `xmlns="http://www.eprosima.com"`, in both rooted or standalone definitions. That link defines the reference of the `xsd` schema that the `xml` document complies with.

The available profile types are:

- *DomainParticipant profiles,*
- *DataWriter profiles,*
- *DataReader profiles,*
- *Topic profiles,*
- *Transport descriptors,*
- *Intra-process delivery profiles,*
- *Log profiles,* and
- *Dynamic Types profiles.*

The following sections will show implementation examples for each of these profiles.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <domainparticipant_factory profile_name="domainparticipant_factory_profile">
      <!-- ... -->
    </domainparticipant_factory>

    <participant profile_name="participant_profile">
      <!-- ... -->
    </participant>

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

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

    <topic profile_name="topic_profile">
      <!-- ... -->
    </topic>
```

(continues on next page)

(continued from previous page)

```

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

<library_settings>
    <!-- ... -->
</library_settings>

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

<types>
    <!-- ... -->
</types>
</dds>

```

Note: The *Example* section shows an XML file with all the possible configurations and profile types. This example is useful as a quick reference to look for a particular property and how to use it. The [Fast DDS XSD scheme](#) can be used as a quick reference too.

Loading and applying profiles

In case the user defines the Entity profiles via XML files, it is required to load these XML files using the `load_XML_profiles_file()` public member function before creating any entity. It is also possible to load directly the XML information as a string data buffer using the `load_XML_profiles_string()` public member function. Moreover, `create_participant_with_profile()`, `create_publisher_with_profile()`, `create_subscriber_with_profile()`, `create_datawriter_with_profile()`, and `create_datareader_with_profile()` member functions expect a profile name as an argument. *Fast DDS* searches the given profile name over all the loaded XML profiles, applying the profile to the entity if founded.

```

if (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");

    Topic* topic =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);

    Publisher* publisher = participant->create_publisher_with_profile("publisher_xml_
↪profile");
    DataWriter* datawriter = publisher->create_datawriter_with_profile(topic,
↪"datawriter_xml_profile");
    Subscriber* subscriber = participant->create_subscriber_with_profile("subscriber_xml_
↪profile");

```

(continues on next page)

(continued from previous page)

```

    DataReader* datareader = subscriber->create_datareader_with_profile(topic,
↪ "datareader_xml_profile");
}

// Load XML as string data buffer
std::string xml_profile =
    "\
    <?xml version=\"1.0\" encoding=\"UTF-8\" ?>\
    <dds>\
        <profiles xmlns=\"http://www.eprosima.com\" >\
            <data_writer profile_name=\"test_datawriter_profile\" is_default_
↪ profile=\"true\">\
                <qos>\
                    <durability>\
                        <kind>TRANSIENT_LOCAL</kind>\
                    </durability>\
                </qos>\
            </data_writer>\
        </profiles>\
    </dds>\
    ";
if (RETCODE_OK ==
    DomainParticipantFactory::get_instance()->load_XML_profiles_string(xml_profile.c_
↪ str(),
    xml_profile.length()))
{
    // Create DDS entities with profiles
}

```

For simplicity, the `create_participant_with_default_profile()` method takes the default profile set in the environment to create a participant. It requires the XML profile to have been already loaded. Please, refer to [XML profiles](#) for further information regarding loading profiles.

Warning: It is worth mentioning that if the same XML profile file is loaded multiple times, the second loading of the file will result in an error together with the consequent error log.

Note: To load dynamic types from XML files see the *Loading XML Types profile in Fast DDS application* subsection of *Dynamic Types profiles*.

Rooted vs Standalone profiles definition

Fast DDS offers various options for the definition of XML profiles. These options are:

- Stand-alone: The element defining the XML profile is the root element of the XML file. Elements `<dds>`, `<profiles>`, `<library_settings>`, `<types>`, and `<log>` can be defined in a stand-alone manner.
- Rooted: The element defining the XML profile is the child element of another element. For example, the `<participant>`, `<data_reader>`, `<data_writer>`, `<topic>`, and `<transport_descriptors>` elements must be defined as child elements of the `<profiles>` element.

The following is an example of the definition of the `<types>` XML profile using the two previously discussed approaches.

Stand-alone

```
<?xml version="1.0" encoding="UTF-8" ?>
<types xmlns="http://www.eprosima.com">
  <type>
    <!-- Type definition -->
  </type>

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

Rooted

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <types>
    <type>
      <!-- Type definition -->
    </type>

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

Note: Make sure that the first element of the xml profile must have the `xmlns` tag with the link `xmlns="http://www.eprosima.com"`, in both rooted or standalone definitions.

Modifying predefined XML profiles

Some scenarios may require to modify some of the QoS after loading the XML profiles. For such cases the *Types of Entities* which act as factories provide methods to get the QoS from the XML profile. This allows the user to read and modify predefined XML profiles before applying them to a new entity.

```
if (RETCODE_OK ==
    DomainParticipantFactory::get_instance()->load_XML_profiles_file("my_profiles.xml
↪"))
{
    DomainParticipantQos participant_qos;
    DomainParticipantFactory::get_instance()->get_participant_qos_from_profile(
        "participant_xml_profile",
        participant_qos);

    // Name obtained in another section of the code
    participant_qos.name() = custom_name;

    // Modify number of preallocations (this overrides the one set in the XML profile)
    participant_qos.allocation().send_buffers.preallocated_number = 10;

    // Create participant using the modified XML Qos
    DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(
            0, participant_qos);
}
```

Dynamic content by leveraging environment variables

For deployment scenarios that require part of the XML content to be dynamically generated, Fast DDS supports using environment variables on the text content of any XML tag. The format for environment variables expansion is `${ENV_VAR_NAME}`, where `ENV_VAR_NAME` follows the restrictions from [IEEE 1003.1](#):

Note: Environment variable names ... consist solely of uppercase letters, digits, and the ‘_’ (underscore) from the characters defined in Portable Character Set and do not begin with a digit.

More than one environment variable can be used, and they can be mixed with literal text.

The expansion will take place when the XML file is loaded, so changing the value of an environment variable afterwards will have no effect.

The following is an example of an XML allowing a participant to exclusively communicate with the participants on a fixed IP address, taken from `REMOTE_IP_ADDRESS` environment variable. It also gives the participant a name that mixes literal text with the content from two environment variables.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <participant profile_name="env_var_parsing_example">
      <rtps>
        <name>This is app '${MY_APP_NAME}' running on host '${MY_HOST_NAME}'</
↪name>
```

(continues on next page)

(continued from previous page)

```

        <builtin>
          <initialPeersList>
            <locator>
              <udp4>
                <address>${REMOTE_IP_ADDRESS}</address>
              </udp4>
            </locator>
          </initialPeersList>
        </builtin>
      </rtsp>
    </participant>
  </profiles>
</dds>

```

Warning: The Fast DDS XSD schema does not support the environment variables expansion feature, so validation of an XML file with environment variables expansion expressions will fail.

7.25.2 DomainParticipantFactory profiles

The *DomainParticipantFactory* profiles allow the definition of the configuration of DomainParticipantFactory through XML files. These profiles are defined within the `<domainparticipant_factory>` XML tags.

DomainParticipantFactory XML attributes

The `<domainparticipant_factory>` element has two attributes defined: `profile_name` and `is_default_profile`.

Name	Description	Use
<code>profile_n</code>	Sets the name under which the <code><domainparticipant_factory></code> profile is registered in the DDS Domain, so that it can be loaded later by the DomainParticipantFactory, as shown in <i>Loading and applying profiles</i> .	Mandatory
<code>is_default</code>	Sets the <code><domainparticipant_factory></code> profile as the default profile. Thus, if a default profile exists, it will be used when creating the DomainParticipantFactory	Optional

DomainParticipantFactory configuration

The `<domainparticipant_factory>` element has the following children elements:

Name	Description	Values
<code><qos></code>	DomainParticipantFactory QoS.	<i>QoS element type</i>

QoS element type

Name	Description	Values
<code><entity_factory></code>	Entity factory QoS Policy.	<i>Entity Factory</i>
<code><shm_watchdog_threa</code>	<i>ThreadSettings</i> for the SHM watchdog thread. See <i>Concurrency and mul-</i> <i>tithreading</i> .	<i>ThreadSet-</i> <i>tings</i>
<code><file_watch_threads</code>	<i>ThreadSettings</i> for the File watch threads. See <i>Concurrency and multi-</i> <i>threading</i> .	<i>ThreadSet-</i> <i>tings</i>

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com">
    <domainparticipant_factory profile_name="domainparticipant_factory_profile_name">
      <qos>
        <entity_factory>
          <autoenable_created_entities>true</autoenable_created_entities>
        </entity_factory>
        <shm_watchdog_thread>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </shm_watchdog_thread>
        <file_watch_threads>
          <scheduling_policy>-1</scheduling_policy>
          <priority>0</priority>
          <affinity>0</affinity>
          <stack_size>-1</stack_size>
        </file_watch_threads>
      </qos>
    </domainparticipant_factory>
  </profiles>
</dds>
```

7.25.3 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
<code>profile_name</code>	Sets the name under which the <code><participant></code> profile is registered in the DDS Domain, so that it can be loaded later by the <code>DomainParticipantFactory</code> , as shown in <i>Loading and applying profiles</i> .	Mandatory
<code>is_default_profile</code>	Sets the <code><participant></code> profile as the default profile. Thus, if a default profile exists, it will be used when no other <code>DomainParticipant</code> profile is specified at the <code>DomainParticipant</code> 's creation.	Optional

DomainParticipant configuration

The `<participant>` element has two child elements: `<domainId>` and `<rtps>`. All the `DomainParticipant` configuration options belong to the `<rtps>` element, except for the DDS `DomainId` which is defined by the `<domainId>` element. Below a list with the configuration XML elements is presented:

Name	Description	Values	Default
<code><domainId></code>	<code>DomainId</code> to be used by the <code>DomainParticipant</code> . See <i>Profile based creation of a DomainParticipant</i> .	<code>uint32_t</code>	0
<code><rtps></code>	<i>Fast DDS</i> <code>DomainParticipant</code> configurations. See <i>RTPS element type</i> .	<i>RTPS element type</i>	

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	De- fault
<name>	The DomainParticipant's name.	string_255	
<defaultUnicastLocators>	List of default reception unicast locators for user data traffic (see <metatrafficUnicastLocatorList> defined in <i>Builtin parameters</i>). It expects a <i>LocatorListType</i> .	<locator>	
<defaultMulticastLocators>	List of default reception multicast locators for user data traffic (see <metatrafficMulticastLocatorList> defined in <i>Builtin parameters</i>). It expects a <i>LocatorListType</i> .	<locator>	
<defaultExternalLocators>	List of <i>External Locators</i> to announce for the default user traffic of this participant.	<i>ExternalLocatorListType</i>	
<ignoreNonMatchingLocators>	Whether to ignore locators received on announcements from other participants when they don't match with any of the locators announced by this participant.	bool	false
<sendSocketBufferSize>	Size in bytes of the send socket buffer. If the value is zero then <i>Fast DDS</i> 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 <i>Fast DDS</i> will use the system default socket size.	uint32_t	0
<netmaskFiltering>	Participant's netmask filtering configuration. See the <i>Netmask filtering</i> section.	NetmaskFilteringType	AUTO
<builtinPublicData>	builtin public data member of the WireProtocolConfigQos class. See the <i>Builtin parameters</i> section.	<i>Builtin parameters</i>	
<port>	Allows defining the port and gains related to the RTPS protocol. See the <i>Port</i> section.	<i>Port</i>	
<participantId>	DomainParticipant's identifier. Typically it will be automatically generated by the <i>DomainParticipantFactory</i> .	int32_t	0
<userTransports>	Transport descriptors to be used by the DomainParticipant. See <i>Transport descriptors</i> .	List<string>	
<useBuiltinTransports>	Boolean field to indicate the system whether the DomainParticipant will use the default builtin transports in addition to its <userTransports>.	bool	true
<builtinTransportType>	Configuration option to determine which transports will be instantiated if the useBuiltinTransports is set to true. See <i>Managing the Builtin Transports</i> .	<i>BuiltinTransportType</i>	
<properties>	Additional configuration properties. See <i>PropertyPolicyQos</i> .	<i>PropertiesPolicyType</i>	
<allocation>	Configuration regarding allocation behavior. It expects a <i>DomainParticipantAllocationType</i> .	<i>DomainParticipantAllocationType</i>	
<userData>	Additional information attached to the DomainParticipant and transmitted with the discovery information. See <i>UserDataQosPolicy</i> .	List<string>	Empty
<prefix>	<i>DomainParticipant</i> 's GuidPrefix_t identifies peers running in the same process. Two participants with identical 8 first bytes on the GuidPrefix_t are considered to be running in the same process, and therefore intra-process delivery is used. See <i>Intra-process delivery</i> .	string	Empty
<builtinFlowControllerThreadSettings>	<i>ThreadSettings</i> for the builtin flow controllers sender thread.	<i>ThreadSettings</i>	
<builtinParticipantThreadSettings>	<i>ThreadSettings</i> participant's timed events thread.	<i>ThreadSettings</i>	
<builtinDiscoveryServerThreadSettings>	<i>ThreadSettings</i> for the discovery server thread.	<i>ThreadSettings</i>	
<builtinTypeLookupThreadSettings>	<i>ThreadSettings</i> for the threads used by the builtin TypeLookup service to discover unknown remote types. See <i>Remote Data Types Discovery</i> .	<i>ThreadSettings</i>	
<builtinTransportReceptionThreadSettings>	<i>ThreadSettings</i> for the builtin transports reception threads.	<i>ThreadSettings</i>	
<securityLogThreadSettings>	<i>ThreadSettings</i> for the security log thread.	<i>ThreadSettings</i>	

Example

```

<?xml version="1.0" encoding="UTF-8" ?>
<dds>
  <profiles xmlns="http://www.eprosima.com">
    <participant profile_name="domainparticipant_profile_name">
      <domainId>4</domainId>
      <rtps>
        <name>DomainParticipant Name</name>

        <defaultUnicastLocatorList>
          <!-- LOCATOR_LIST -->
          <locator>
            <udp4>
              <port>7400</port>
              <address>192.168.1.41</address>
            </udp4>
          </locator>
        </defaultUnicastLocatorList>

        <defaultMulticastLocatorList>
          <!-- LOCATOR_LIST -->
          <locator>
            <udp4>
              <port>7400</port>
              <address>192.168.2.41</address>
            </udp4>
          </locator>
        </defaultMulticastLocatorList>

        <default_external_unicast_locators>
          <!-- EXTERNAL_LOCATOR_LIST -->
          <udp4 externality="1" cost="0" mask="24">
            <address>100.100.100.10</address>
            <port>23456</port>
          </udp4>
        </default_external_unicast_locators>

        <ignore_non_matching_locators>true</ignore_non_matching_locators>

        <sendSocketBufferSize>8192</sendSocketBufferSize>

        <listenSocketBufferSize>8192</listenSocketBufferSize>

        <netmask_filter>ON</netmask_filter>

        <builtin>
          <!-- BUILTIN -->
        </builtin>

        <port>
          <portBase>7400</portBase>
          <domainIDGain>200</domainIDGain>

```

(continues on next page)

(continued from previous page)

```

    <participantIDGain>10</participantIDGain>
    <offsetd0>0</offsetd0>
    <offsetd1>1</offsetd1>
    <offsetd2>2</offsetd2>
    <offsetd3>3</offsetd3>
  </port>

  <participantID>99</participantID>

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

  <useBuiltinTransports>false</useBuiltinTransports>

  <builtinTransports>DEFAULT</builtinTransports>

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

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

  <userData>
    <value>3.4.7.0.C</value>
  </userData>

  <prefix>72.61.73.70.66.61.72.6d.74.65.73.74</prefix>

  <builtin_controllers_sender_thread>
    <scheduling_policy>-1</scheduling_policy>
    <priority>0</priority>
    <affinity>0</affinity>
    <stack_size>-1</stack_size>
  </builtin_controllers_sender_thread>

  <timed_events_thread>
    <scheduling_policy>-1</scheduling_policy>
    <priority>0</priority>
    <affinity>0</affinity>
    <stack_size>-1</stack_size>
  </timed_events_thread>

```

(continues on next page)

(continued from previous page)

```
<discovery_server_thread>
  <scheduling_policy>-1</scheduling_policy>
  <priority>0</priority>
  <affinity>0</affinity>
  <stack_size>-1</stack_size>
</discovery_server_thread>

<typelookup_service_thread>
  <scheduling_policy>-1</scheduling_policy>
  <priority>0</priority>
  <affinity>0</affinity>
  <stack_size>-1</stack_size>
</typelookup_service_thread>

<builtin_transports_reception_threads>
  <scheduling_policy>-1</scheduling_policy>
  <priority>0</priority>
  <affinity>0</affinity>
  <stack_size>-1</stack_size>
</builtin_transports_reception_threads>

<security_log_thread>
  <priority>0</priority>
  <affinity>0</affinity>
```

Note:

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

Builtin parameters

By calling the `wire_protocol()` member function of the `DomainParticipantQos`, it is possible to access the builtin public data member of the `WireProtocolConfigQos` class. This section specifies the available XML members for the configuration of this builtin parameters.

Name	Description	Values	Default
<code><discovery_config></code>	This is the main element within which discovery-related settings can be configured. See Discovery .	<i>discovery_config</i>	
<code><avoid_builtin_multicast></code>	Restricts multicast metatraffic to PDP only.	bool	true
<code><use_writer_liveliness></code>	Indicates whether to use the DataWriterLiveliness protocol.	bool	true
<code><metatraffic_unicast_locator_list></code>	Metatraffic Unicast Locator List.	A set of <code><locator></code> members. See LocatorListType	
<code><metatraffic_multicast_locator_list></code>	Metatraffic Multicast Locator List.	A set of <code><locator></code> members. See LocatorListType	
<code><initial_peers_list></code>	The list of IP-port address pairs of all other <i>DomainParticipants</i> with which a <i>DomainParticipant</i> will communicate. See Initial peers	A set of <code><locator></code> members. See LocatorListType	
<code><metatraffic_external_locator_list></code>	List of <i>External Locators</i> to announce for the metatraffic of this participant.	<i>ExternalLocatorListType</i>	
<code><reader_history_memory_policy_kind></code>	Memory policy for DataReaders. See HistoryQosPolicyKind .	<i>HistoryMemoryPolicy</i>	PREALLOCATED
<code><writer_history_memory_policy_kind></code>	Memory policy for DataWriters. See HistoryQosPolicyKind .	<i>HistoryMemoryPolicy</i>	PREALLOCATED
<code><reader_payload_size_limit></code>	Maximum DataReader's History payload size. Allows to reserve all the required memory at DataReader initialization. See MemoryManagementPolicy .	uint32_t	512
<code><writer_payload_size_limit></code>	Maximum DataWriter's History payload size. Allows to reserve all the required memory at DataWriter initialization. See MemoryManagementPolicy .	uint32_t	512
<code><muation_tries></code>	Number of different ports to try if DataReader's physical port is already in use.	uint32_t	100

Example

```
<builtin>
  <discovery_config>
    <discoveryProtocol>CLIENT</discoveryProtocol>

    <discoveryServersList>
      <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.74">
        <metatrafficUnicastLocatorList>
          <locator>
            <udp4>
              <address>192.168.10.57</address>
              <port>56542</port>
            </udp4>
          </locator>
        </metatrafficUnicastLocatorList>
      </RemoteServer>
    </discoveryServersList>
  </discovery_config>
</builtin>
```

(continues on next page)

(continued from previous page)

```

        <metatrafficMulticastLocatorList>
            <locator>
                <udp4>
                    <address>192.168.10.58</address>
                    <port>24565</port>
                </udp4>
            </locator>
        </metatrafficMulticastLocatorList>
    </RemoteServer>
</discoveryServersList>

<ignoreParticipantFlags>FILTER_DIFFERENT_HOST</ignoreParticipantFlags>

<EDP>SIMPLE</EDP>

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

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

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

<initialAnnouncements>
    <!-- INITIAL_ANNOUNCEMENTS -->
    <count>10</count>
    <period>
        <nanosec>50</nanosec>
    </period>
</initialAnnouncements>

<clientAnnouncementPeriod>
    <nanosec>2500000000</nanosec>
</clientAnnouncementPeriod>

<static_edp_xml_config>file://filename1.xml</static_edp_xml_config>
<static_edp_xml_config>file://filename2.xml</static_edp_xml_config>
<static_edp_xml_config>file://filename3.xml</static_edp_xml_config>
</discovery_config>

<avoid_builtin_multicast>true</avoid_builtin_multicast>

<use_WriterLivelinessProtocol>>false</use_WriterLivelinessProtocol>

<metatrafficUnicastLocatorList>

```

(continues on next page)

(continued from previous page)

```

    <!-- LOCATOR_LIST -->
    <locator>
      <udp4>
        <address>192.168.0.1</address>
      </udp4>
    </locator>
  </metatrafficUnicastLocatorList>

  <metatrafficMulticastLocatorList>
    <!-- LOCATOR_LIST -->
    <locator>
      <udp4>
        <address>192.168.0.1</address>
      </udp4>
    </locator>
  </metatrafficMulticastLocatorList>

  <initialPeersList>
    <!-- LOCATOR_LIST -->
    <locator>
      <udp4>
        <address>192.168.0.1</address>
      </udp4>
    </locator>
  </initialPeersList>

  <metatraffic_external_unicast_locators>
    <!-- EXTERNAL_LOCATOR_LIST -->
    <udp4 externality="1" cost="0" mask="24">
      <address>100.100.100.10</address>
      <port>34567</port>
    </udp4>
  </metatraffic_external_unicast_locators>

  <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</readerHistoryMemoryPolicy>

  <writerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</writerHistoryMemoryPolicy>

  <readerPayloadSize>512</readerPayloadSize>

  <writerPayloadSize>512</writerPayloadSize>

  <mutation_tries>55</mutation_tries>
</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	De- fault
<code><discoveryProtocol></code>	Indicates which discovery protocol the DomainParticipant will use. See Discovery mechanisms . If set to <code>CLIENT</code> , <code><discoveryServersList></code> element would be used.	SIMPLE CLIENT SERVER BACKUP NONE	SIMPLE
<code><discoveryServersList></code>	Describes servers from which it receives only the discovery information they require to establish communication with matching endpoints. See Discovery Server Settings	discoveryServersList	
<code><ignoreParticipantFlags></code>	Restricts metatraffic using several filtering criteria. See Ignore Participant flags .	ignoreParticipantFlags	NO_FILTER
<code><EDP></code>	If set to <code>SIMPLE</code> , <code><simpleEDP></code> element would be used. If set to <code>STATIC</code> , <code>EDPStatic</code> will be performed, configured with the contents of the XML file set in <code><static_edp_xml_config></code> . See Discovery .	SIMPLE STATIC	SIMPLE
<code><simpleEDP></code>	Attributes of the Simple Discovery Protocol. See Simple EDP Attributes .	simpleEDP	
<code><leaseDuration></code>	Indicates how long the DomainParticipant should consider remote DomainParticipants alive. See Lease Duration .	DurationType	20s
<code><leaseAnnouncementPeriod></code>	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
<code><initialAnnouncements></code>	Allows the user to configure the number and period of the DomainParticipant's initial discovery messages. See Initial Announcements .	Initial Announcements	
<code><clientAnnouncementPeriod></code>	The period for the DomainParticipant to send its Discovery Message to its servers and check for EDP endpoints matching.	DurationType	450 ms
<code><static_edp_xml_config></code>	The XML filename(s) with the static EDP configuration. Only necessary if the <code><EDP></code> member is set to <code>STATIC</code> . See STATIC Discovery Settings .	List <string>	

discoveryServersList

Contains a list of `<RemoteServer>` (*server*) elements, which are defined by the `GuidPrefix_t` and their own locator list which must be populated with `RemoteServerAttributes`. The DomainParticipant set as *client* would discover the *servers* described in this section.

Server attributes

The `<RemoteServer>` element has a mandatory attribute defined: `prefix`.

Name	Description	Use
prefi	RTPS standard participant unique identifier, a 12-byte chain. This identifier allows clients to assess whether they are receiving messages from the right server.	Mandatory

RemoteServer configuration

Each *client* (or a *server* connecting to another *server*) must keep a list of locators associated to the *servers* to which it wants to link. Those locator would be defined as `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList`.

Name	Description	Values
<metatrafficUnicastLocatorList>	Metatraffic Unicast Locator List.	A set of <locator> members. See LocatorListType
<metatrafficMulticastLocatorList>	Metatraffic Multicast Locator List.	A set of <locator> members. See LocatorListType

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <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>
                  <udp4>
                    <address>192.168.10.57</address>
                    <port>56542</port>
                  </udp4>
                </locator>
              </metatrafficUnicastLocatorList>
              <metatrafficMulticastLocatorList>
                <locator>
                  <udp4>
                    <address>192.168.10.58</address>
                    <port>24565</port>
                  </udp4>
                </locator>
              </metatrafficMulticastLocatorList>
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
      </builtin>
    </rtps>
  </participant>
</profiles>
```

ignoreParticipantFlags

Possible values	Description
NO_FILTER	All Discovery traffic is processed.
FILTER_DIFFERENT_HOST	Discovery traffic from another host is discarded.
FILTER_DIFFERENT_PROCESS	Discovery traffic from another process on the same host is discarded.
FILTER_SAME_PROCESS	Discovery traffic from DomainParticipant's own process is discarded.

This option also supports the OR (|) operator to filter discovery traffic from other configurations. For instance, `FILTER_DIFFERENT_PROCESS|FILTER_SAME_PROCESS` value discards discovery traffic from the DomainParticipant's own host.

simpleEDP

Name	Description	Values	Default
<PUBWRITER_SUBREAD	Indicates if the participant must use Publication DataWriter and Subscription DataReader.	bool	true
<PUBREADER_SUBWRIT	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	5
<period>	The period for the DomainParticipant to send its discovery messages.	<i>DurationType</i>	100 ms

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_lo	Defines the limits for the remote locators' collections. See RemoteLocatorsAllocationAttributes .	<i>Remote Locators Allocations</i>	
<total_par	DomainParticipant Allocation Configuration to specify the total number of DomainParticipants in the domain (local and remote). See ResourceLimitedContainerConfig .	<i>Allocation Configuration</i>	
<total_rea	DomainParticipant Allocation Configuration to specify the total number of DataReader on each DomainParticipant (local and remote). See ResourceLimitedContainerConfig .	<i>Allocation Configuration</i>	
<total_wri	DomainParticipant Allocation Configuration related to the total number of DataWriters on each DomainParticipant (local and remote). See ResourceLimitedContainerConfig .	<i>Allocation Configuration</i>	
<max_parti	Maximum size of the partitions submessage. Set to zero for no limit.	uint32_t	
<max_user_	Maximum size of the user data submessage. Set to zero for no limit.	uint32_t	
<max_prope	Maximum size of the properties submessage. Set to zero for no limit.	uint32_t	
<send_buff	Allocation behaviour for the send buffer manager.	<i>Send buffers</i>	

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>
```

(continues on next page)

(continued from previous page)

```

        <maximum>0</maximum>
        <increment>1</increment>
    </total_participants>

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

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

    <max_partitions>256</max_partitions>

    <max_user_data>256</max_user_data>

    <max_properties>512</max_properties>

    <send_buffers>
        <preallocated_number>127</preallocated_number>
        <dynamic>true</dynamic>
    </send_buffers>

    <!-- content_filter cannot be configured using XML (yet) -->
</allocation>

```

Remote Locators Allocations

Name	Description	Val- ues	De- fault
<max_unica	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 .	uint	4
<max_multi	Maximum number of multicast locators expected on a remote entity. May be set to zero to disable multicast traffic. See RemoteLocatorsAllocationAttributes .	uint	1

Send buffers

Name	Description	Values	De- fault
<preallocated_num	Initial number of send buffers to allocate. See <i>SendBuffersAllocation-Attributes</i> .	uint32_	0
<dynamic>	Whether the number of send buffers is allowed to grow. See <i>SendBuffer- sAllocationAttributes</i> .	bool	false

Note: The default value 0 of <preallocated_number> will perform an initial guess of the number of buffers required, based on the number of threads from which a send operation could be started. So it does not mean there are no buffers, instead it would use the maximum amount of buffers available.

7.25.4 DataWriter profiles

The DataWriter profiles allow for configuring *DataWriters* from an XML file. These profiles are defined within the <data_writer> XML tags.

DataWriter XML attributes

The <data_writer> element has two attributes defined: `profile_name` and `is_default_profile`.

Name	Description	Use
<code>profile_na</code>	Sets the name under which the <data_writer> profile is registered in the DDS Domain, so that it can be loaded later by the <i>DomainParticipant</i> , as shown in <i>Loading and applying profiles</i> .	Manda- tory
<code>is_default</code>	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.	Op- tional

DataWriter configuration

The DataWriter configuration is performed through the XML elements listed in the following table.

Name	Description	Values	De- fault
<topic>	<i>TopicType</i> configuration of the DataWriter.	<i>TopicType</i>	
<qos>	DataWriter <i>QoS</i> configuration.	<i>QoS</i>	
<times>	It configures some time related parameters of the DataWriter.	<i>WriterTimes</i>	
<unicastLocator	List of input unicast locators. It expects a <i>LocatorListType</i> .	<locator>	
<multicastLocat	List of input multicast locators. It expects a <i>LocatorListType</i> .	<locator>	
<external_unica	List of <i>External Locators</i> to announce for the communication with this DataWriter.	<i>ExternalLocatorListType</i>	
<ignore_non_mat	Whether to ignore locators received on announcements from other entities when they don't match with any of the locators announced by this DataWriter.	bool	false
<historyMemoryF	Memory allocation kind for DataWriter's history. See <i>HistoryQosPolicyKind</i> .	<i>HistoryMemoryPolicy</i>	PREALLOCATED
<propertiesPoli	Additional configuration properties.	<i>PropertiesPolicyType</i>	
<userDefinedID>	Used for EDPStatic.	int16_t	-1
<entityID>	Sets the <code>entity_id</code> of the <i>RTPSEndpointQos</i> class.	int16_t	-1
<matchedSubscri	Sets the limits of the collection of matched DataReaders. See <i>ParticipantResourceLimitsQos</i> .	<i>AllocationConfiguration</i>	

Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <data_writer profile_name="datawriter_profile_name">
      <topic>
        <historyQos>
          <kind>KEEP_LAST</kind>
        </historyQos>
      </topic>

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

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

        <heartbeatPeriod>
```

(continues on next page)

(continued from previous page)

```

        <sec>3</sec>
    </heartbeatPeriod>

    <nackResponseDelay>
        <nanosec>5</nanosec>
    </nackResponseDelay>

    <nackSupressionDuration>
        <sec>0</sec>
    </nackSupressionDuration>
</times>

<unicastLocatorList>
    <!-- LOCATOR_LIST -->
    <locator>
        <udp4>
            <address>192.168.0.1</address>
        </udp4>
    </locator>
</unicastLocatorList>

<multicastLocatorList>
    <!-- LOCATOR_LIST -->
    <locator>
        <udp4>
            <address>192.168.0.1</address>
        </udp4>
    </locator>
</multicastLocatorList>

<external_unicast_locators>
    <!-- EXTERNAL_LOCATOR_LIST -->
    <udp4 externality="1" cost="0" mask="24">
        <address>100.100.100.10</address>
        <port>12345</port>
    </udp4>
</external_unicast_locators>

<ignore_non_matching_locators>true</ignore_non_matching_locators>

<historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

<!-- PROPERTIES_POLICY -->
<propertiesPolicy>
    <properties>
        <property>
            <name>PropertyName</name>
            <value>PropertyValue</value>
        </property>
    </properties>
</propertiesPolicy>

```

(continues on next page)

(continued from previous page)

```

<userDefinedID>55</userDefinedID>

<entityID>66</entityID>

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

  <!-- reader_filters_allocation cannot be configured using XML (yet) -->
</data_writer>
</profiles>
</dds>

```

Note:

- LOCATOR_LIST means a *LocatorListType* is expected.
- EXTERNAL_LOCATOR_LIST means a *ExternalLocatorListType* is expected.
- PROPERTIES_POLICY means that the label is a *PropertiesPolicyType* block.
- For QoS details, please refer to *QoS*.
- TOPIC_TYPE is detailed in section *TopicType*.

WriterTimes

These parameters are included within *RTPSReliableWriterQos* in the *WriterTimes* structure.

Name	Description	Values	De- fault
<initialHeartbeatDelay>	Initial heartbeat delay.	<i>DurationType</i>	12 ms
<heartbeatPeriod>	Periodic heartbeat period.	<i>DurationType</i>	3 s
<nackResponseDelay>	Delay to apply to the response of an ACKNACK message.	<i>DurationType</i>	5 ms
<nackSuppressionDelay>	This time allows the DataWriter to ignore NACK messages for a given period of time right after the data has been sent.	<i>DurationType</i>	0 ms

7.25.5 DataReader profiles

The DataReader profiles allow declaring *DataReaders* from an XML file. These profiles are defined within the `<data_reader>` XML tags.

DataReader XML attributes

The `<data_reader>` element has two attributes defined: `profile_name` and `is_default_profile`.

Name	Description	Use
<code>profile_name</code>	Sets the name under which the <code><data_reader></code> profile is registered in the DDS Domain, so that it can be loaded later by the <i>DomainParticipant</i> , as shown in <i>Loading and applying profiles</i> .	Mandatory
<code>is_default</code>	Sets the <code><data_reader></code> 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	De- fault
<topic>	<i>TopicType</i> configuration of the DataReader.	<i>TopicType</i>	
<qos>	Subscriber <i>QoS</i> configuration.	<i>QoS</i>	
<times>	It allows configuring some time related parameters of the DataReader.	<i>ReaderTimes</i>	
<unicastLocator	List of input unicast locators. It expects a <i>LocatorListType</i> .	List of <i>LocatorListType</i>	
<multicastLocat	List of input multicast locators. It expects a <i>LocatorListType</i> .	List of <i>LocatorListType</i>	
<external_unica	List of <i>External Locators</i> to announce for the communication with this DataReader.	<i>ExternalLocatorListType</i>	
<ignore_non_mat	Whether to ignore locators received on announcements from other entities when they don't match with any of the locators announced by this DataReader.	bool	false
<expectsInlineC	It indicates if QoS is expected inline.	bool	false
<historyMemoryF	Memory allocation kind for DataReaders's history.	<i>HistoryMemoryPolicy</i>	PREALLOCATED
<propertiesPoli	Additional configuration properties.	<i>PropertiesPolicyType</i>	
<userDefinedID>	Used for StaticEndpointDiscovery.	int16_t	-1
<entityID>	Set the entity_id of the <i>RTPSEndpointQos</i> class.	int16_t	-1
<matchedPublish	Sets the limits of the collection of matched DataWriters. See <i>ParticipantResourceLimitsQos</i> .	<i>AllocationConfiguration</i>	

Example

```

<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <data_reader profile_name="data_reader_profile_name">
      <topic>
        <historyQos>
          <kind>KEEP_LAST</kind>
        </historyQos>
      </topic>

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

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

```

(continues on next page)

(continued from previous page)

```

        <heartbeatResponseDelay>
            <nanosec>5</nanosec>
        </heartbeatResponseDelay>
    </times>

    <unicastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udp4>
                <address>192.168.0.1</address>
            </udp4>
        </locator>
    </unicastLocatorList>

    <multicastLocatorList>
        <!-- LOCATOR_LIST -->
        <locator>
            <udp4>
                <address>192.168.0.1</address>
            </udp4>
        </locator>
    </multicastLocatorList>

    <external_unicast_locators>
        <!-- EXTERNAL_LOCATOR_LIST -->
        <udp4 externality="1" cost="0" mask="24">
            <address>100.100.100.10</address>
            <port>12345</port>
        </udp4>
    </external_unicast_locators>

    <ignore_non_matching_locators>true</ignore_non_matching_locators>

    <expectsInlineQos>true</expectsInlineQos>

    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

    <!-- PROPERTIES_POLICY -->
    <propertiesPolicy>
        <properties>
            <property>
                <name>PropertyName</name>
                <value>PropertyValue</value>
            </property>
        </properties>
    </propertiesPolicy>

    <userDefinedID>55</userDefinedID>

    <entityID>66</entityID>

```

(continues on next page)

(continued from previous page)

```

    <matchedPublishersAllocation>
      <initial>0</initial>
      <maximum>0</maximum>
      <increment>1</increment>
    </matchedPublishersAllocation>
  </data_reader>
</profiles>
</dds>

```

Note:

- LOCATOR_LIST means it expects a *LocatorListType*.
- EXTERNAL_LOCATOR_LIST means a *ExternalLocatorListType* is expected.
- PROPERTIES_POLICY means that the label is a *PropertiesPolicyType* block.
- For QoS details, please refer to *QoS*.
- TOPIC_TYPE is detailed in section *TopicType*.

ReaderTimes

These parameters are included within *RTPSReliableReaderQos* in the *ReaderTimes* structure.

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

7.25.6 Topic profiles

The topic profiles allow for configuring *Topic* from an XML file. These profiles are defined within the <topic> XML tags.

Topic XML attributes

The <topic> element has two attributes defined: `profile_name` and `is_default_profile`.

Name	Description	Use
<code>profile_name</code>	Sets the name under which the <topic> profile is registered in the DDS Domain, so that it can be loaded later by the <i>DataWriter</i> or the <i>DataReader</i>	Mandatory
<code>is_default_</code>	Sets the <topic> profile as the default profile. Thus, if a default profile exists, it will be used when no other Topic profile is specified at the Topic's creation.	Optional

Topic configuration

This XML element allows the configuration of the *TopicQos*.

Name	Description	Values
<code><historyQos></code>	It controls the behavior of <i>Fast DDS</i> when the value of an instance changes before it is finally communicated to some of its existing DataReaders.	<i>HistoryQoS</i>
<code><resourceLim</code>	It controls the resources that <i>Fast DDS</i> can use in order to meet the requirements imposed by the application and other QoS settings.	<i>ResourceLimitsQoS</i>

Example

```
<topic profile_name="topic_example">
  <historyQos>
    <kind>KEEP_LAST</kind>
    <depth>20</depth>
  </historyQos>
  <resourceLimitsQos>
    <max_samples>5</max_samples>
    <max_instances>2</max_instances>
    <max_samples_per_instance>1</max_samples_per_instance>
    <allocated_samples>20</allocated_samples>
    <extra_samples>10</extra_samples>
  </resourceLimitsQos>
</topic>
```

7.25.7 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 `<transport_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
<code><transport_id></code>	Unique name to identify each transport descriptor.
<code><type></code>	Type of the transport descriptor.
<code><sendBufferSize></code>	Size in bytes of the send socket buffer. If the value is zero then <i>Fast DDS</i> will use the system default.
<code><receiveBufferSize></code>	Size in bytes of the reception socket buffer. If the value is zero then <i>Fast DDS</i> will use the system default.

Table 13 – con

Name	Description
<maxMessageSize>	The maximum size in bytes of the transport's message buffer.
<maxInitialPeersRange>	Number of channels opened with each initial remote peer.
<netmask_filter>	Transport's <i>Netmask filter</i> -ing configuration.
<interfaces>	Allows defining an <i>Interfaces configuration</i> .
<interfaceWhiteList>	Allows defining an interfaces <i>Whitelist</i> .
<TTL>	<i>Time To Live (UDP only)</i> . See <i>UDP Transport</i> .
<non_blocking_send>	Whether to set the non-blocking send mode on the socket (NOT available for SHM type). See <i>U</i>
<output_port>	Port used for output bound. If this field isn't defined, the output port will be random (UDP only).
<wan_addr>	Public WAN address when using TCPv4 transports . This field is optional if the transport doesn
<keep_alive_frequency_ms>	Frequency in milliseconds for sending <i>RTCP</i> keep-alive requests (TCP only).
<keep_alive_timeout_ms>	Time in milliseconds since the last keep-alive request was sent to consider a connection as broken
<max_logical_port>	The maximum number of logical ports to try during <i>RTCP</i> negotiations (TCP only).
<logical_port_range>	The maximum number of logical ports per request to try during <i>RTCP</i> negotiations (TCP only).
<logical_port_increment>	Increment between logical ports to try during <i>RTCP</i> negotiation (TCP only).
<listening_ports>	Local port to work as TCP acceptor for input connections. If not set, the transport will work as T
<tls>	Allows to define TLS related parameters and options (TCP only).
<calculate_crc>	Calculates the Cyclic Redundancy Code (CRC) for error control (TCP only).
<check_crc>	Check the CRC for error control (TCP only).
<enable_tcp_nodelay>	Socket option for disabling the Nagle algorithm. (TCP only).
<tcp_negotiation_timeout>	Time to wait for logical port negotiation (in ms) (TCP only).
<segment_size>	Size (in bytes) of the shared-memory segment. (Optional, SHM only).
<port_queue_capacity>	Capacity (in number of messages) available to every Listener (Optional, SHM only).
<healthy_check_timeout_ms>	Maximum time-out (in milliseconds) used when checking whether a Listener is alive (Optional, S
<rtps_dump_file>	Complete path (including file) where RTPS messages will be stored for debugging purposes. An

The following XML code shows an example of transport protocol configuration using all configurable parameters. More examples of transports descriptors can be found in the *Transport Layer* section.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>my_udp4_transport</transport_id>
        <type>UDPV4</type>
        <sendBufferSize>8192</sendBufferSize>
        <receiveBufferSize>8192</receiveBufferSize>
        <maxMessageSize>16384</maxMessageSize>
        <maxInitialPeersRange>100</maxInitialPeersRange>
        <netmask_filter>AUTO</netmask_filter>
        <interfaces>
          <allowlist>
            <interface name="wlp59s0" netmask_filter="ON"/>
          </allowlist>
          <blocklist>
            <interface name="127.0.0.1"/>
            <interface name="docker0"/>
          </blocklist>
        </interfaces>
      </transport_descriptor>
    </transport_descriptors>
  </profiles>
</dds>
```

(continues on next page)

(continued from previous page)

```

    </interfaces>
    <interfaceWhiteList>
      <address>192.168.1.41</address>
      <interface>lo</interface>
    </interfaceWhiteList>
    <TTL>250</TTL>
    <non_blocking_send>>false</non_blocking_send>
    <output_port>5101</output_port>
    <wan_addr>80.80.55.44</wan_addr>
    <keep_alive_frequency_ms>5000</keep_alive_frequency_ms>
    <keep_alive_timeout_ms>25000</keep_alive_timeout_ms>
    <max_logical_port>9000</max_logical_port>
    <logical_port_range>100</logical_port_range>
    <logical_port_increment>2</logical_port_increment>
    <listening_ports>
      <port>5100</port>
      <port>5200</port>
    </listening_ports>
    <tls><!-- TLS Section --></tls>
    <calculate_crc>>false</calculate_crc>
    <check_crc>>false</check_crc>
    <enable_tcp_nodelay>>false</enable_tcp_nodelay>
    <segment_size>262144</segment_size>
    <port_queue_capacity>512</port_queue_capacity>
    <healthy_check_timeout_ms>1000</healthy_check_timeout_ms>
    <rtps_dump_file>rtps_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	De- fault
<password>	Password of the <private_key_file> or <rsa_private_key_file> if provided.	string	
<private_key_file>	Path to the private key certificate file.	string	
<rsa_private_key_file>	Path to the private key RSA certificate file.	string	
<cert_chain_file>	Path to the public certificate chain file.	string	
<tmp_dh_file>	Path to the Diffie-Hellman parameters file	string	
<verify_file>	Path to the Certification Authority (CA) file.	string	
<verify_mode>	Establishes the verification mode mask. Several verification options can be combined in the same <transport_descriptor>.	VERIFY_NONE VERIFY_PEER VERIFY_FAIL_IF_NO_PEER_CERT VERIFY_CLIENT_ONCE	
<options>	Establishes the SSL Context options mask. Several options can be combined in the same <transport_descriptor>.	DEFAULT_WORKAROUNDS NO_COMPRESSION NO_SSLV2 NO_SSLV3 NO_TLSV1 NO_TLSV1_1 NO_TLSV1_2 NO_TLSV1_3 SINGLE_DH_USE	
<verify_paths>	Paths where the system will look for verification files.	List <string>	
<verify_depth>	Maximum allowed depth to verify intermediate certificates.	uint32_t	
<default_verify_files>	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
<server_name>	server name or host name required in case Server Name Indication (SNI) is used.	string	

An example of TLS protocol parameter configuration is shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <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>
            <verify>VERIFY_PEER</verify>
          </verify_mode>
          <options>
```

(continues on next page)

(continued from previous page)

```

        <option>NO_TLSV1</option>
        <option>NO_TLSV1_1</option>
    </options>
    <verify_paths>
        <verify_path>Path1</verify_path>
        <verify_path>Path2</verify_path>
        <verify_path>Path3</verify_path>
    </verify_paths>
    <verify_depth>55</verify_depth>
    <default_verify_path>true</default_verify_path>
    <handshake_role>SERVER</handshake_role>
    <server_name>my_server.com</server_name>
</tls>
</transport_descriptor>
<!-->
</profiles>

```

7.25.8 Intra-process delivery profiles

This section defines the XML elements available for configuring the *Intra-process delivery* settings parameters in *Fast DDS*. These elements are defined within the XML tag `<library_settings>`.

Intra-process delivery configuration

The Intra-process delivery configuration is performed through the XML elements listed in the following table.

Name	Description	Values	De- fault
<code><intraprocess_</code>	Speed up communications between entities within the same process by avoiding any of the overhead involved in the transport layer.	OFF USER_DATA_ONL FULL	OFF

Example

```

<library_settings>
  <intraprocess_delivery>FULL</intraprocess_delivery> <!-- OFF | USER_DATA_ONLY | FULL -->
</library_settings>

```

7.25.9 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	De- fault
<code><use_default></code>	If set to FALSE, a call to <code>Log::ClearConsumers()</code> is performed. See <i>Register Consumers</i> .	bool	true
<code><consumer></code>	Defines the class and configuration of the consumer to be registered. Multiple consumers can be registered this way. See <i>Consumers</i> .	<i>Consumer- DataType</i>	
<code><thread_set</code>	<i>ThreadSettings</i> for the logging thread.	<i>ThreadSet- tings</i>	

The following constitutes an example of an XML configuration file that sets the Log to use one `StdoutConsumer`, one `StdoutErrConsumer`, and one `FileConsumer`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <log>
    <!--
      Clear consumers
    -->
    <use_default>false</use_default>

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

    <!--
      StdoutErrConsumer with threshold set to Log::Kind::Error
    -->
    <consumer>
      <class>StdoutErrConsumer</class>
      <property>
        <name>stderr_threshold</name>
        <value>Log::Kind::Error</value>
      </property>
    </consumer>

    <!--
      FileConsumer opening "execution.log" in append mode
    -->
    <consumer>
      <class>FileConsumer</class>
      <property>
        <name>filename</name>
```

(continues on next page)

(continued from previous page)

```

        <value>execution.log</value>
      </property>
    <property>
      <name>append</name>
      <value>true</value>
    </property>
  </consumer>

  <!--
  ThreadSettings for the log's consumer thread
  -->
  <thread_settings>
    <scheduling_policy>-1</scheduling_policy>
    <priority>0</priority>
    <affinity>0</affinity>
    <stack_size>-1</stack_size>
  </thread_settings>
</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.	<i>PropertyType</i>

PropertyType

Name	Description	Values	Default
<name>	Name of the property to be configured.	filename append stderr_threshold	
<value>	The value of the property. <ul style="list-style-type: none">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	<i>output.log</i>
	<ul style="list-style-type: none">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
	<ul style="list-style-type: none">If <name> is set to stderr_threshold, then this element defines the threshold used by the <i>Log consumers</i>. This property only applies if <class> is set to StdoutErrConsumer	Log::Kind	Log::Kind::Warning

7.25.10 Dynamic Types profiles

Fast DDS supports the implementation of *Dynamic Language Binding* by defining them through XML files. Thus the topic data types can be modified without the need to modify the source code of the DDS application.

XML Structure

The definition of data type profiles in the XML file is done with the `types` tag. Each `types` element can contain one or more *Type definitions*. Defining several types within a `types` element or a single type for each `types` element has the same result. Below, an example of a stand-alone types definition via XML is shown.

```
<types xmlns="http://www.eprosima.com">
  <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 *eProsima Fast DDS* are presented. For further information about the supported *Dynamic Language Binding*, please, refer to *Supported Types*. For each type listed below, an example of how to build the type's XML profile is provided.

- *Primitive types*
- *String Types*
- *Enumeration Types*
- *Bitmask Types*
- *Alias Types*
- *Sequence Types*
- *Array Types*
- *Map Types*
- *Structure Types*
- *Union Types*
- *Bitset Types*

Member types

Member types are defined as any type that can belong to a *Struct* or a *Union*, or be aliased by a *Alias*. These can be defined by the `<member>` XML tag. A member can be annotated as key (equivalent of the IDL's `@key`) by setting the key attribute to `"true"`.

```
<member name="my_long" type="int32" key="true"/>
```

Primitive types

Primitive types are built-in types and they should be declared as members of an aggregated type (*Structure Types* or *Union Types*). Primitive types are declared by attribute `type` and the possible values are listed in the table below. Please, refer to *Primitive Types* for more information on primitive types.

boolean	byte	char8	char16	int32
uint32	int8	uint8	int16	uint16
int64	uint64	float32	float64	float128

All of them are declared as follows:

XML

```
<struct name="PrimitivesStruct">
  <member name="my_bool" type="boolean"/>
  <member name="my_octet" type="byte"/>
  <member name="my_char" type="char8"/>
  <member name="my_wchar" type="char16"/>
  <member name="my_long" type="int32"/>
  <member name="my_ulong" type="uint32"/>
  <member name="my_int8" type="int8"/>
  <member name="my_uint8" type="uint8"/>
  <member name="my_short" type="int16"/>
  <member name="my_ushort" type="uint16"/>
  <member name="my_longlong" type="int64"/>
  <member name="my_ulonglong" type="uint64"/>
  <member name="my_float" type="float32"/>
  <member name="my_double" type="float64"/>
  <member name="my_longdouble" type="float128"/>
</struct>
```

IDL

```
struct PrimitivesStruct
{
    boolean my_bool;
    octet my_octet;
    char my_char;
    wchar my_wchar;
    long my_long;
    unsigned long my_ulong;
    int8 my_int8;
```

(continues on next page)

(continued from previous page)

```

uint8 my_uint8;
short my_short;
unsigned short my_ushort;
long long my_longlong;
unsigned long long my_ulonglong;
float my_float;
double my_double;
long double my_longdouble;
};

```

String Types

String types should be defined as members of an aggregated type (*Structure Types* or *Union Types*). String types are defined with attribute `type` set to `string` or `wstring`. An optional attribute `stringMaxLength` might be used to set a maximum length for the string collection. Please, refer to *String Types* for more information on string types.

XML

```

<struct name="StringsStruct">
  <member name="my_string" type="string"/>
  <member name="my_wstring" type="wstring"/>
  <member name="my_bounded_string" type="string" stringMaxLength="41925"/>
  <member name="my_bounded_wstring" type="wstring" stringMaxLength="20925"/>
</struct>

```

IDL

```

struct StringsStruct
{
    string my_string;
    wstring my_wstring;
    string<41925> my_bounded_string;
    wstring<20925> my_bounded_wstring;
};

```

Enumeration Types

Enumeration types are defined using the `<enum>` tag. Attribute `name` and at least one `<enumerator>` child element are mandatory. Enumeration literals are defined using the `<enumerator>` tag with mandatory attribute `name`. Optionally, unsigned integer attribute `value` might be added to set a specific value for the enumeration literal.

Note: `value` attribute is equivalent to `@value` builtin annotation which is not still supported in neither the plain (IDL) nor *Dynamic Language Binding*.

Please, refer to *Enumeration Types* for more information on enumeration types.

XML

```

<enum name="MyEnum">
  <enumerator name="A" value="0"/>

```

(continues on next page)

(continued from previous page)

```

    <enumerator name="B" value="1"/>
    <enumerator name="C"/>
</enum>

<struct name="EnumStruct">
  <member name="my_enum" type="nonBasic" nonBasicTypeName="MyEnum"/>
</struct>

```

IDL

```

enum MyEnum
{
    A,
    B,
    C
};

struct EnumStruct
{
    MyEnum my_enum;
};

```

Bitmask Types

Bitmask types are defined using the `<bitmask>` tag. Attribute `name` and at least on `<bit_value>` child element are mandatory. Optionally, `bit_bound` attribute might be set to specify the bitmask bound (by default 32 bits). Bitflag elements are defined using the `<bit_value>` tag with mandatory attribute `name`. Optionally, `position` attribute might be defined to set the bitflag position within the bitmask. Please, refer to *Bitmask Types* for more information on bitmask types.

XML

```

<bitmask name="MyBitMask" bit_bound="8">
  <bit_value name="flag0" position="0"/>
  <bit_value name="flag1"/>
  <bit_value name="flag2"/>
  <bit_value name="flag5" position="5"/>
</bitmask>

<struct name="BitmaskStruct">
  <member name="my_bitmask" type="nonBasic" nonBasicTypeName="MyBitMask"/>
</struct>

```

IDL

```

@bit_bound(8)
bitmask MyBitMask
{
    @position(0) flag0,
    flag1,
    flag2,
    @position(5) flag5
}

```

(continues on next page)

(continued from previous page)

```
};

struct BitmaskStruct
{
    MyBitMask my_bitmask;
};
```

Alias Types

Alias types are defined using the `<typedef>` tag. Attributes `name` and `type` are mandatory. Depending on the aliased type, some other mandatory and/or optional attributes might be necessary or available. Non-primitive types must define the `type` attribute as `nonBasic` and include the `nonBasicTypeName` attribute with the name of the aliased type. Please, refer to *Alias Types* for more information on alias types.

XML

```
<typedef name="MyAliasedEnum" type="nonBasic" nonBasicTypeName="MyEnum"/>
<!-- XSD does not allow to set bounds to aliased strings -->
<typedef name="MyAliasedBoundedString" type="string"/>
<typedef name="MyRecursiveAlias" type="nonBasic" nonBasicTypeName="MyAliasedEnum"/>

<struct name="AliasStruct">
  <member name="my_aliased_enum" type="nonBasic" nonBasicTypeName="MyAliasedEnum"/>
  <member name="my_aliased_bounded_string" type="nonBasic" nonBasicTypeName=
  ↪ "MyAliasedBoundedString"/>
  <member name="my_recursive_alias" type="nonBasic" nonBasicTypeName="MyRecursiveAlias
  ↪ "/>
</struct>
```

IDL

```
typedef MyEnum MyAliasedEnum;
typedef string<100> MyAliasedBoundedString;
typedef MyAliasedEnum MyRecursiveAlias;

struct AliasStruct
{
    MyAliasedEnum my_aliased_enum;
    MyAliasedBoundedString my_aliased_bounded_string;
    MyRecursiveAlias my_recursive_alias;
};
```

Sequence Types

Sequence types should be defined as members of an aggregated type (*Structure Types* or *Union Types*). Sequence types are defined with mandatory attributes `type` set to the collection's element type, and `sequenceMaxLength` used to set the maximum collection's length. Unbounded sequences should set `sequenceMaxLength` attribute to `-1`. Please, refer to *Sequence types* for more information on sequence types.

XML

```
<struct name="SequenceStruct">
  <member name="bitmask_sequence" type="nonBasic" nonBasicTypeName="MyBitMask"
  ↪sequenceMaxLength="-1"/>
  <member name="short_sequence" sequenceMaxLength="5" type="int16"/>
</struct>
```

IDL

```
struct SequenceStruct
{
    sequence<MyBitMask> bitmask_sequence;
    sequence<short, 5> short_sequence;
};
```

Array Types

Array types should be defined as members of an aggregated type (*Structure Types* or *Union Types*). Array types are defined with mandatory attributes `type` set to the collection's element type, and `arrayDimensions` used to set the collection's dimensions. The format of `arrayDimensions` attribute value is the size of each dimension separated by commas. Please, refer to *Array types* for more information on array types.

XML

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

IDL

```
struct ArrayStruct
{
    long long_array[2][3][4];
};
```

Map Types

Map types should be defined as members of an aggregated type (*Structure Types* or *Union Types*). Map types are defined with mandatory attributes `type` set to the map's value type, `key_type` set to the map's key type, and `mapMaxLength` used to set the maximum map's number of key-value pairs. Unbounded maps should set `mapMaxLength` attribute to `-1`. Please, refer to *Map Types* for more information on map types.

XML

```
<struct name="MapStruct">
  <member name="string_alias_unbounded_map" type="nonBasic" nonBasicTypeName=
    "MyAliasedBoundedString" key_type="string" mapMaxLength="-1"/>
  <member name="short_long_map" type="int32" key_type="int16" mapMaxLength="2"/>
</struct>
```

IDL

```
struct MapStruct
{
    map<string, MyAliasedBoundedString> string_alias_unbounded_map;
    map<short, long, 2> short_long_map;
};
```

Structure Types

Structure types are defined using the `<struct>` tag with mandatory attribute `name`. Structure inheritance may be configured setting optional attribute `baseType`. XML Structure Types require at least one member defined.

Note: IDL specification introduced in version 4.1 the possibility of void content structures. Empty structures are not supported in XML Types profiles yet.

Structure members are defined using the `<member>` tag with mandatory attributes `name` and `type`. Depending on the member type, some other mandatory and/or optional attributes might be necessary or available. Non-primitive types must define the `type` attribute as `nonBasic` and include the `nonBasicTypeName` attribute with the name of the member type.

Note: Currently, XML Types profiles does not support setting the member ID or marking a member as key.

Please, refer to *Structure Types* for more information on structure types.

XML

```
<struct name="InnerStruct">
  <!-- XML does not support setting Member ID -->
  <member name="first" type="int32"/>
</struct>

<!-- TODO(XTypes: Fix inheritance loading from XML profile) Fast DDS#4626 -->
<!-- <struct name="ParentStruct">
  <member name="first" type="float32"/>
  <member name="second" type="int64"/>
</struct>

<struct name="ComplexStruct" baseType="ParentStruct">
  <member name="complex_member" type="nonBasic" nonBasicTypeName="InnerStruct"/>
</struct> -->
```

IDL

```

struct InnerStruct
{
    @id(0x10) long first;
};

struct ParentStruct
{
    float first;
    long long second;
};

struct ComplexStruct : ParentStruct
{
    InnerStruct complex_member;
};

```

Union Types

Union types are defined using the `<union>` tag with mandatory attribute `name`. A mandatory discriminator child must be defined using `<discriminator>` tag. Discriminator element requires `<type>` as mandatory attribute.

Union types also require at least one case child defined using the `<case>` tag. Each case child requires at least one label child using the `<caseDiscriminator>` tag. `value` attribute is mandatory and defines the label value. Several labels might be defined using several `<caseDiscriminator>` elements. Each case child must have exclusively one union member defined.

Union members are defined using the `<member>` tag with mandatory attributes `name` and `type`. Depending on the member type, some other mandatory and/or optional attributes might be necessary or available. Non-primitive types must define the `type` attribute as `nonBasic` and include the `nonBasicTypeName` attribute with the name of the member type. At least one union member must be defined for the union type to be consistent.

Note: Currently, XML Types profiles does not support setting the member ID or marking a member as key.

Please, refer to *Union Types* for more information on the union types.

XML

```

<union name="InnerUnion">
  <discriminator type="int16"/>
  <case>
    <caseDiscriminator value="0"/>
    <member name="first" type="nonBasic" nonBasicTypeName="PrimitivesStruct"/>
  </case>
  <case>
    <caseDiscriminator value="1"/>
    <caseDiscriminator value="default"/>
    <member name="second" type="int64"/>
  </case>
</union>
<union name="ComplexUnion">
  <discriminator type="int32"/>
  <case>

```

(continues on next page)

(continued from previous page)

```

    <caseDiscriminator value="0"/>
    <caseDiscriminator value="1"/>
    <member name="third" type="int32"/>
  </case>
  <case>
    <caseDiscriminator value="default"/>
    <member name="fourth" type="nonBasic" nonBasicTypeName="InnerUnion"/>
  </case>
</union>

```

IDL

```

union InnerUnion switch (short)
{
    case 0:
        @id(0x10) PrimitivesStruct first;
    case 1:
    default:
        long long second;
};

union ComplexUnion switch (long)
{
    case 0:
    case 1:
        long third;
    default:
        InnerUnion fourth;
};

```

Bitset Types

Bitset types are defined using the `<bitset>` tag with mandatory attribute `name`. Bitset inheritance may be configured setting optional attribute `baseType`. At least one bitfield child must be defined using `bitfield` tag.

Bitfield elements require mandatory attribute `bit_bound` with the number of bits managed by the bitfield (maximum 64 bits). Optionally, attributes `name` and `type` might be defined. An anonymous bitfield (attribute `name` not set) is not accessible and serves as padding between named bitfields. The `type` attribute can ease bitfield management explicitly setting an integer type that handles the bitfield.

Please, refer to *Bitset Types* for more information about the bitset types.

XML

```

<bitset name="ParentBitset">
  <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>

```

(continues on next page)

(continued from previous page)

```

<!-- TODO(XTypes: Fix inheritance loading from XML profile) Fast DDS#4626 -->
<!--<bitset name="ChildBitSet" baseType="ParentBitSet">
  <bitfield name="e" bit_bound="1"/>
  <bitfield name="f" bit_bound="20" type="uint32"/>
</bitset>

<struct name="BitsetStruct">
  <member name="my_bitset" type="nonBasic" nonBasicTypeName="ChildBitSet"/>
</struct>-->

```

IDL

```

bitset ParentBitSet
{
    bitfield<3> a;
    bitfield<1> b;
    bitfield<4>;
    bitfield<10> c;
    bitfield<12, short> d;
};

bitset ChildBitSet : ParentBitSet
{
    bitfield<1> e;
    bitfield<20, unsigned long> f;
};

struct BitsetStruct
{
    ChildBitSet my_bitset;
};

```

Loading XML Types profile in *Fast DDS* application

Fast DDS application can use types defined in XML configuration files once those files have been loaded into the DomainParticipantFactory using `load_XML_profiles_file()`. Types might be retrieved using `DomainParticipantFactory::get_dynamic_type_builder_from_xml_by_name()`. After getting the `DynamicType`, objects of `DynamicPubSubType` class might be instantiated and used to write/read data.

```

// Create a DomainParticipant
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Load the XML File
if (RETCODE_OK ==

```

(continues on next page)

(continued from previous page)

```

        DomainParticipantFactory::get_instance()->load_XML_profiles_file("my_profiles.xml
↪"))
{
    // Retrieve instance of the desired type
    DynamicType::_ref_type my_struct_type;
    if (RETCODE_OK !=
        DomainParticipantFactory::get_instance()->get_dynamic_type_builder_from_xml_
↪by_name(
        "MyStruct", my_struct_type))
    {
        // Error
        return;
    }

    // Register MyStruct type
    TypeSupport my_struct_type_support(new 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;
}

```

7.25.11 Common

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

- *LocatorListType*
- *ExternalLocatorListType*
- *PropertiesPolicyType*
- *DurationType*
- *TopicType*
 - *HistoryQoS*
 - *ResourceLimitsQos*
- *ThreadSettings*
- *BuiltinTransports*
- *QoS*
 - *Data-Sharing*
 - *Deadline*
 - *DisableHeartbeatPiggyback*
 - *DisablePositiveAcks*
 - *Durability*

- *Entity Factory*
- *GroupData*
- *LatencyBudget*
- *Lifespan*
- *Liveliness*
- *Ownership*
- *Ownership Strength*
- *Partition*
- *PublishMode*
- *ReliabilityQosPolicy*
- *TopicData*
- *UserData*
- *HistoryMemoryPolicy*
- *Allocation Configuration*

LocatorListType

It represents a list of `Locator_t`. `LocatorListType` is used inside other configuration parameter labels that expect a list of locators, for example, in `<defaultUnicastLocatorList>`. Therefore, `LocatorListType` is defined as a set of `<locator>` elements. The `<locator>` element has a single child element that defines the transport protocol for which the locator is defined. These are: `<udp4>`, `<tcp4>`, `<udp6>`, and `<tcp6>`. 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	De- fault
<code><port></code>	RTPS port number of the locator. <i>Physical port</i> in UDP, <i>logical port</i> in TCP.	<code>uint16_t</code>	0
<code><physical_port></code>	TCP's <i>physical port</i> .	<code>uint16_t</code>	0
<code><address></code>	IP address of the locator.	<code>string</code> (IPv4/IPv6 format or DNS name)	Empty
<code><unique_lan_id></code>	The LAN ID uniquely identifies the LAN the locator belongs to (TCPv4 only).	<code>string</code> (16 bytes)	Empty
<code><wan_address></code>	WAN IPv4 address (TCPv4 only).	<code>string</code> (IPv4 format)	0.0.0.0

Example

The following example shows the implementation of one locator of each transport protocol in `<defaultUnicastLocatorList>`.

```
<defaultUnicastLocatorList>
  <locator>
    <udp4>
      <!-- Access as physical, typical UDP usage -->
      <port>7400</port>
      <address>192.168.1.41</address>
    </udp4>
    <udp4>
      <!-- Access as physical, typical UDP usage -->
      <port>7600</port>
      <address>localhost</address>
    </udp4>
  </locator>
  <locator>
    <tcp4>
      <!-- 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>
    </tcp4>
  </locator>
  <locator>
    <udp6>
      <port>8844</port>
      <address>::1</address>
    </udp6>
    <udp6>
      <port>8888</port>
      <address>localhost</address>
    </udp6>
  </locator>
  <locator>
    <tcp6>
      <!-- Both physical and logical (port), useful in TCP transports -->
      <physical_port>5100</physical_port>
      <port>7400</port>
      <address>fe80::55e3:290:165:5af8</address>
    </tcp6>
  </locator>
</defaultUnicastLocatorList>
```

ExternalLocatorListType

It represents a list of external locator entries. Each entry can be a `<udp4>` or a `<udp6>` tag. These tags can be configured with the following attributes:

Name	Description	Values	Default
<code>external</code>	Number of hops from the participant's host to the LAN represented by the external locator. Valid values: from 1 to 255.	<code>uint8_t</code>	1
<code>cost</code>	Communication cost relative to other locators on the same externality level. Valid values: from 0 to 255.	<code>uint8_t</code>	0
<code>mask</code>	Number of significant bits on the LAN represented by the external locator. Valid values: from 1 to 31 (UDPv4) or 127 (UDPv6)	<code>uint8_t</code>	24

They should contain the following tags:

Name	Description	Values
<code><port></code>	UDP port number of the locator. The UDP port number should be valid.	<code>uint16_t</code>
<code><address></code>	IP address of the locator.	string (IPv4/IPv6 format or DNS name)

Example

The following example shows the implementation of one locator of each transport protocol in `<default_external_unicast_locators>`.

```
<default_external_unicast_locators>
  <udp4 externality="1" cost="0" mask="24">
    <address>100.100.100.10</address>
    <port>23456</port>
  </udp4>
  <udp6 externality="2" cost="0" mask="48">
    <address>::1</address>
    <port>1234</port>
  </udp6>
</default_external_unicast_locators>
```

PropertiesPolicyType

`PropertiesPolicyType` defines the `<propertiesPolicy>` element. It allows the user to define a set of generic properties inside a `<properties>` element. It is useful at defining extended or custom configuration parameters.

Name	Description	Values	Default
<code><name></code>	Name to identify the property.	string	
<code><value></code>	Property's value.	string	
<code><propagate></code>	Indicates if it is going to be serialized along with the object it belongs to.	bool	false

Example

```

<propertiesPolicy>
  <properties>
    <property>
      <name>Property1Name</name>
      <value>Property1Value</value>
      <propagate>>false</propagate>
    </property>
    <property>
      <name>Property2Name</name>
      <value>Property2Value</value>
      <propagate>>true</propagate>
    </property>
  </properties>
</propertiesPolicy>

```

DurationType

DurationType expresses a period of time and it is commonly used inside other XML elements, such as in `<leaseAnnouncement>` or `<leaseDuration>`. A DurationType is defined by at least one mandatory element of two possible ones: `<sec>` plus `<nanosec>`. An infinite value can be specified by using the values `DURATION_INFINITY`, `DURATION_INFINITE_SEC` and `DURATION_INFINITE_NSEC`.

Name	Description	Values	Default
<code><sec></code>	Number of seconds.	<code>int32_t</code>	0
<code><nanosec></code>	Number of nanoseconds.	<code>uint32_t</code>	0

Example

```

<discovery_config>
  <leaseDuration>
    <sec>DURATION_INFINITY</sec>
  </leaseDuration>
  <leaseAnnouncement>
    <sec>1</sec>
    <nanosec>856000</nanosec>
  </leaseAnnouncement>
</discovery_config>

```

TopicType

This XML element allows the configuration of the specific *HistoryQoSPolicy* and *ResourceLimitsQoSPolicy* QoS of the Datawriters and DataReaders in which this element is defined inside of. Also, it sets the *TopicQoS* configuration with the policies detailed.

Name	Description	Values
<code><historyQos></code>	It controls the behavior of <i>Fast DDS</i> when the value of an instance changes before it is finally communicated to some of its existing DataReaders.	<i>HistoryQoS</i>
<code><resourceLim</code>	It controls the resources that <i>Fast DDS</i> can use in order to meet the requirements imposed by the application and other QoS settings.	<i>ResourceLimitsQoS</i>

Example

```
<data_writer profile_name="dataWriter_topic_example">
  <topic>
    <historyQos>
      <kind>KEEP_LAST</kind>
      <depth>20</depth>
    </historyQos>
    <resourceLimitsQos>
      <max_samples>5</max_samples>
      <max_instances>2</max_instances>
      <max_samples_per_instance>1</max_samples_per_instance>
      <allocated_samples>20</allocated_samples>
      <extra_samples>10</extra_samples>
    </resourceLimitsQos>
  </topic>
</data_writer>
```

HistoryQoS

It controls the behavior of *Fast DDS* when the value of an instance changes before it is finally communicated to some of its existing DataReaders. Please refer to [HistoryQosPolicyKind](#) for further information on HistoryQoS.

Name	Description	Values	Default
<kind>	<i>Fast DDS</i> will only attempt to keep the latest values of the instance and discard the older ones.	KEEP_LAST	KEEP_LAST
	<i>Fast DDS</i> 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	1

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>.	int32_t	5000
<max_instances>	It defines the maximum number of instances.	int32_t	10
<max_samples_per_instance>	It must verify that: HistoryQos <depth> <= <max_samples_per_instance>.	int32_t	400
<allocated_samples>	It controls the maximum number of samples to be stored.	int32_t	100
<extra_samples>	The number of extra samples to allocate on the pool.	int32_t	1

ThreadSettings

It controls some OS settings for the *Fast DDS* created threads. Please refer to [ThreadSettings](#) for further information on ResourceLimitsQos.

Name	Description	Values	Default
<code>scheduling_policy</code>	Configures the scheduling policy used for the thread.	<code>int32</code>	-1
<code>priority</code>	Configures the thread's priority.	<code>int32</code>	-2 ³¹
<code>affinity</code>	On some systems (Windows, Linux), this is a bit mask for setting the threads affinity to each core individually. On MacOS, this sets the affinity tag for the thread, and the OS tries to share the L2 cache between threads with the same affinity.	<code>uint32</code>	0
<code>stack_size</code>	Configures the thread's stack size in bytes.	<code>int32</code>	-1

BuiltinTransports

It controls the builtin transports that will be used during the initialization of the DomainParticipant. Please refer to [Managing the Builtin Transports](#) for further information on builtin transports.

This type must follow this configuration:

Name	Description	Values	Default
<code><builtinTransport></code>	Defines the builtin transport mode.	NONE DEFAULT DEFAULTv6 SHM UDPv4 UDPv6 LARGE_DATA	DEFAULT

The `<builtinTransport>` tag can be configured with the following attributes:

Name	Description	Values	Default
<code>max_msg_size</code>	Maximum message size that will be specified in the transport layer. Valid values: from 1 to (2 ³²)-1.	<code>uint32</code>	65500
<code>sockets_size</code>	Size of the send and receive socket buffers. Valid values: from 0 to (2 ³²)-1.	<code>uint32</code>	OS default
<code>non_blocking</code>	Whether to use non-blocking send calls or not. Valid values: <code>true</code> or <code>false</code> .	<code>bool</code>	<code>false</code>
<code>tcp_negotiation</code>	Timeout duration for logical port negotiation. Valid values: from 1 to (2 ³²)-1. Only valid for LARGE_DATA mode.	<code>uint32</code>	0

QoS

The Quality of Service (QoS) is used to specify the behavior of the Service, allowing the user to define how each *Entity* will behave. Please refer to the *Policy* section for more information on QoS.

Name	Description	Values
<data_sharing>	See <i>DataSharingQosPolicy</i>	<i>Data-Sharing</i>
<deadline>	See <i>DeadlineQosPolicy</i> .	<i>Deadline</i>
<disable_heartbeat_piggyback>	See <i>DisableHeartbeatPiggyback</i> .	<i>DisableHeartbeatPiggyback</i>
<disablePositiveAcks>	See <i>DisablePositiveACKsQosPolicy</i> .	<i>DisablePositiveAcks</i>
<durability>	See <i>DurabilityQosPolicy</i> .	<i>Durability</i>
<entity_factory>	See <i>EntityFactoryQosPolicy</i> .	<i>Entity Factory</i>
<groupData>	See <i>GroupDataQosPolicy</i> .	<i>GroupData</i>
<latencyBudget>	See <i>LatencyBudgetQosPolicy</i> .	<i>LatencyBudget</i>
<lifespan>	See <i>LifespanQosPolicy</i> .	<i>Lifespan</i>
<liveliness>	See <i>LivelinessQosPolicy</i> .	<i>Liveliness</i>
<ownership>	See <i>OwnershipQosPolicy</i> .	<i>Ownership</i>
<ownershipStrength>	See <i>OwnershipStrengthQosPolicy</i> .	<i>Ownership Strength</i>
<partition>	See <i>PartitionQosPolicy</i> .	<i>Partition</i>
<publishMode>	See <i>PublishModeQosPolicy</i> .	<i>PublishMode</i>
<reliability>	See <i>ReliabilityQosPolicy</i> .	<i>ReliabilityQosPolicy</i>
<topicData>	See <i>TopicDataQosPolicy</i> .	<i>TopicData</i>
<userData>	See <i>UserDataQosPolicy</i> .	<i>UserData</i>

Example

```
<data_writer profile_name="pub_topic_qos">
<qos> <!-- writerQosPoliciesType -->
  <data_sharing>
    <kind>AUTOMATIC</kind>
    <shared_dir>/home</shared_dir>
    <max_domains>10</max_domains>
    <domain_ids>
      <domainId>0</domainId>
      <domainId>11</domainId>
    </domain_ids>
  </data_sharing>

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

  <!-- DataWriter specific QoS -->
  <disable_heartbeat_piggyback>true</disable_heartbeat_piggyback>

  <disablePositiveAcks>
    <enabled>true</enabled>
    <duration>
      <sec>1</sec>
    </duration>
```

(continues on next page)

(continued from previous page)

```

</disablePositiveAcks>

<durability>
  <kind>VOLATILE</kind>
</durability>

<groupData>
  <value>1.a.2</value>
</groupData>

<!-- QoS policy pending implementation -->
<latencyBudget>
  <duration>
    <sec>1</sec>
  </duration>
</latencyBudget>

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

<liveliness>
  <kind>AUTOMATIC</kind>

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

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

<ownership>
  <kind>EXCLUSIVE</kind>
</ownership>

<!-- DataWriter specific QoS -->
<ownershipStrength>
  <value>50</value>
</ownershipStrength>

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

<!-- DataWriter specific QoS -->

```

(continues on next page)

(continued from previous page)

```

<publishMode>
  <kind>ASYNCHRONOUS</kind>
</publishMode>

<reliability>
  <kind>BEST_EFFORT</kind>
  <max_blocking_time>
    <sec>1</sec>
  </max_blocking_time>
</reliability>

<topicData>
  <value>2.b.1</value>
</topicData>

<userData>
  <value>3.c.0</value>
</userData>
</qos>

```

Data-Sharing

Name	Description	Values	Default
<kind>	See <i>DataSharingKind</i>	AUTOMATIC ON OFF	AUTOMATIC
<shared_dir>	Directory used for the memory-mapped files.	string	Empty
<max_domains>	Maximum number of Data-Sharing domain IDs in the local or remote endpoints.	uint32_t	0 (unlimited)
<domain_ids>	List of Data-Sharing domain IDs configured for the current endpoint.	<domainId>	Empty list

Name	Description	Values
<domainId>	Domain ID to be used by the endpoint for Data-Sharing.	uint32_t

Deadline

Name	Description	Values	Default
<period>	See <i>DeadlineQosPolicy</i> .	<i>DurationType</i>	c_TimeInfinite

DisableHeartbeatPiggyback

Name	Description	Values	Default
<disable_heartbeat_piggyback>	See <i>DisableHeartbeatPiggyback</i> .	bool	false

Important: This configuration is only available for *DataWriter QoS profile configuration*.

DisablePositiveAcks

Name	Description	Values	Default
<enabled>	See <i>DisablePositiveACKsQosPolicy</i> .	bool	false
<duration>	See <i>DisablePositiveACKsQosPolicy</i> .	<i>DurationType</i>	c_TimeInfinite

Durability

Name	Description	Values	Default
<kind>	See <i>DurabilityQosPolicyKind</i> .	VOLATILE TRANSIENT_LOCAL TRANSIENT PERSISTENT	DataReaders: VOLATILE DataWriters: TRANSIENT_LOCAL

Entity Factory

Name	Description	Values	Default
<autoenable_created_entities>	See <i>EntityFactoryQosPolicy</i> .	bool	true

GroupData

Name	Description	Values	Default
<value>	See <i>GroupDataQosPolicy</i> .	string (std::vector<octet>)	Empty

LatencyBudget

Name	Description	Values	Default
<duration>	See <i>LatencyBudgetQosPolicy</i> .	<i>DurationType</i>	0

Lifespan

Name	Description	Values	Default
<duration>	See <i>LifespanQosPolicy</i> .	<i>DurationType</i>	c_TimeInfinite

Liveliness

Name	Description	Values	Default
<kind>	See <i>LivelinessQosPolicyKind</i> .	AUTOMATIC MANUAL_BY_PARTICIPANT MANUAL_BY_TOPIC	AUTOMATIC
<lease_duration>	See <i>LivelinessQosPolicy</i> .	<i>DurationType</i>	c_TimeInfinite
<announcement_period>	See <i>LivelinessQosPolicy</i> .	<i>DurationType</i>	c_TimeInfinite

Ownership

Name	Description	Values	Default
<kind>	See <i>OwnershipQosPolicyKind</i> .	SHARED EXCLUSIVE	SHARED

Ownership Strength

Name	Description	Values	Default
<value>	See <i>OwnershipStrengthQosPolicy</i> .	uint32_t	0

Important: This configuration is only available for *DataWriter QoS profile configuration*.

Partition

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

PublishMode

Name	Description	Values	Default
<kind>	See <i>PublishModeQosPolicy</i> .	ASYNCHRONOUS SYNCHRONOUS	ASYNCHRONOUS

Important: This configuration is only available for *DataWriter QoS profile configuration*.

ReliabilityQosPolicy

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

TopicData

Name	Description	Values	Default
<value>	See <i>TopicDataQosPolicy</i> .	string (std::vector<octet>)	Empty

UserData

Name	Description	Values	Default
<value>	See <i>UserDataQosPolicy</i> .	string (std::vector<octet>)	Empty

HistoryMemoryPolicy

Indicates the way the memory is managed in terms of dealing with the CacheChanges of the *RTPSEndpointQos*.

Name	Description	Values	Default
<historyMemor	Four different options as described in <i>MemoryManagementPolicy</i> .	PREALLOCATED PREALLOCATED_WITH_REALLOC DYNAMIC DYNAMIC_REUSABLE	PREALLOCATED

Example

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

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

7.25.12 Example

In this section, there is a full XML example with all possible configuration.

Warning: This example can be used as a quick reference, but it may not be correct due to incompatibility or exclusive properties. **Do not take it as a working example.**

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

(continues on next page)

(continued from previous page)

```

5      <!-- TCP sample transport descriptor -->
6      <transport_descriptor>
7          <transport_id>ExampleTransportId1</transport_id>
8          <type>TCPv4</type>
9          <sendBufferSize>8192</sendBufferSize>
10         <receiveBufferSize>8192</receiveBufferSize>
11         <maxMessageSize>16384</maxMessageSize>
12         <maxInitialPeersRange>100</maxInitialPeersRange>
13         <netmask_filter>AUTO</netmask_filter>
14         <interfaces>
15             <allowlist>
16                 <interface name="wlp59s0" netmask_filter="ON"/>
17             </allowlist>
18             <blocklist>
19                 <interface name="127.0.0.1"/>
20                 <interface name="docker0"/>
21             </blocklist>
22         </interfaces>
23         <interfaceWhiteList>
24             <address>192.168.1.41</address>
25             <interface>lo</interface>
26         </interfaceWhiteList>
27         <wan_addr>80.80.55.44</wan_addr>
28         <keep_alive_frequency_ms>5000</keep_alive_frequency_ms>
29         <keep_alive_timeout_ms>25000</keep_alive_timeout_ms>
30         <max_logical_port>200</max_logical_port>
31         <logical_port_range>20</logical_port_range>
32         <logical_port_increment>2</logical_port_increment>
33         <listening_ports>
34             <port>5100</port>
35             <port>5200</port>
36         </listening_ports>
37         <tls>
38             <password>Password</password>
39             <private_key_file>Key_file.pem</private_key_file>
40             <rsa_private_key_file>RSA_file.pem</rsa_private_key_file>
41             <cert_chain_file>Chain.pem</cert_chain_file>
42             <tmp_dh_file>DH.pem</tmp_dh_file>
43             <verify_file>verify.pem</verify_file>
44             <verify_mode>
45                 <verify>VERIFY_PEER</verify>
46             </verify_mode>
47             <options>
48                 <option>NO_TLSV1</option>
49                 <option>NO_TLSV1_1</option>
50             </options>
51             <verify_paths>
52                 <verify_path>Path1</verify_path>
53                 <verify_path>Path2</verify_path>
54                 <verify_path>Path3</verify_path>
55             </verify_paths>
56             <verify_depth>55</verify_depth>

```

(continues on next page)

(continued from previous page)

```

57         <default_verify_path>true</default_verify_path>
58         <handshake_role>SERVER</handshake_role>
59         <server_name>my_server.com</server_name>
60     </tls>
61     <calculate_crc>false</calculate_crc>
62     <check_crc>false</check_crc>
63     <enable_tcp_nodelay>false</enable_tcp_nodelay>
64     <default_reception_threads>
65         <scheduling_policy>-1</scheduling_policy>
66         <priority>0</priority>
67         <affinity>0</affinity>
68         <stack_size>-1</stack_size>
69     </default_reception_threads>
70     <reception_threads>
71         <reception_thread port="12345">
72             <scheduling_policy>-1</scheduling_policy>
73             <priority>0</priority>
74             <affinity>0</affinity>
75             <stack_size>-1</stack_size>
76         </reception_thread>
77     </reception_threads>
78 </transport_descriptor>
79 <!-- UDP sample transport descriptor. Several options are common with TCP -->
80 <transport_descriptor>
81     <transport_id>ExampleTransportId2</transport_id>
82     <type>UDIPv6</type>
83     <TTL>250</TTL>
84     <non_blocking_send>false</non_blocking_send>
85     <output_port>5101</output_port>
86     <default_reception_threads>
87         <scheduling_policy>-1</scheduling_policy>
88         <priority>0</priority>
89         <affinity>0</affinity>
90         <stack_size>-1</stack_size>
91     </default_reception_threads>
92     <reception_threads>
93         <reception_thread port="12345">
94             <scheduling_policy>-1</scheduling_policy>
95             <priority>0</priority>
96             <affinity>0</affinity>
97             <stack_size>-1</stack_size>
98         </reception_thread>
99     </reception_threads>
100 </transport_descriptor>
101 <!-- SHM sample transport descriptor -->
102 <transport_descriptor>
103     <transport_id>SHM_SAMPLE_DESCRIPTOR</transport_id>
104     <type>SHM</type> <!-- REQUIRED -->
105     <maxMessageSize>524288</maxMessageSize> <!-- OPTIONAL uint32 valid of
↪all transports-->
106     <segment_size>1048576</segment_size> <!-- OPTIONAL uint32 SHM only-->
107     <port_queue_capacity>1024</port_queue_capacity> <!-- OPTIONAL uint32 SHM

```

(continues on next page)

(continued from previous page)

```

108   <!-- only -->
109   <uint32 SHM only -->
110   <rtps_dump_file>test_file.dump</rtps_dump_file> <!-- OPTIONAL string SHM -->
111   <!-- only -->
112   <default_reception_threads> <!-- OPTIONAL -->
113     <scheduling_policy>-1</scheduling_policy>
114     <priority>0</priority>
115     <affinity>0</affinity>
116     <stack_size>-1</stack_size>
117   </default_reception_threads>
118   <reception_threads> <!-- OPTIONAL -->
119     <reception_thread port="12345">
120       <scheduling_policy>-1</scheduling_policy>
121       <priority>0</priority>
122       <affinity>0</affinity>
123       <stack_size>-1</stack_size>
124     </reception_thread>
125   </reception_threads>
126   <dump_thread>
127     <scheduling_policy>-1</scheduling_policy>
128     <priority>0</priority>
129     <affinity>0</affinity>
130     <stack_size>-1</stack_size>
131   </dump_thread>
132 </transport_descriptor>
133 </transport_descriptors>
134
135 <domainparticipant_factory profile_name="domainparticipant_factory_profile_name">
136   <qos>
137     <entity_factory>
138       <autoenable_created_entities>true</autoenable_created_entities>
139     </entity_factory>
140     <shm_watchdog_thread>
141       <scheduling_policy>-1</scheduling_policy>
142       <priority>0</priority>
143       <affinity>0</affinity>
144       <stack_size>-1</stack_size>
145     </shm_watchdog_thread>
146     <file_watch_threads>
147       <scheduling_policy>-1</scheduling_policy>
148       <priority>0</priority>
149       <affinity>0</affinity>
150       <stack_size>-1</stack_size>
151     </file_watch_threads>
152   </qos>
153 </domainparticipant_factory>
154
155 <participant profile_name="participant_profile_example">
156   <domainId>4</domainId>
157   <rtps>
158     <name>Participant Name</name> <!-- String -->

```

(continues on next page)

(continued from previous page)

```

157     <defaultUnicastLocatorList>
158         <locator>
159             <udp4>
160                 <!-- Access as physical, like UDP -->
161                 <port>7400</port>
162                 <address>localhost</address>
163             </udp4>
164         </locator>
165         <locator>
166             <tcp4>
167                 <!-- Both physical and logical (port), like TCP -->
168                 <physical_port>5100</physical_port>
169                 <port>7400</port>
170                 <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>
171                 <wan_address>80.80.99.45</wan_address>
172                 <address>192.168.1.55</address>
173             </tcp4>
174         </locator>
175         <locator>
176             <udp6>
177                 <port>8844</port>
178                 <address>::1</address>
179             </udp6>
180         </locator>
181     </defaultUnicastLocatorList>
182
183     <defaultMulticastLocatorList>
184         <locator>
185             <udp4>
186                 <!-- Access as physical, like UDP -->
187                 <port>7400</port>
188                 <address>192.168.1.41</address>
189             </udp4>
190         </locator>
191         <locator>
192             <tcp4>
193                 <!-- Both physical and logical (port), like TCP -->
194                 <physical_port>5100</physical_port>
195                 <port>7400</port>
196                 <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>
197                 <wan_address>80.80.99.45</wan_address>
198                 <address>192.168.1.55</address>
199             </tcp4>
200         </locator>
201         <locator>
202             <udp6>
203                 <port>8844</port>
204                 <address>::1</address>
205             </udp6>
206         </locator>
207     </defaultMulticastLocatorList>
208

```

(continues on next page)

(continued from previous page)

```

209     <default_external_unicast_locators>
210         <!-- EXTERNAL_LOCATOR_LIST -->
211         <udp4 externality="1" cost="0" mask="24">
212             <address>100.100.100.10</address>
213             <port>23456</port>
214         </udp4>
215         <udp6 externality="1" cost="1" mask="48">
216             <address>::1</address>
217             <port>1234</port>
218         </udp6>
219     </default_external_unicast_locators>
220
221     <ignore_non_matching_locators>true</ignore_non_matching_locators>
222     <sendSocketBufferSize>8192</sendSocketBufferSize>
223     <listenSocketBufferSize>8192</listenSocketBufferSize>
224     <netmask_filter>AUTO</netmask_filter>
225
226     <builtin>
227         <discovery_config>
228             <discoveryProtocol>NONE</discoveryProtocol>
229             <discoveryServersList>
230                 <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.74">
231                     <metatrafficUnicastLocatorList>
232                         <locator>
233                             <udp4>
234                                 <address>192.168.10.57</address>
235                                 <port>56542</port>
236                             </udp4>
237                         </locator>
238                     </metatrafficUnicastLocatorList>
239                     <metatrafficMulticastLocatorList>
240                         <locator>
241                             <udp4>
242                                 <address>192.168.10.58</address>
243                                 <port>24565</port>
244                             </udp4>
245                         </locator>
246                     </metatrafficMulticastLocatorList>
247                 </RemoteServer>
248                 <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.75">
249                     <metatrafficUnicastLocatorList>
250                         <locator>
251                             <udp4>
252                                 <address>192.168.10.59</address>
253                                 <port>56543</port>
254                             </udp4>
255                         </locator>
256                     </metatrafficUnicastLocatorList>
257                     <metatrafficMulticastLocatorList>
258                         <locator>
259                             <udp4>
260

```

(continues on next page)

(continued from previous page)

```

261         <address>192.168.10.60</address>
262         <port>34565</port>
263     </udp4>
264 </locator>
265 </metatrafficMulticastLocatorList>
266 </RemoteServer>
267 </discoveryServersList>
268 <ignoreParticipantFlags>FILTER_DIFFERENT_PROCESS|FILTER_SAME_
269 ←PROCESS</ignoreParticipantFlags>
270 <EDP>SIMPLE</EDP>
271 <simpleEDP>
272     <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
273     <PUBREADER_SUBWRITER>true</PUBREADER_SUBWRITER>
274 </simpleEDP>
275 <leaseDuration>
276     <sec>DURATION_INFINITY</sec>
277 </leaseDuration>
278 <leaseAnnouncement>
279     <sec>1</sec>
280     <nanosec>856000</nanosec>
281 </leaseAnnouncement>
282 <initialAnnouncements>
283     <count>10</count>
284     <period>
285         <nanosec>50</nanosec>
286     </period>
287 </initialAnnouncements>
288 <clientAnnouncementPeriod>
289     <nanosec>2500000000</nanosec>
290 </clientAnnouncementPeriod>
291 <static_edp_xml_config>filename1.xml</static_edp_xml_config>
292 <static_edp_xml_config>filename2.xml</static_edp_xml_config>
293 <static_edp_xml_config>filename3.xml</static_edp_xml_config>
294 </discovery_config>
295
296 <avoid_builtin_multicast>true</avoid_builtin_multicast>
297 <use_WriterLivelinessProtocol>>false</use_WriterLivelinessProtocol>
298
299 <metatrafficUnicastLocatorList>
300     <locator>
301         <udp4>
302             <!-- Access as physical, like UDP -->
303             <port>7400</port>
304             <address>192.168.1.41</address>
305         </udp4>
306     </locator>
307     <locator>
308         <tcp4>
309             <!-- Both physical and logical (port), like TCP -->
310             <physical_port>5100</physical_port>
311             <port>7400</port>
312             <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>

```

(continues on next page)

(continued from previous page)

```

312         <wan_address>80.80.99.45</wan_address>
313         <address>192.168.1.55</address>
314     </tcpv4>
315 </locator>
316 <locator>
317     <udp6>
318         <port>8844</port>
319         <address>::1</address>
320     </udp6>
321 </locator>
322 </metatrafficUnicastLocatorList>
323
324 <metatrafficMulticastLocatorList>
325     <locator>
326         <udp4>
327             <!-- Access as physical, like UDP -->
328             <port>7400</port>
329             <address>192.168.1.41</address>
330         </udp4>
331     </locator>
332     <locator>
333         <tcpv4>
334             <!-- Both physical and logical (port), like TCP -->
335             <physical_port>5100</physical_port>
336             <port>7400</port>
337             <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>
338             <wan_address>80.80.99.45</wan_address>
339             <address>192.168.1.55</address>
340         </tcpv4>
341     </locator>
342     <locator>
343         <udp6>
344             <port>8844</port>
345             <address>::1</address>
346         </udp6>
347     </locator>
348 </metatrafficMulticastLocatorList>
349
350 <initialPeersList>
351     <locator>
352         <udp4>
353             <!-- Access as physical, like UDP -->
354             <port>7400</port>
355             <address>192.168.1.41</address>
356         </udp4>
357     </locator>
358     <locator>
359         <tcpv4>
360             <!-- Both physical and logical (port), like TCP -->
361             <physical_port>5100</physical_port>
362             <port>7400</port>
363             <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>

```

(continues on next page)

(continued from previous page)

```

364         <wan_address>80.80.99.45</wan_address>
365         <address>192.168.1.55</address>
366     </tcpv4>
367 </locator>
368 <locator>
369     <udp6>
370         <port>8844</port>
371         <address>::1</address>
372     </udp6>
373 </locator>
374 </initialPeersList>
375
376 <metatraffic_external_unicast_locators>
377     <udp4 externality="1" cost="0" mask="24">
378         <address>100.100.100.10</address>
379         <port>34567</port>
380     </udp4>
381 </metatraffic_external_unicast_locators>
382
383 <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</
↪ readerHistoryMemoryPolicy>
384 <writerHistoryMemoryPolicy>PREALLOCATED</writerHistoryMemoryPolicy>
385 <readerPayloadSize>512</readerPayloadSize>
386 <writerPayloadSize>512</writerPayloadSize>
387 <mutation_tries>55</mutation_tries>
388 </builtin>
389
390 <port>
391     <portBase>7400</portBase>
392     <domainIDGain>200</domainIDGain>
393     <participantIDGain>10</participantIDGain>
394     <offsetd0>0</offsetd0>
395     <offsetd1>1</offsetd1>
396     <offsetd2>2</offsetd2>
397     <offsetd3>3</offsetd3>
398 </port>
399
400 <participantID>99</participantID>
401
402 <userTransports>
403     <transport_id>ExampleTransportId1</transport_id>
404     <transport_id>ExampleTransportId2</transport_id>
405 </userTransports>
406
407 <useBuiltinTransports>>false</useBuiltinTransports>
408
409 <builtinTransports max_msg_size="50KB" sockets_size="50KB" non_blocking=
↪ "false">DEFAULT</builtinTransports>
410
411 <propertiesPolicy>
412     <properties>
413         <property>

```

(continues on next page)

(continued from previous page)

```

414         <name>Property1Name</name>
415         <value>Property1Value</value>
416         <propagate>>false</propagate>
417     </property>
418     <property>
419         <name>Property2Name</name>
420         <value>Property2Value</value>
421         <propagate>>false</propagate>
422     </property>
423 </properties>
424 </propertiesPolicy>
425
426 <allocation>
427     <remote_locators>
428         <max_unicast_locators>4</max_unicast_locators> <!-- uint32 -->
429         <max_multicast_locators>1</max_multicast_locators> <!-- uint32 --
430
431     </remote_locators>
432     <total_participants>
433         <initial>0</initial>
434         <maximum>0</maximum>
435         <increment>1</increment>
436     </total_participants>
437     <total_readers>
438         <initial>0</initial>
439         <maximum>0</maximum>
440         <increment>1</increment>
441     </total_readers>
442     <total_writers>
443         <initial>0</initial>
444         <maximum>0</maximum>
445         <increment>1</increment>
446     </total_writers>
447     <max_partitions>256</max_partitions>
448     <max_user_data>256</max_user_data>
449     <max_properties>512</max_properties>
450     <send_buffers>
451         <preallocated_number>127</preallocated_number>
452         <dynamic>>true</dynamic>
453     </send_buffers>
454 </allocation>
455
456 <builtin_controllers_sender_thread>
457     <scheduling_policy>-1</scheduling_policy>
458     <priority>0</priority>
459     <affinity>0</affinity>
460     <stack_size>-1</stack_size>
461 </builtin_controllers_sender_thread>
462
463 <timed_events_thread>
464     <scheduling_policy>-1</scheduling_policy>
465     <priority>0</priority>

```

(continues on next page)

(continued from previous page)

```

465     <affinity>0</affinity>
466     <stack_size>-1</stack_size>
467 </timed_events_thread>
468
469 <discovery_server_thread>
470     <scheduling_policy>-1</scheduling_policy>
471     <priority>0</priority>
472     <affinity>0</affinity>
473     <stack_size>-1</stack_size>
474 </discovery_server_thread>
475
476 <typelookup_service_thread>
477     <scheduling_policy>-1</scheduling_policy>
478     <priority>0</priority>
479     <affinity>0</affinity>
480     <stack_size>-1</stack_size>
481 </typelookup_service_thread>
482
483 <builtin_transports_reception_threads>
484     <scheduling_policy>-1</scheduling_policy>
485     <priority>0</priority>
486     <affinity>0</affinity>
487     <stack_size>-1</stack_size>
488 </builtin_transports_reception_threads>
489
490 <security_log_thread>
491     <scheduling_policy>-1</scheduling_policy>
492     <priority>0</priority>
493     <affinity>0</affinity>
494     <stack_size>-1</stack_size>
495 </security_log_thread>
496 </rtsp>
497 </participant>
498
499 <data_writer profile_name="datawriter_profile_example">
500   <topic>
501     <historyQos>
502       <kind>KEEP_LAST</kind>
503       <depth>20</depth>
504     </historyQos>
505     <resourceLimitsQos>
506       <max_samples>5</max_samples>
507       <max_instances>2</max_instances>
508       <max_samples_per_instance>1</max_samples_per_instance>
509       <allocated_samples>20</allocated_samples>
510       <extra_samples>10</extra_samples>
511     </resourceLimitsQos>
512   </topic>
513   <qos> <!-- dataWriterQosPoliciesType -->
514     <data_sharing>
515       <kind>AUTOMATIC</kind>
516       <shared_dir>/home</shared_dir>

```

(continues on next page)

(continued from previous page)

```

517         <max_domains>10</max_domains>
518         <domain_ids>
519             <domainId>0</domainId>
520             <domainId>11</domainId>
521         </domain_ids>
522     </data_sharing>
523     <deadline>
524         <period>
525             <sec>1</sec>
526         </period>
527     </deadline>
528     <disable_heartbeat_piggyback>true</disable_heartbeat_piggyback>
529     <disablePositiveAcks>
530         <enabled>true</enabled>
531         <duration>
532             <sec>1</sec>
533         </duration>
534     </disablePositiveAcks>
535     <durability>
536         <kind>VOLATILE</kind>
537     </durability>
538     <!-- QoS policy pending implementation -->
539     <latencyBudget>
540         <duration>
541             <sec>1</sec>
542         </duration>
543     </latencyBudget>
544     <lifespan>
545         <duration>
546             <sec>5</sec>
547         </duration>
548     </lifespan>
549     <liveliness>
550         <kind>AUTOMATIC</kind>
551         <lease_duration>
552             <sec>1</sec>
553             <nanosec>856000</nanosec>
554         </lease_duration>
555         <announcement_period>
556             <sec>1</sec>
557             <nanosec>856000</nanosec>
558         </announcement_period>
559     </liveliness>
560     <ownership>
561         <kind>EXCLUSIVE</kind>
562     </ownership>
563     <ownershipStrength>
564         <value>50</value>
565     </ownershipStrength>
566     <partition>
567         <names>
568             <name>part1</name>

```

(continues on next page)

(continued from previous page)

```

569         <name>part2</name>
570     </names>
571 </partition>
572 <publishMode>
573     <kind>ASYNCHRONOUS</kind>
574 </publishMode>
575 <reliability>
576     <kind>BEST_EFFORT</kind>
577     <max_blocking_time>
578         <sec>1</sec>
579         <nanosec>856000</nanosec>
580     </max_blocking_time>
581 </reliability>
582 </qos>
583
584 <times>
585     <initialHeartbeatDelay>
586         <sec>1</sec>
587         <nanosec>856000</nanosec>
588     </initialHeartbeatDelay>
589     <heartbeatPeriod>
590         <sec>1</sec>
591         <nanosec>856000</nanosec>
592     </heartbeatPeriod>
593     <nackResponseDelay>
594         <sec>1</sec>
595         <nanosec>856000</nanosec>
596     </nackResponseDelay>
597     <nackSupressionDuration>
598         <sec>1</sec>
599         <nanosec>856000</nanosec>
600     </nackSupressionDuration>
601 </times>
602
603 <unicastLocatorList>
604     <locator>
605         <udp4>
606             <!-- Access as physical, like UDP -->
607             <port>7400</port>
608             <address>192.168.1.41</address>
609         </udp4>
610     </locator>
611     <locator>
612         <tcpv4>
613             <!-- Both physical and logical (port), like TCP -->
614             <physical_port>5100</physical_port>
615             <port>7400</port>
616             <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>
617             <wan_address>80.80.99.45</wan_address>
618             <address>192.168.1.55</address>
619         </tcpv4>
620     </locator>

```

(continues on next page)

(continued from previous page)

```

621         <locator>
622             <udp6>
623                 <port>8844</port>
624                 <address>::1</address>
625             </udp6>
626         </locator>
627     </unicastLocatorList>
628
629     <multicastLocatorList>
630         <locator>
631             <udp4>
632                 <!-- Access as physical, like UDP -->
633                 <port>7400</port>
634                 <address>192.168.1.41</address>
635             </udp4>
636         </locator>
637         <locator>
638             <tcp4>
639                 <!-- Both physical and logical (port), like TCP -->
640                 <physical_port>5100</physical_port>
641                 <port>7400</port>
642                 <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>
643                 <wan_address>80.80.99.45</wan_address>
644                 <address>192.168.1.55</address>
645             </tcp4>
646         </locator>
647         <locator>
648             <udp6>
649                 <port>8844</port>
650                 <address>::1</address>
651             </udp6>
652         </locator>
653     </multicastLocatorList>
654
655     <external_unicast_locators>
656         <udp4 externality="1" cost="0" mask="24">
657             <address>100.100.100.10</address>
658             <port>12345</port>
659         </udp4>
660     </external_unicast_locators>
661
662     <ignore_non_matching_locators>true</ignore_non_matching_locators>
663     <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
664
665     <propertiesPolicy>
666         <properties>
667             <property>
668                 <name>Property1Name</name>
669                 <value>Property1Value</value>
670                 <propagate>>false</propagate>
671             </property>
672             <property>

```

(continues on next page)

(continued from previous page)

```

673         <name>Property2Name</name>
674         <value>Property2Value</value>
675         <propagate>>false</propagate>
676     </property>
677 </properties>
678 </propertiesPolicy>
679
680 <userDefinedID>45</userDefinedID>
681 <entityID>76</entityID>
682
683 <matchedSubscribersAllocation>
684     <initial>3</initial>
685     <maximum>3</maximum>
686     <increment>0</increment>
687 </matchedSubscribersAllocation>
688 </data_writer>
689
690 <data_reader profile_name="datareader_profile_example">
691     <topic>
692         <historyQos>
693             <kind>KEEP_LAST</kind>
694             <depth>20</depth>
695         </historyQos>
696         <resourceLimitsQos>
697             <max_samples>5</max_samples>
698             <max_instances>2</max_instances>
699             <max_samples_per_instance>1</max_samples_per_instance>
700             <allocated_samples>20</allocated_samples>
701             <extra_samples>10</extra_samples>
702         </resourceLimitsQos>
703     </topic>
704     <qos> <!-- dataReaderQosPoliciesType -->
705         <data_sharing>
706             <kind>AUTOMATIC</kind>
707             <shared_dir>/home</shared_dir>
708             <max_domains>10</max_domains>
709             <domain_ids>
710                 <domainId>0</domainId>
711                 <domainId>11</domainId>
712             </domain_ids>
713             <data_sharing_listener_thread>
714                 <scheduling_policy>-1</scheduling_policy>
715                 <priority>0</priority>
716                 <affinity>0</affinity>
717                 <stack_size>-1</stack_size>
718             </data_sharing_listener_thread>
719         </data_sharing>
720         <deadline>
721             <period>
722                 <sec>1</sec>
723             </period>
724         </deadline>

```

(continues on next page)

(continued from previous page)

```

725     <disablePositiveAcks>
726         <enabled>true</enabled>
727         <duration>
728             <sec>1</sec>
729         </duration>
730     </disablePositiveAcks>
731     <durability>
732         <kind>PERSISTENT</kind>
733     </durability>
734     <!-- QoS policy pending implementation -->
735     <latencyBudget>
736         <duration>
737             <sec>1</sec>
738         </duration>
739     </latencyBudget>
740     <lifespan>
741         <duration>
742             <sec>5</sec>
743         </duration>
744     </lifespan>
745     <liveliness>
746         <kind>MANUAL_BY_PARTICIPANT</kind>
747         <lease_duration>
748             <sec>1</sec>
749             <nanosec>856000</nanosec>
750         </lease_duration>
751         <announcement_period>
752             <sec>1</sec>
753             <nanosec>856000</nanosec>
754         </announcement_period>
755     </liveliness>
756     <ownership>
757         <kind>EXCLUSIVE</kind>
758     </ownership>
759     <partition>
760         <names>
761             <name>part1</name>
762             <name>part2</name>
763         </names>
764     </partition>
765     <reliability>
766         <kind>BEST_EFFORT</kind>
767         <max_blocking_time>
768             <sec>1</sec>
769             <nanosec>856000</nanosec>
770         </max_blocking_time>
771     </reliability>
772 </qos>
773
774 <times>
775     <initialAcknackDelay>
776         <sec>1</sec>

```

(continues on next page)

(continued from previous page)

```

777         <nanosec>856000</nanosec>
778     </initialAcknackDelay>
779     <heartbeatResponseDelay>
780         <sec>1</sec>
781         <nanosec>856000</nanosec>
782     </heartbeatResponseDelay>
783 </times>
784
785 <unicastLocatorList>
786     <locator>
787         <udp4>
788             <!-- Access as physical, like UDP -->
789             <port>7400</port>
790             <address>192.168.1.41</address>
791         </udp4>
792     </locator>
793     <locator>
794         <tcp4>
795             <!-- Both physical and logical (port), like TCP -->
796             <physical_port>5100</physical_port>
797             <port>7400</port>
798             <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>
799             <wan_address>80.80.99.45</wan_address>
800             <address>192.168.1.55</address>
801         </tcp4>
802     </locator>
803     <locator>
804         <udp6>
805             <port>8844</port>
806             <address>::1</address>
807         </udp6>
808     </locator>
809 </unicastLocatorList>
810
811 <multicastLocatorList>
812     <locator>
813         <udp4>
814             <!-- Access as physical, like UDP -->
815             <port>7400</port>
816             <address>192.168.1.41</address>
817         </udp4>
818     </locator>
819     <locator>
820         <tcp4>
821             <!-- Both physical and logical (port), like TCP -->
822             <physical_port>5100</physical_port>
823             <port>7400</port>
824             <unique_lan_id>192.168.1.1.1.2.55</unique_lan_id>
825             <wan_address>80.80.99.45</wan_address>
826             <address>192.168.1.55</address>
827         </tcp4>
828     </locator>

```

(continues on next page)

(continued from previous page)

```

829         <locator>
830             <udp6>
831                 <port>8844</port>
832                 <address>::1</address>
833             </udp6>
834         </locator>
835     </multicastLocatorList>
836
837     <external_unicast_locators>
838         <udp4 externality="1" cost="0" mask="24">
839             <address>100.100.100.10</address>
840             <port>12345</port>
841         </udp4>
842     </external_unicast_locators>
843
844     <ignore_non_matching_locators>true</ignore_non_matching_locators>
845     <expectsInlineQos>true</expectsInlineQos>
846     <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
847
848     <propertiesPolicy>
849         <properties>
850             <property>
851                 <name>Property1Name</name>
852                 <value>Property1Value</value>
853                 <propagate>>false</propagate>
854             </property>
855             <property>
856                 <name>Property2Name</name>
857                 <value>Property2Value</value>
858                 <propagate>>false</propagate>
859             </property>
860         </properties>
861     </propertiesPolicy>
862
863     <userDefinedID>55</userDefinedID>
864     <entityID>66</entityID>
865
866     <matchedPublishersAllocation>
867         <initial>1</initial>
868         <maximum>1</maximum>
869         <increment>0</increment>
870     </matchedPublishersAllocation>
871 </data_reader>
872
873 <topic profile_name="topic_profile_example">
874     <historyQos>
875         <kind>KEEP_LAST</kind>
876         <depth>20</depth>
877     </historyQos>
878     <resourceLimitsQos>
879         <max_samples>5</max_samples>
880         <max_instances>2</max_instances>

```

(continues on next page)

(continued from previous page)

```

881         <max_samples_per_instance>1</max_samples_per_instance>
882         <allocated_samples>20</allocated_samples>
883         <extra_samples>10</extra_samples>
884     </resourceLimitsQos>
885 </topic>
886 </profiles>
887
888 <library_settings>
889     <intraprocess_delivery>USER_DATA_ONLY</intraprocess_delivery>
890 </library_settings>
891
892 <log>
893     <use_default>>false</use_default>
894
895     <consumer>
896         <class>StdoutConsumer</class>
897     </consumer>
898
899     <consumer>
900         <class>StdoutErrConsumer</class>
901         <property>
902             <name>stderr_threshold</name>
903             <value>Log::Kind::Warning</value>
904         </property>
905     </consumer>
906
907     <consumer>
908         <class>FileConsumer</class>
909         <property>
910             <name>filename</name>
911             <value>execution.log</value>
912         </property>
913         <property>
914             <name>append</name>
915             <value>TRUE</value>
916         </property>
917     </consumer>
918
919     <thread_settings>
920         <scheduling_policy>-1</scheduling_policy>
921         <priority>0</priority>
922         <affinity>0</affinity>
923         <stack_size>-1</stack_size>
924     </thread_settings>
925 </log>
926
927 <types>
928     <type> <!-- Types can be defined in its own type of tag or sharing the same tag -
929     <enum name="MyAloneEnumType">
930         <enumerator name="A" value="0"/>
931         <enumerator name="B" value="1"/>

```

(continues on next page)

(continued from previous page)

```

932         <enumerator name="C"/>
933     </enum>
934 </type>
935 <type>
936     <!-- All possible members struct type -->
937     <struct name="MyFullStruct">
938         <!-- Primitives & basic -->
939         <member name="my_bool" type="boolean"/>
940         <member name="my_char" type="char8"/>
941         <member name="my_wchar" type="char16"/>
942         <member name="my_byte" type="byte"/>
943         <member name="my_octet" type="octet"/>
944         <member name="my_uint8" type="uint8"/>
945         <member name="my_short" type="int16"/>
946         <member name="my_long" type="int32"/>
947         <member name="my_unsignedshort" type="uint16"/>
948         <member name="my_unsignedlong" type="uint32"/>
949         <member name="my_longlong" type="int64"/>
950         <member name="my_unsignedlonglong" type="uint64"/>
951         <member name="my_float" type="float32"/>
952         <member name="my_double" type="float64"/>
953         <member name="my_longdouble" type="float128"/>
954         <member name="my_string" type="string"/>
955         <member name="my_wstring" type="wstring"/>
956
957         <!-- string my_boundedString[41925] -->
958         <member name="my_boundedString" type="string" stringMaxLength="41925"/>
959         <!-- wstring my_boundedWString[41925] -->
960         <member name="my_boundedWString" type="wstring" stringMaxLength="41925"/>
961
962         <!-- short short_sequence[5]; -->
963         <member name="short_sequence" sequenceMaxLength="5" type="int16"/>
964
965         <!-- long long_array[2][3][4]; -->
966         <member name="long_array" arrayDimensions="2,3,4" type="int32"/>
967
968         <!-- map<long,long,2> my_map_inner -->
969         <member name="my_map" type="int32" key_type="int32" mapMaxLength="2"/>
970     </struct>
971
972     <typedef name="inner_map" type="char8" key_type="int16"/>
973     <struct name="MyComplexStruct">
974         <!-- Complex types -->
975         <member name="my_other_struct" type="nonBasic" nonBasicTypeName=
976 ↪ "MyFullStruct"/>
977         <!-- map<long,map<long,long,2>,2> my_map_map; -->
978         <member name="my_map_map" type="nonBasic" nonBasicTypeName="inner_map"
979 ↪ key_type="int32" mapMaxLength="2"/>
980     </struct>
981
982     <enum name="MyEnum">
983         <enumerator name="A" value="0"/>

```

(continues on next page)

(continued from previous page)

```

982         <enumerator name="B" value="1"/>
983         <enumerator name="C"/>
984     </enum>
985
986     <typedef name="MyAlias1" type="nonBasic" nonBasicTypeName="MyEnum"/>
987     <typedef name="MyAlias2" type="int32" arrayDimensions="2,2"/>
988
989     <struct name="MyStruct">
990         <member name="first" type="int32"/>
991         <member name="second" type="int64"/>
992     </struct>
993
994     <!-- TODO(XTypes: Fix inheritance loading from XML profile) Fast DDS#4626 -->
995     <!-- <struct name="OtherInheritedStruct" baseType="MyStruct">
996         <member name="my_enum" type="nonBasic" nonBasicTypeName="MyEnum"/>
997         <member name="my_struct" type="nonBasic" nonBasicTypeName="MyFullStruct"
998         ↪arrayDimensions="5"/>
999     </struct> -->
1000
1001     <union name="MyUnion1">
1002         <discriminator type="byte"/>
1003         <case>
1004             <caseDiscriminator value="0"/>
1005             <caseDiscriminator value="1"/>
1006             <member name="first" type="int32"/>
1007         </case>
1008         <case>
1009             <caseDiscriminator value="2"/>
1010             <member name="second" type="nonBasic" nonBasicTypeName="MyStruct"/>
1011         </case>
1012         <case>
1013             <caseDiscriminator value="default"/>
1014             <member name="third" type="int64"/>
1015         </case>
1016     </union>
1017
1018     <bitset name="MyBitSet">
1019         <bitfield name="a" bit_bound="3"/>
1020         <bitfield name="b" bit_bound="10"/>
1021         <bitfield name="c" bit_bound="12" type="int16"/>
1022     </bitset>
1023
1024     <!-- TODO(XTypes: Fix inheritance loading from XML profile) Fast DDS#4626 -->
1025     <!-- <bitset name="OtherInheritedBitSet" baseType="MyBitSet">
1026         <bitfield bit_bound="8"/>
1027         <bitfield bit_bound="15" type="byte"/>
1028     </bitset> -->
1029
1030     <bitmask name="MyBitMask" bit_bound="8">
1031         <bit_value name="flag0" position="0"/>
1032         <bit_value name="flag1"/>
1033     </bitmask>

```

(continues on next page)

(continued from previous page)

```
1033     </type>
1034   </types>
1035 </dds>
```

7.26 Environment variables

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

7.26.1 FASTDDS_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 FASTDDS_DEFAULT_PROFILES_FILE=/home/user/profiles.xml
```

Windows

```
set FASTDDS_DEFAULT_PROFILES_FILE=C:\profiles.xml
```

7.26.2 SKIP_DEFAULT_XML

Skips looking for a default profile configuration XML file. If this variable is set to *1*, *Fast DDS* will load the configuration parameters directly from the classes' definitions without looking for the *DEFAULT_FASTDDS_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
```

7.26.3 FASTDDS_BUILTIN_TRANSPORTS

Setting this variable allows to modify the builtin transports that are initialized during the *DomainParticipant* creation. It is a simple way of changing the default configuration of the *Transport Layer* and it directly affects how DDS entities communicate between them.

All existing values, along with a brief description, are shown below:

Builtin Trans-ports Op-tions	Description
NONE	No transport will be instantiated. Hence, the user must manually add the desired transports. Otherwise, the participant creation will fail.
DEFAULT	UDPv4 and SHM transports will be instantiated. SHM transport has priority over the UDPv4 transport. Meaning that SHM will always be used when possible.
DEFAULT	UDPv6 and SHM transports will be instantiated. SHM transport has priority over the UDPv4 transport. Meaning that SHM will always be used when possible.
SHM	Only a SHM transport will be instantiated.
UDPv4	Only a UDPv4 transport will be instantiated.
UDPv6	Only a UDPv6 transport will be instantiated.
LARGE_D	UDPv4, TCPv4, and SHM transports will be instantiated. However, UDP will only be used for multi-cast announcements during the participant discovery phase (see <i>Discovery phases</i>) while the participant liveliness and the application data delivery occurs over TCP or SHM. This configuration is useful when working with large data.(See <i>Large Data mode and Fast DDS over TCP</i>).

Note: The environment variable is only used in the case where `use_builtin_transports` is set to `TRUE`. In any other case, the environment variable has no effect.

Note: TCPv4 transport is initialized with the following configuration:

- `calculate_crc`, `check_crc` and `apply_security` are set to `false`.
- `enable_tcp_nodelay` is set to `true`.
- `keep_alive_thread` and `accept_thread` use the default configuration.

Configuring builtin transports options

Fast DDS offers a straightforward method to configure three main parameters of every builtin transport via the environment variable. However, this feature proves particularly valuable when employing the `LARGE_DATA` builtin transports option. The `LARGE_DATA` mode has been designed to improve performance when working with large data. However, according to each specific use case, the user might want to configure several options to better fit their needs. This mode can also be configured with the `tcp_negotiation_timeout` parameter:

Builtin Transports Options	Description	Type
<code>max_msg_size</code>	It determines the maximum message size that will be specified in the transport layer. Selecting a message size large enough to accommodate the largest data message will prevent fragmentation, which can significantly enhance the overall sending rate.	<code>uint32_t</code>
<code>sockets_size</code>	It determines the size of the send and receive socket buffers. This parameter needs to be higher or equal to the maximum message size specified in order to be valid.	<code>uint32_t</code>
<code>non_blocking</code>	It determines whether to use non-blocking send calls or not. When activated, the transport will discard messages if the socket buffers are full.	<code>bool</code>
<code>tcp_negotiation_timeout</code>	It determines the time to wait for logical port negotiation. Only valid if the <code>LARGE_DATA</code> mode is being used. It only accepts milliseconds.	<code>uint32_t</code>

The environment variable accepts several types of units to specify the values of the parameters. Also, both lowercase and uppercase letters are valid. The following list shows the available units and their corresponding symbols:

- B: Bytes. This is the default unit, so it is not necessary to specify it.
- KB: Kilobytes.
- MB: Megabytes.
- GB: Gigabytes.
- KiB: Kibibyte.
- MiB: Mebibyte.
- GiB: Gibibyte.

```
export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA?max_msg_size=200KB&sockets_size=1MB&non_blocking=true&tcp_negotiation_timeout=50
```

Note: When working with `LARGE_DATA`, it is recommended to set the `max_msg_size` and `sockets_size` to a value large enough to accommodate the largest data message and to set the `non_blocking` to `TRUE`. Note that activating the `non_blocking` option with a small message size (with fragmentation) can lead to an increase of messages drop rate and produce undesired results. For more information, please refer to [Large Data with configuration options](#).

Warning: Setting a `max_msg_size` higher than 65500 KB is only possible when using the `LARGE_DATA` builtin transports option. Trying to set it with any other builtin transports will result in an error and the creation of the participant will fail.

7.26.4 ROS_DISCOVERY_SERVER

Warning: The environment variable is only used in the case where *discovery protocol* is set to SIMPLE, SERVER, or BACKUP. In any other case, the environment variable has no effect.

Setting this variable configures the *DomainParticipant* to connect to one or more *servers* using the *Discovery Server* discovery mechanism.

- If ROS_DISCOVERY_SERVER is defined, and the DomainParticipant's *discovery protocol* is set to SIMPLE, then Fast DDS will instead configure it as CLIENT of the given *server*.
- If ROS_DISCOVERY_SERVER is defined, and the DomainParticipant's *discovery protocol* is set to SERVER or BACKUP, then the variable is used to add remote *servers* to the given *server*, leaving the *discovery protocol* as SERVER or BACKUP respectively.
- The value of the variable must list the locator of the server in the form of:
 - An IPv4 address like 192.168.2.23. The UDP protocol is used by default. The UDP port can be appended using : as in 192.168.2.23:35665.
 - An IPv6 address that follows RFC3513 address convention like 1080::8:800:200C:417A. Again, it uses the UDP protocol by default. An UDP port can be appended like in [1080::8:800:200C:417A]:35665. Note the use of square brackets to avoid ambiguities.
 - TCPv4 specifier + IPv4 address like TCPv4:[127.0.0.1]. The TCP protocol is used to communicate with the server. The TCP port can be appended using : as in TCPv4:[127.0.0.1]:42100.
 - TCPv6 specifier + IPv6 address like TCPv6:[::1]. The TCP protocol is used to communicate with the server. The TCP port can be appended using : as in TCPv6:[::1]:42100.
 - A DNS name can be specified. This name will be used to query known hosts and available DNS servers to try to resolve valid IP addresses. Several formats are acceptable:
 - * Plain domain name: eprosima.com. This will include all available IP addresses.
 - * Domain name + port: eprosima.com:35665. As above but using a specific port.
 - * UDPv4 specifier + domain name: UDPv4:[eprosima.com]. Only the first IPv4 address resolved will be used.
 - * UDPv4 specifier + domain name + port: UDPv4:[eprosima.com]:35665. As above but using a specific port.
 - * UDPv6 specifier + domain name: UDPv6:[<dns>]. Only the first IPv6 address resolved will be used.
 - * UDPv6 specifier + domain name + port: UDPv6:[<dns>]:35665. As above but using a specific port.
 - * TCPv4 specifier + domain name: TCPv4:[eprosima.com]. Only the first IPv4 address resolver will be used.
 - * TCPv4 specifier + domain name + port: TCPv4:[eprosima.com]:42100. As above but using a specific port.
 - * TCPv6 specifier + domain name: TCPv6:[<dns>]. Only the first IPv4 address resolver will be used.
 - * TCPv6 specifier + domain name + port: TCPv6:[<dns>]:42100. As above but using a specific port.
- If no port is specified when using default UDP transport, the default port 11811 is used.
- If no port is specified when using TCP transport, the default port 42100 is used.
- To set more than one *server*'s address, they must be separated by semicolons.

- The server's ID is determined by their position in the list. Two semicolons together means the corresponding ID is free.
- When using IPv6 with DNS, the specified domain name space (<*dns*>) must be able to resolve to an IPv6 address. Otherwise an error will be raised.

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

Linux

```
export ROS_DISCOVERY_SERVER="84.22.259.329:8888;;localhost:1234"
```

Windows

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

Important: IP addresses specified in `ROS_DISCOVERY_SERVER` must be either valid IPv4/IPv6 addresses or domain names. If a name can be resolved into several addresses, it is possible to either use them all or restrict the selection to the first IPv4 using the *UDPv4*: or *TCPv4* prefixes or to the first IPv6 address using the *UDPv6*: or *TCPv6* prefixes.

Important: This environment variable is meant to be used in combination with *Fast DDS discovery CLI*. The *server*'s ID is used by *Fast DDS* to derived the `GuidPrefix_t` of the *server*. If the *server* is not instantiated using the CLI, the *server*'s GUID prefix should adhere to the same schema as the one generated from the CLI. Else, the *clients* configured with this environment variable will not be able to establish a connection with the *server*, thus not being able to connect to other *clients* either. The *server*'s GUID prefixes generated by the CLI comply with the following schema: 44.53.<server-id-in-hex>.5f.45.50.52.4f.53.49.4d.41. This prefix schema has been chosen for its ASCII translation: DS<id_in_hex>_EPROSIMA.

Important: This environment variable can be changed at runtime adding new remote servers to a `SERVER`, `BACKUP` or `CLIENT` (that has been initialized with this environment variable previously) if loaded from an environment file using *FASTDDS_ENVIRONMENT_FILE*.

7.26.5 ROS_SUPER_CLIENT

If the `DomainParticipant`'s *discovery protocol* is set to `SIMPLE`, and `ROS_SUPER_CLIENT` is set to `TRUE`, the participant is automatically promoted to a `SUPER_CLIENT`.

Important: This environment variable is meant to be used in combination with `ROS_DISCOVERY_SERVER` to promote a participant from `SIMPLE` to `SUPER_CLIENT`. The participant will have the *servers* list defined in `ROS_DISCOVERY_SERVER`.

The possible values are: `TRUE`, `true`, `True`, `1`, `FALSE`, `false`, `False`, `0`.

Important: If the variable is not set, the default behavior of Fast DDS is equivalent to the case in which the variable is set to false.

The following example shows how to set the environment variable to true.

Linux
<code>export ROS_SUPER_CLIENT=TRUE</code>
Windows
<code>set ROS_SUPER_CLIENT=TRUE</code>

7.26.6 FASTDDS_STATISTICS

Warning: The environment variable is only used in the case where the CMake option *FASTDDS_STATISTICS* has been enabled. In any other case, the environment variable has no effect. Please, refer to *CMake options* for more information.

Setting this variable configures the *DomainParticipant* to enable the statistics DataWriters which topics are contained in the list set in this environment variable. The elements of the list should be separated by semicolons and match the *statistics topic name aliases*.

For example, to enable the statistics DataWriters that report the latency measurements, the environment variable should be set as follows:

Linux
<code>export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC"</code>
Windows
<code>set FASTDDS_STATISTICS=HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC</code>

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).

7.26.7 FASTDDS_ENVIRONMENT_FILE

Setting this environment variable to an existing json file allows to load the environment variables from the file instead of from the environment. This allows to change the value of some environment variables at run time with just modifying and saving the changes to the file. The environment value can be either an absolute or relative path. The file format is as follows:

```
{
  "environment_variable_name_1": "environment_variable_value_1",
  "environment_variable_name_2": "environment_variable_value_2"
}
```

Important: The environment variables set in the environment file have precedence over the environment.

Warning: Currently only `ROS_DISCOVERY_SERVER` environment variable allows for changes at run time. (see *Modifying remote servers list at run time*)

7.27 PropertyPolicyQos Options

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

7.27.1 Non consolidated QoS

The *PropertyPolicyQos Options* are used to develop new *eProsima Extensions* QoS. Before consolidating a new QoS Policy, it is usually set using this generic QoS Policy. Consequently, this section is prone to frequent updates so the user is advised to check latest changes after upgrading to a different release version.

DataWriter operating mode QoS Policy

By default, *Fast DDS DataWriters* are enabled using `push` mode. This implies that they will add new samples into their queue, and then immediately deliver them to matched readers. For writers that produce non periodic bursts of data, this may imply saturating the network with a lot of packets, increasing the possibility of losing them on unreliable (i.e. UDP) transports. Depending on their QoS, DataReaders may also have to ignore some received samples, so they will have to be resent.

Configuring the DataWriters on `pull` mode offers an alternative by letting each reader pace its own data stream. It works by the writer notifying the reader what it is available, and waiting for it to request only as much as it can handle. At the cost of greater latency, this model can deliver reliability while using far fewer packets than `push` mode.

DataWriters periodically announce the state of their queue by means of a heartbeat. Upon reception of the heartbeat, DataReaders will request the DataWriter to send the samples they want to process. Consequently, the publishing rate can be tuned setting the heartbeat period accordingly. See *Tuning Heartbeat Period* for more details.

PropertyPolicyQos name	PropertyPolicyQos value	Default value
"fastdds.push_mode"	"true"/"false"	"true"

C++

```
DataWriterQos wqos;

// Enable pull mode
wqos.properties().properties().emplace_back(
    "fastdds.push_mode",
    "false");
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <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 <i>statistics topic name aliases</i>	""

C++

```
DomainParticipantQos pqos;

// Activate Fast DDS Statistics module
pqos.properties().properties().emplace_back("fastdds.statistics",
    "HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC
→");
```

XML

```
<participant profile_name="statistics_domainparticipant_conf_xml_profile">
  <rtps>
    <propertiesPolicy>
      <properties>
        <!-- Activate Fast DDS Statistics Module -->
        <property>
          <name>fastdds.statistics</name>
          <value>HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
→PHYSICAL_DATA_TOPIC</value>
        </property>
      </properties>
    </propertiesPolicy>
  </rtps>
</participant>
```

Physical Data in Discovery Information

It is possible to include the information conveyed in the *PHYSICAL_DATA_TOPIC* into the participant discovery message, a.k.a *DATA[p]* (see *Discovery phases*). This is done by setting the following properties within the *PropertyPolicyQos*:

PropertyPolicyQos name	PropertyPolicyQos value	Default value without FASTDDS_STATISTICS	Default value with FASTDDS_STATISTICS
"fastdds.physical_data.host"	Name of the host computer in which the application runs	Not set	""
"fastdds.physical_data.user"	Name of the user running the application	Not set	""
"fastdds.physical_data.process"	Name of the process running the application	Not set	""

Whenever any of these properties is defined within the *DomainParticipantQos*, the *DomainParticipant DATA[p]*

will contained the set value. Furthermore, if any of these properties is set to a value of "", which is the default when FASTDDS_STATISTICS is defined (see *CMake options*), *Fast DDS* will automatically populate the value using the following convention:

- "fastdds.physical_data.host": Host name as returned by `asio::ip::host_name()`, followed by " :<default data sharing domain id>"
- "fastdds.physical_data.user": Name of the user running the application, or "unknown" if it could not be retrieved.
- "fastdds.physical_data.process": The process ID of the process in which the application is running.

All the previous entails that adding physical information to the *DATA[p]* can be done regardless of whether FASTDDS_STATISTICS is defined, and that it is possible to let *Fast DDS* set some default values into the reported host, user, and process:

1. If FASTDDS_STATISTICS is defined, and the user does not specify otherwise, *Fast DDS* will set default values to the physical properties of the *DATA[p]*.
2. If FASTDDS_STATISTICS is defined, and the user sets values to the properties, the user settings are honored.
3. If FASTDDS_STATISTICS is defined, and the user removes the physical properties from the *DomainParticipantQos*, then no physical information is transmitted in the *DATA[p]*.
4. If FASTDDS_STATISTICS is not defined, it is still possible to transmit physical information in the *DATA[p]* by setting the aforementioned properties:
 - a) If set to "", then *Fast DDS* will populate their value according to the described rules.
 - b) If set to something other than "", then the set value will be transmitted in the *DATA[p]* as-is.

In case FASTDDS_STATISTICS is defined, and the reporting of statistics over the DISCOVERY_TOPIC is enabled (see *Statistics Module Settings*), then the physical information included in the *DATA[p]* is also transmitted over the DISCOVERY_TOPIC (see *PHYSICAL_DATA_TOPIC*) whenever one *DomainParticipant* discovers another one.

C++

```
/* Create participant which announces default physical properties */
DomainParticipantQos pqos_default_physical;
// NOTE: If FASTDDS_STATISTICS is defined, then setting the properties to "" is not
↳ necessary
pqos_default_physical.properties().properties().emplace_back("fastdds.physical_data.host", "");
pqos_default_physical.properties().properties().emplace_back("fastdds.physical_data.user", "");
pqos_default_physical.properties().properties().emplace_back("fastdds.physical_data.process", "");
DomainParticipant* participant_with_physical = DomainParticipantFactory::get_instance()->
↳ create_participant(0,
                    pqos_default_physical);

/* Create participant which announces custom physical properties */
DomainParticipantQos pqos_custom_physical;
// NOTE: If FASTDDS_STATISTICS is defined, then clear the properties before setting them
// pqos_custom_physical.properties().properties().clear()
pqos_custom_physical.properties().properties().emplace_back("fastdds.physical_data.host",
↳ "custom_hostname");
pqos_custom_physical.properties().properties().emplace_back("fastdds.physical_data.user",
↳ "custom_username");
```

(continues on next page)

(continued from previous page)

```

pqos_custom_physical.properties().properties().emplace_back("fastdds.physical_data.
↳process", "custom_process");
DomainParticipant* participant_custom_physical = DomainParticipantFactory::get_
↳instance()->create_participant(0,
    pqos_custom_physical);

/* Create participant which does not announce physical properties */
DomainParticipantQos pqos_no_physical;
pqos_no_physical.properties().properties().clear();
DomainParticipant* participant_without_physical = DomainParticipantFactory::get_
↳instance()->create_participant(
    0, pqos_no_physical);

/* Load physical properties from default XML file */
DomainParticipantFactory::get_instance()->load_profiles();
DomainParticipantQos pqos_default_xml_physical =
    DomainParticipantFactory::get_instance()->get_default_participant_qos();
DomainParticipant* participant_default_xml_physical =
    DomainParticipantFactory::get_instance()->create_participant(0, pqos_default_xml_
↳physical);

/* Load physical properties from specific XML file */
DomainParticipantFactory::get_instance()->load_XML_profiles_file("somefile.xml");
DomainParticipantFactory::get_instance()->load_profiles();
DomainParticipantQos pqos_custom_xml_physical =
    DomainParticipantFactory::get_instance()->get_default_participant_qos();
DomainParticipant* participant_custom_xml_physical =
    DomainParticipantFactory::get_instance()->create_participant(0, pqos_custom_xml_
↳physical);

```

XML

```

<?xml version="1.0" encoding="utf-8"?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <participant profile_name="statistics_participant" is_default_profile="true">
      <rtps>
        <propertiesPolicy>
          <properties>
            <property>
              <name>fastdds.physical_data.host</name>
              <value>custom_hostname</value>
            </property>
            <property>
              <name>fastdds.physical_data.user</name>
              <value>custom_username</value>
            </property>
            <property>
              <name>fastdds.physical_data.process</name>
              <value>custom_process</value>
            </property>
          </properties>
        </propertiesPolicy>
      </rtps>
    </participant>
  </profiles>

```

(continues on next page)

(continued from previous page)

```
        </propertiesPolicy>
    </rtps>
</participant>
</profiles>
</dds>
```

Important: The properties set using XML override those in the default QoS, which means that it is possible to set the physical properties using XML regardless of whether FASTDDS_STATISTICS is defined. However, it is not possible to remove the properties using XML, meaning that an application using *Fast DDS* with FASTDDS_STATISTICS enabled which does not want for the physical information to be transmitted in the DomainParticipant *DATA[p]* must remove the properties using the aforementioned C++ API.

Endpoint Partitions

Fast DDS uses this *PropertyPolicyQos* to define which partitions does an endpoint belong to. This property follows the same logic regarding matching as the *PartitionQosPolicy* that can be defined for Publishers and Subscribers.

This property’s value is a semicolon separated list containing the partition names the user wants this endpoint to belong to.

Important: If both a Publisher and one of its DataWriters have conflicting partition configuration, this is, a DataWriter has this property defined while the Publisher has the *PartitionQosPolicy* defined, the DataWriter configuration takes precedence and the Publisher *PartitionQosPolicy* is ignored for this endpoint. This applies to Subscribers and their DataReaders as well.

This property will be automatically set when creating DataReaders and DataWriters using the create_with_profile functions. It cannot be changed after the entity has been created.

PropertyPolicyQos name	PropertyPolicyQos value	Default value
"partitions"	Semicolon separated list of partition names	""

C++

```
DataWriterQos wqos;

// Add partitions
wqos.properties().properties().emplace_back(
    "partitions",
    "part1;part2");

DataReaderQos rqos;

// Add partitions
rqos.properties().properties().emplace_back(
    "partitions",
    "part1;part2");
```

XML

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

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

Static Discovery's Exchange Format

Static Discovery exchanges data in the Participant Discovery Phase (PDP). Currently there are two different exchange formats which can be selected using the property `dds.discovery.static_edp.exchange_format`.

PropertyPolicyQos value	Description	Default
"v1"	Standard exchange format for Static Discovery.	
"v1_Reduced"	Format which reduces the necessary network bandwidth to transmit Static Discovery's information in the Participant Discovery Phase (PDP).	

C++

```
DomainParticipantQos participant_qos;
participant_qos.properties().properties().emplace_back(
    "dds.discovery.static_edp.exchange_format",
    "v1_Reduced"
);
```

XML

```
<participant profile_name="participant_xml_conf_static_discovery_format_profile">
  <rtps>
    <propertiesPolicy>
      <properties>
        <property>
          <name>dds.discovery.static_edp.exchange_format</name>
          <value>v1_Reduced</value>
        </property>
      </properties>
    </propertiesPolicy>
  </rtps>
</participant>
```

SHM transport meta-traffic enforcement

A DomainParticipant will by default configure both a *UDP Transport* and a *Shared Memory Transport*. When a participant on another process in the same host is discovered, the endpoint discovery might be done using either transport.

Avoiding Shared Memory communication for discovery traffic can save valuable resources. The behavior regarding this can be configured using the property `fastdds.shm.enforce_metatraffic`.

Property-PolicyQos value	Description	De-fault
"none"	Use other transports for meta-traffic.	
"unicast"	Enable SHM transport unicast communications.	
"all"	Enable SHM transport unicast and multicast communications. This will enable discovery between SHM only participants and participants having several transports.	

Note: When SHM is the only transport configured for a participant, the setting of this property is ignored, and considered to be "all".

C++

```
DomainParticipantQos participant_qos;

// SHM transport will listen for unicast meta-traffic
participant_qos.properties().properties().emplace_back(
    "fastdds.shm.enforce_metatraffic",
    "unicast");
```

XML

```
<participant profile_name="participant_xml_conf_shm_enforce_metatraffic_profile">
  <rtps>
    <propertiesPolicy>
      <properties>
        <property>
          <name>fastdds.shm.enforce_metatraffic</name>
          <value>unicast</value>
        </property>
      </properties>
    </propertiesPolicy>
  </rtps>
</participant>
```

7.27.2 Flow Controller Settings

When using *Flow Controllers*, the DataWriter may need specific parameters to be set. Properties related with this feature lie on the `fastdds.sfc` namespace.

- Property `fastdds.sfc.priority` is used to set the priority of the DataWriter for `HIGH_PRIORITY` and `PRIORITY_WITH_RESERVATION` flow controllers. Allowed values are from -10 (highest priority) to 10 (lowest priority). If the property is not present, it will be set to the lowest priority.
- Property `fastdds.sfc.bandwidth_reservation` is used to set the percentage of the bandwidth that the DataWriter is requesting for `PRIORITY_WITH_RESERVATION` flow controllers. Allowed values are from 0 to 100, and express a percentage of the total flow controller limit. If the property is not present, it will be set to 0 (no bandwidth is reserved for the DataWriter).

7.27.3 Persistence Service Settings

Warning: This section is still under work.

7.27.4 Security Plugins Settings

As described in the *Security* section, the security plugins admit a set of settings that can be configured.

Authentication plugin settings

The `DDS:Auth:PKI-DH` authentication plugin, can be activated setting the `DomainParticipantQos` `properties()` `dds.sec.auth.plugin` with the value `builtin.PKI-DH`. The following table outlines the properties used for the `DDS:Auth:PKI-DH` plugin configuration.

PropertyPolicyQos name	PropertyPolicyQos value
<code>identity_ca</code>	URI to the X.509 v3 certificate of the Identity CA in PEM format. Supported URI schemes: file.
<code>identity_key</code>	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.
<code>identity_crl</code> (optional)	URI to a X.509 Certificate Revocation List (CRL). Supported URI schemes: file.
<code>private_key</code>	URI to access the private Private Key for the Participant. Supported URI schemes: file, <i>PKCS#11</i> .
<code>password</code> (optional)	A password used to decrypt the <i>private_key</i> . If the <i>password</i> property is not present, then the value supplied in the <i>private_key</i> property must contain the decrypted private key. The <i>password</i> property is ignored if the <i>private_key</i> is given in PKCS#11 scheme.

Note: All properties listed above have the `dds.sec.auth.builtin.PKI-DH.` prefix. For example: `dds.sec.auth.builtin.PKI-DH.identity_ca`. For examples and further information, please refer to the [Authentication plugin: DDS:Auth:PKI-DH](#) section.

Authentication handshake settings

The authentication phase starts when discovery information is received from the remote *DomainParticipants*. At this moment, the participant sends a handshake request until a handshake response is received from the remote participant. Some parameters are involved in the behavior of this exchange:

- `max_handshake_requests` controls the maximum number of handshake requests to be sent.
- `initial_handshake_resend_period` represents the initial waiting time (in milliseconds) for the first handshake request that has to be resent.
- `handshake_resend_period_gain` is the gain against which the period is multiplied between two handshake requests.

Hence, the period of time to wait for sending a new handshake request is computed at each iteration as the period between the last two handshake requests multiplied by the gain (so that the period increases).

The following table lists the settings to configure the authentication handshake behavior within the `dds.sec.auth.builtin.PKI-DH` plugin:

PropertyPolicyQos name	PropertyPolicyQos value	PropertyPolicyQos bounds	Default value
<code>max_handshake_requests</code>	<int>	[1, max)	10
<code>initial_handshake_resend_period</code>	<int>	[1, max)	125
<code>handshake_resend_period_gain</code>	<double>	(1.0, max)	1.5

Note: All listed properties have the `dds.sec.auth.builtin.PKI-DH.` prefix. For example: `dds.sec.auth.builtin.PKI-DH.max_handshake_requests`.

The following is an example of how to set the properties of `DomainParticipantQoS` for the authentication handshake configuration.

C++

```
DomainParticipantQos pqos;

pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.max_handshake_requests",
    "5");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.initial_handshake_resend_period",
    "250");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.handshake_resend_period_gain",
    "1.5");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_auth_handshake_props_xml_
↪profile">
  <rtps>
    <propertiesPolicy>
      <properties>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.max_handshake_requests</name>
          <value>5</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.initial_handshake_resend_period</
↪name>
          <value>250</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.handshake_resend_period_gain</
↪name>
          <value>1.5</value>
        </property>
      </properties>
    </propertiesPolicy>
  </rtps>
</participant>
```

Cryptographic plugin settings

The *DDS:Crypto:AES-GCM-GMAC* authentication plugin, can be activated setting the *DomainParticipantQos* properties() `dds.sec.crypto.plugin` with the value `builtin.AES-GCM-GMAC`. Moreover, this plugin needs the activation of the *Authentication plugin: DDS:Auth:PKI-DH*. The *DDS:Crypto:AES-GCM-GMAC* plugin is configured using the *Access control plugin: DDS:Access:Permissions*, i.e the cryptography plugin is configured through the properties and configuration files of the access control plugin. For further information and examples in this regard please refer to *Cryptographic plugin: DDS:Crypto:AES-GCM-GMAC*.

Logging plugin settings

The *DDS:Logging:DDS_LogTopic* authentication plugin, can be activated setting the *DomainParticipantQos* properties() `dds.sec.log.plugin` with the value `builtin.DDS_LogTopic`. The following table outlines the properties used for the *DDS:Logging:DDS_LogTopic* plugin configuration. For further information and examples follow the dedicated documentation: *Logging plugin: DDS:Logging:DDS_LogTopic*.

7.27.5 Logging Module Settings

Warning: This section is still under work.

7.27.6 Ignore Local Endpoints

By default, Fast DDS will automatically match all the endpoints (meaning *DataReaders* and *DataWriters*) belonging to a given *DomainParticipant* as soon as they share the same *Topic* and have compatible Qos. This however can result in undesired feedback whenever an application creates a *DataReader* and a *DataWriter* under the same *DomainParticipant* on a shared *Topic*. Although this feedback can be filtered out at the application level upon data reception by filtering out messages coming from a *DataWriter* belonging to the same *DomainParticipant* on the *DataReader* receiving the data (by looking at the `GuidPrefix_t`), this entails for a data sample to go all the way to the *DataReaderListener* just to be discarded by an overcomplicated application business logic. For this reason, Fast DDS offers the possibility of instructing the *DomainParticipant* to avoid the matching of local endpoints through the following property:

PropertyPolicyQos name	PropertyPolicyQos value	Default value
"fastdds.ignore_local_endpoints"	"true"/"false"	"false"

C++

```
DomainParticipantQos participant_qos;

// Avoid local matching of this participant's endpoints
participant_qos.properties().properties().emplace_back(
    "fastdds.ignore_local_endpoints",
    "true");
```

XML

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

(continues on next page)

(continued from previous page)

```
<rtps>
  <propertiesPolicy>
    <properties>
      <!-- Avoid local matching of this participant's endpoints -->
      <property>
        <name>fastdds.ignore_local_endpoints</name>
        <value>true</value>
      </property>
    </properties>
  </propertiesPolicy>
</rtps>
</participant>
</profiles>
</dds>
```

Note: An invalid value of `fastdds.ignore_local_endpoints` results in the default behaviour.

7.28 Statistics Module

The *Fast DDS Statistics module* is an extension of Fast DDS that enables the recollection of data concerning the DDS communication. The collected data is published using DDS over dedicated topics using builtin DataWriters within the *Statistics module*. Consequently, by default, Fast DDS does not compile this module because it may entail affecting the application's performance. Nonetheless, the Statistics module and the Monitor Service can be activated using the `-DFASTDDS_STATISTICS=ON` at CMake configuration step. For more information about *Fast DDS* compilation, see *Linux installation from sources* and *Windows installation from sources*.

Besides enabling the *Statistics Module* compilation, the user must enable those DataWriters that are publishing data on the topics of interest for the user's application. Therefore, the standard *DDS Layer* has been extended. The following section explains this DDS extended API.

Note: Please refer to *Statistics QoS Troubleshooting* for any problems related to the statistics module.

7.28.1 Statistics Module DDS Layer

This section explains the extended DDS API provided for the *Statistics Module*. First, the Statistics Topic List is presented together with the corresponding collected data. Next, the methods to enable/disable the corresponding DataWriters are explained. Then, the recommended QoS for enabling the DataWriters and creating the user's DataReaders that subscribe to the Statistics topics are described. Finally, a guide on how to overcome common problems when using the module are presented.

Statistics Topic names

Data collected by the *Fast DDS Statistics module* is published in one of the topics listed below. In order to simplify its use, the API provides aliases for the different statistics topics (see *Topic names*). The following table shows the correlation between the topic name and the corresponding alias.

Topic name	Alias
<code>_fastdds_statistics_history2history_latency</code>	<code>HISTORY_LATENCY_TOPIC</code>
<code>_fastdds_statistics_network_latency</code>	<code>NETWORK_LATENCY_TOPIC</code>
<code>_fastdds_statistics_publication_throughput</code>	<code>PUBLICATION_THROUGHPUT_TOPIC</code>
<code>_fastdds_statistics_subscription_throughput</code>	<code>SUBSCRIPTION_THROUGHPUT_TOPIC</code>
<code>_fastdds_statistics_rtps_sent</code>	<code>RTPS_SENT_TOPIC</code>
<code>_fastdds_statistics_rtps_lost</code>	<code>RTPS_LOST_TOPIC</code>
<code>_fastdds_statistics_heartbeat_count</code>	<code>HEARTBEAT_COUNT_TOPIC</code>
<code>_fastdds_statistics_acknack_count</code>	<code>ACKNACK_COUNT_TOPIC</code>
<code>_fastdds_statistics_nackfrag_count</code>	<code>NACKFRAG_COUNT_TOPIC</code>
<code>_fastdds_statistics_gap_count</code>	<code>GAP_COUNT_TOPIC</code>
<code>_fastdds_statistics_data_count</code>	<code>DATA_COUNT_TOPIC</code>
<code>_fastdds_statistics_resent_datas</code>	<code>RESENT_DATAS_TOPIC</code>
<code>_fastdds_statistics_sample_datas</code>	<code>SAMPLE_DATAS_TOPIC</code>
<code>_fastdds_statistics_pdp_packets</code>	<code>PDP_PACKETS_TOPIC</code>
<code>_fastdds_statistics_edp_packets</code>	<code>EDP_PACKETS_TOPIC</code>
<code>_fastdds_statistics_discovered_entity</code>	<code>DISCOVERY_TOPIC</code>
<code>_fastdds_statistics_physical_data</code>	<code>PHYSICAL_DATA_TOPIC</code>

HISTORY_LATENCY_TOPIC

The `_fastdds_statistics_history2history_latency` statistics topic collects data related with the latency between any two matched endpoints. This measurement provides information about the DDS overall latency independent of the user's application overhead. Specifically, the measured latency corresponds to the time spent between the instant when the sample is written to the DataWriter's history and the time when the sample is added to the DataReader's history and the notification is issued to the corresponding user's callback.

NETWORK_LATENCY_TOPIC

The `_fastdds_statistics_network_latency` statistics topic collects data related with the network latency (expressed in *ns*) between any two communicating locators. This measurement provides information about the transport layer latency. The measured latency corresponds to the time spent between the message being written in the RTPSMessageGroup until the message being received in the MessageReceiver.

Important: In the case of *TCP Transport*, the reported latency also includes the time spent on the datagram's CRC related operations. Mind that is possible to disable CRC operations when defining the *TCPTransportDescriptor*.

PUBLICATION_THROUGHPUT_TOPIC

The `_fastdds_statistics_publication_throughput` statistics topic collects the amount of data (expressed in *B/s*) that is being sent by each DataWriter. This measurement provides information about the publication's throughput.

SUBSCRIPTION_THROUGHPUT_TOPIC

The `_fastdds_statistics_subscription_throughput` statistics topic collects the amount of data (expressed in *B/s*) that is being received by each DataReader. This measurement provides information about the subscription's throughput.

RTPS_SENT_TOPIC

The `_fastdds_statistics_rtps_sent` statistics topic collects the number of RTPS packets and bytes that are being sent from each DDS entity to each locator.

RTPS_LOST_TOPIC

The `_fastdds_statistics_rtps_lost` statistics topic collects the number of RTPS packets and bytes that are being lost in the transport layer (dropped somewhere in between) in the communication between each DDS entity and locator.

HEARTBEAT_COUNT_TOPIC

The `_fastdds_statistics_heartbeat_count` statistics topic collects the number of heartbeat messages sent by each user's DataWriter. This topic does not apply to builtin (related to *Discovery*) and statistics DataWriters. Heartbeat messages are only sent if the *ReliabilityQosPolicy* is set to `RELIABLE_RELIABILITY_QOS`. These messages report the DataWriter's status.

ACKNACK_COUNT_TOPIC

The `_fastdds_statistics_acknack_count` statistics topic collects the number of acknack messages sent by each user's DataReader. This topic does not apply to builtin DataReaders (related to *Discovery*). Acknack messages are only sent if the *ReliabilityQosPolicy* is set to `RELIABLE_RELIABILITY_QOS`. These messages report the DataReader's status.

NACKFRAG_COUNT_TOPIC

The `_fastdds_statistics_nackfrag_count` statistics topic collects the number of nackfrag messages sent by each user's DataReader. This topic does not apply to builtin DataReaders (related to *Discovery*). Nackfrag messages are only sent if the *ReliabilityQosPolicy* is set to `RELIABLE_RELIABILITY_QOS`. These messages report the data fragments that have not been received yet by the DataReader.

GAP_COUNT_TOPIC

The `_fastdds_statistics_gap_count` statistics topic collects the number of gap messages sent by each user's DataWriter. This topic does not apply to builtin (related to [Discovery](#)) and statistics DataWriters. Gap messages are only sent if the *ReliabilityQosPolicy* is set to `RELIABLE_RELIABILITY_QOS`. These messages report that some specific samples are not relevant to a specific DataReader.

DATA_COUNT_TOPIC

The `_fastdds_statistics_data_count` statistics topic collects the total number of user's data messages and data fragments (in case that the message size is large enough to require RTPS fragmentation) that have been sent by each user's DataWriter. This topic does not apply to builtin (related to [Discovery](#)) and statistics DataWriters.

RESENT_DATAS_TOPIC

The `_fastdds_statistics_resent_data` statistics topic collects the total number of user's data messages and data fragments (in case that the message size is large enough to require RTPS fragmentation) that have been necessary to resend by each user's DataWriter. This topic does not apply to builtin (related to [Discovery](#)) and statistics DataWriters.

SAMPLE_DATAS_TOPIC

The `_fastdds_statistics_sample_datas` statistics topic collects the number of user's data messages (or data fragments in case that the message size is large enough to require RTPS fragmentation) that have been sent by the user's DataWriter to completely deliver a single sample. This topic does not apply to builtin (related to [Discovery](#)) and statistics DataWriters.

PDP_PACKETS_TOPIC

The `_fastdds_statistics_pdp_packets` statistics topic collects the number of PDP discovery traffic RTPS packets transmitted by each DDS DomainParticipant. PDP packets are the data messages exchanged during the PDP discovery phase (see [Discovery phases](#) for more information).

EDP_PACKETS_TOPIC

The `_fastdds_statistics_edp_packets` statistics topic collects the number of EDP discovery traffic RTPS packets transmitted by each DDS DomainParticipant. EDP packets are the data messages exchanged during the EDP discovery phase (see [Discovery phases](#) for more information).

DISCOVERY_TOPIC

The `_fastdds_statistics_discovered_entity` statistics topic reports the time when each local DomainParticipant discovers any remote DDS entity (with the exception of those DDS entities related with the *Fast DDS Statistics module*). This topic also carries the `PHYSICAL_DATA_TOPIC` information for the case of discovered DomainParticipant; if the discovered entity is either a DataReader or DataWriter, then the physical information is empty (see [Physical Data in Discovery Information](#) for more information about how to configure the physical data conveyed on the discovery messages).

PHYSICAL_DATA_TOPIC

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

Statistics Domain Participant

In order to start collecting data in one of the statistics topics (*Statistics Topic names*), the corresponding statistics DataWriter should be enabled. In fact, *Fast DDS Statistics module* can be enabled and disabled at runtime. For this purpose, *Fast DDS Statistics module* exposes an extended DDS DomainParticipant API:

- *Enable statistics DataWriters*
- *Disable statistics DataWriters*
- *Obtain pointer to the extended DomainParticipant class*
- *Example*
- *Automatically enabling statistics DataWriters*

Enable statistics DataWriters

Statistics DataWriters can be enabled in different ways. It can be done automatically (see *Automatically enabling statistics DataWriters*). Alternatively, Statistics DataWriters can be enabled at run time using one of two methods: `enable_statistics_datawriter()` or `enable_statistics_datawriter_with_profile()`.

`enable_statistics_datawriter()` method requires as parameters:

- Name of the statistics topic to be enabled (see *Statistics Topic names* for the statistics topic list).
- DataWriter QoS profile (see *Statistics DataWriter recommended QoS* for the recommended profile).

It is possible to define specific desired QoS through DataWriter profile on the `FASTDDS_DEFAULT_PROFILES_FILE` (see *XML profiles*). `enable_statistics_datawriter_with_profile()` method enables a DataWriter by searching a specific DataWriter XML profile. On those profiles, specific QoS can be set.

`enable_statistics_datawriter_with_profile()` method requires as parameters:

- Name of the XML profile to use to fill the QoS structure of the DataWriter.
- Name of the statistics topic name to be enabled. (see *Statistics Topic names* for the statistics topic list).

Disable statistics DataWriters

Statistics DataWriters are disabled using the method `disable_statistics_datawriter()`. This method requires as parameter:

- Name of the statistics topic to be disabled (see *Statistics Topic names* for the statistics topic list).

Obtain pointer to the extended DomainParticipant class

The `DomainParticipant` is created using the `create_participant()` provided by the `DomainParticipantFactory`. This method returns a pointer to the DDS standard `DomainParticipant` created. In order to obtain the pointer to the child `DomainParticipant` which extends the DDS API, the static method `narrow()` is provided.

Example

The following example shows how to use the Statistics module extended DDS API:

```
// Create a DomainParticipant
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Obtain pointer to child class
eprosima::fastdds::statistics::dds::DomainParticipant* statistics_participant =
    eprosima::fastdds::statistics::dds::DomainParticipant::narrow(participant);

// Enable statistics DataWriter
if (statistics_participant->enable_statistics_
    ↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC,
    eprosima::fastdds::statistics::dds::STATISTICS_DATAWRITER_QOS) != RETCODE_OK)
{
    // Error
    return;
}

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

// Disable statistics DataWriter
if (statistics_participant->disable_statistics_
    ↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC) !=
    RETCODE_OK)
{
    // Error
    return;
}

// Delete DomainParticipant
if (DomainParticipantFactory::get_instance()->delete_participant(participant) != RETCODE_
    ↪OK)
{
    // Error
    return;
}
```

(continues on next page)

(continued from previous page)

```
}
```

Automatically enabling statistics DataWriters

The statistics DataWriters can be directly enabled using the *DomainParticipantQos* `properties()` `fastdds.statistics`. The value of this property is a semicolon separated list containing the *statistics topic name aliases* of those DataWriters that the user wants to enable. The property can be set either programmatically or loading an XML file. If the property is set in both ways, the priority would depend on the API and the QoS profile provided:

- XML settings have priority if `create_participant_with_profile()` is called with a valid participant profile.
- XML settings also have priority if `create_participant()` is called using `PARTICIPANT_QOS_DEFAULT` and a participant profile exists in the XML file with the `is_default_profile` option set to `true` (*DomainParticipant XML attributes*).
- The property set programmatically is used only when `create_participant()` is called with the specific QoS.

Another way of enabling statistics DataWriters, compatible with the previous one, is setting the *FASTDDS_STATISTICS* environment variable. The statistics DataWriters that will be enabled when the `DomainParticipant` is enabled would be the union between those specified in the `properties()` `fastdds.statistics` and those included with the environment variable.

The following examples show how to use all the previous methods:

C++

```
DomainParticipantQos pqos;

// Activate Fast DDS Statistics module
pqos.properties().properties().emplace_back("fastdds.statistics",
    "HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC
    ↪");
```

XML

```
<participant profile_name="statistics_domainparticipant_conf_xml_profile">
  <rtps>
    <propertiesPolicy>
      <properties>
        <!-- Activate Fast DDS Statistics Module -->
        <property>
          <name>fastdds.statistics</name>
          <value>HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
    ↪PHYSICAL_DATA_TOPIC</value>
          </property>
        </properties>
      </propertiesPolicy>
    </rtps>
  </participant>
```

Environment Variable Linux

```
export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
    ↪PHYSICAL_DATA_TOPIC"
```

Environment Variable Windows

```
set FASTDDS_STATISTICS=HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;
    ↪PHYSICAL_DATA_TOPIC
```

Note: These are all the statistics topics:

```
HISTORY_LATENCY_TOPIC;NETWORK_LATENCY_TOPIC;PUBLICATION_THROUGHPUT_TOPIC;SUBSCRIPTION_
    ↪THROUGHPUT_TOPIC;RTPS_SENT_TOPIC;RTPS_LOST_TOPIC;HEARTBEAT_COUNT_TOPIC;ACKNACK_COUNT_
    ↪TOPIC;NACKFRAG_COUNT_TOPIC;GAP_COUNT_TOPIC;DATA_COUNT_TOPIC;RESENT_DATAS_TOPIC;SAMPLE_
    ↪DATAS_TOPIC;PDP_PACKETS_TOPIC;EDP_PACKETS_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC
```

Note: Be aware that automatically enabling the statistics DataWriters using all these methods implies using the recommended QoS profile STATISTICS_DATAWRITER_QOS. For more information, please refer to *Statistics DataWriter recommended QoS*. However, if an XML profile is defined, the QoS applied are those defined in the profile, and for those QoS that are not specified in that profile, the default library QoS are applied (see *DataWriterQos* for the standard eProsima's DataWriter QoS), and not the recommended QoS for the Statistics DataWriters.

For the creation of an automatically enabled Datawriter, the priority for setting its QoS is the following:

- First, if a specific profile exists for the statistics topic, that one is applied.
- If that is not the case but a generic profile for statistics DataWriters exists, that one is applied.
- If no profile is defined in XML file, the recommended statistics QoS are applied.

Note: The generic DataWriter profile defined in the *FASTDDS_DEFAULT_PROFILES_FILE* XML needs to be named as *GENERIC_STATISTICS_PROFILE*.

The specific DataWriter profile defined in the *FASTDDS_DEFAULT_PROFILES_FILE* XML needs to be named using the same statistic topic alias or name (see *Statistics Topic names* for the alias corresponding to each statistic topic) that has been used in the *DomainParticipantQos* properties() `fastdds.statistics` (see *Statistics Module Settings*) or the *FASTDDS_STATISTICS* environment variable, where the enabling of the corresponding statistics topic has been set.

Statistics recommended QoS

Although the statistics DataWriters can be enabled using any valid QoS profile, the recommended profile is presented below. Also, the DataReaders created by the user to receive the data being published by the statistics DataWriters can use any compatible QoS profile. However, a recommended DataReader QoS profile is also provided.

Statistics DataWriter recommended QoS

The following table shows the recommended `DataWriterQos` profile for enabling the statistics DataWriters. This profile enables the pull mode *operating mode* on the statistics DataWriters. This entails that the DataWriters will only send information upon the reception of acknack submessages sent by the monitoring DataReader. This QoS profile is always used when the statistics DataWriters are *auto-enabled*. The recommended profile can be accessed through the constant `STATISTICS_DATAWRITER_QOS`.

Qos Policy	Value
<code>ReliabilityQosPolicyKind</code>	<code>RELIABLE_RELIABILITY_QOS</code>
<code>DurabilityQosPolicyKind</code>	<code>TRANSIENT_LOCAL_DURABILITY_QOS</code>
<code>PublishModeQosPolicyKind</code>	<code>ASYNCHRONOUS_PUBLISH_MODE</code>
<code>flow_controller_name</code>	<code>FASTDDS_STATISTICS_FLOW_CONTROLLER_DEFAULT</code>
<code>HistoryQosPolicyKind</code>	<code>KEEP_LAST_HISTORY_QOS</code>
<code>history_depth</code>	10
<code>PropertyPolicyQos name = value</code>	<code>"fastdds.push_mode" = "false"</code>

Statistics DataReader recommended QoS

The following table shows the recommended `DataReaderQos` profile for creating the monitoring DataReaders. The recommended profile can be accessed through constant `STATISTICS_DATAREADER_QOS`.

Qos Policy	Value
ReliabilityQosPolicyKind	RELIABLE_RELIABILITY_QOS
DurabilityQosPolicyKind	TRANSIENT_LOCAL_DURABILITY_QOS
HistoryQosPolicyKind	KEEP_LAST_HISTORY_QOS
history_depth	100
MemoryManagementPolicy	PREALLOCATED_WITH_REALLOC_MEMORY_MODE

Troubleshooting

This section aims to give quick solutions to overcome the most common problems arising from the use of the statistics module.

Monitoring application is not receiving any statistic data

Sometimes, especially in the case of monitoring large applications with many DataWriters and DataReaders, it may happen that the application monitoring Fast DDS statistics does not receive any data. This is generally caused by the default configuration of the statistics DataWriters, which includes the `push_mode` set to `false` (i.e. `pull_mode`), the History Kind set to `KEEP_LAST`, and the History Depth set to `10`. With this configuration, the following may happen:

1. Fast DDS adds new samples to one of the statistics DataWriters.
2. The DataWriter notifies the DataReader of the availability of said samples.
3. The DataReader sends a request to the DataWriter to “pull” those samples.
4. Before the request arrives to the DataWriter, some new statistics samples are added to that same DataWriter, which causes the previous samples to be overwritten.
5. Once the DataReader request arrives to the DataWriter, since the requested samples have been overwritten, they are not available any more, so the DataWriter send a notification to the DataReader informing of the presence of the newer samples instead.
6. The loop starts again.

The easiest fix to overcome this situation is to simply increase the History Depth of the DataWriter to create Some buffer to answer to requests:

Generic profile

```
<?xml version="1.0" encoding="utf-8"?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <participant profile_name="statistics_domainparticipant_conf_xml_general_profile
    <rtps>
      <propertiesPolicy>
        <properties>
          <!-- Activate various Fast DDS Statistics Module DataWriters -->
          <property>
            <name>fastdds.statistics</name>
            <value>HISTORY_LATENCY_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_
TOPIC</value>
          </property>
        </properties>
```

(continues on next page)

(continued from previous page)

```

        </propertiesPolicy>
    </rtps>
</participant>

<!-- Generic profile for all the statistics DataWriter -->
<data_writer profile_name="GENERIC_STATISTICS_PROFILE">
    <!-- Configure History QoS as KEEP_LAST 20 -->
    <!-- History depth depends on the user application constraints (publication_
↪rate for instance) -->
    <topic>
        <historyQos>
            <kind>KEEP_LAST</kind>
            <depth>20</depth>
        </historyQos>
    </topic>
    <!-- Enable pull mode -->
    <propertiesPolicy>
        <properties>
            <property>
                <name>fastdds.push_mode</name>
                <value>false</value>
            </property>
        </properties>
    </propertiesPolicy>
    <!-- Set durability, reliability, and publication mode -->
    <qos>
        <durability>
            <kind>TRANSIENT_LOCAL</kind>
        </durability>

        <reliability>
            <kind>RELIABLE</kind>
        </reliability>

        <publishMode>
            <kind>ASYNCHRONOUS</kind>
        </publishMode>
    </qos>
</data_writer>
</profiles>
</dds>

```

Specific profile

```

<?xml version="1.0" encoding="utf-8"?>
<dds xmlns="http://www.eprosima.com">
    <profiles>
        <participant profile_name="statistics_domainparticipant_conf_xml_specific_profile
↪">
            <rtps>
                <propertiesPolicy>
                    <properties>

```

(continues on next page)

(continued from previous page)

```

        <!-- Activate various Fast DDS Statistics Module DataWriters -->
        <property>
            <name>fastdds.statistics</name>
            <value>HISTORY_LATENCY_TOPIC</value>
        </property>
    </properties>
</propertiesPolicy>
</rtsp>
</participant>

<!-- Generic profile for a specific statistics DataWriters -->
<data_writer profile_name="HISTORY_LATENCY_WRITER">
    <!-- Configure History QoS as KEEP_LAST 20 -->
    <!-- History depth depends on the user application constraints (publication_
↪rate for instance) -->
    <topic>
        <historyQos>
            <kind>KEEP_LAST</kind>
            <depth>20</depth>
        </historyQos>
    </topic>
    <!-- Enable pull mode -->
    <propertiesPolicy>
        <properties>
            <property>
                <name>fastdds.push_mode</name>
                <value>false</value>
            </property>
        </properties>
    </propertiesPolicy>
    <!-- Set durability, reliability, and publication mode -->
    <qos>
        <durability>
            <kind>TRANSIENT_LOCAL</kind>
        </durability>

        <reliability>
            <kind>RELIABLE</kind>
        </reliability>

        <publishMode>
            <kind>ASYNCHRONOUS</kind>
        </publishMode>
    </qos>
</data_writer>
</profiles>
</dds>

```

Note: Increasing the History Depth of the statistics DataWriters has an impact on memory usage, as sufficient space is pre-allocated for each of the DataWriter's histories to hold that number of samples per topic instance.

7.28.2 Monitor Service

The *Fast DDS* Monitor Service is a feature of Fast DDS that grants the user the ability to collect data about the entities existing within a particular *domain* (i.e. *DomainParticipants*, *DataReaders*, *DataWriters*) as well as the capability of detecting possible misconfigurations among them.

Introduction

The **Monitor Service** targets any application implementing the subscription side of the *Monitor Service Status Topic*, giving the possibility of retrieving the *Monitoring Information* of the local entities (incompatible QoS, deadlines missed, active connections,...).

Keywords

- **Proxy**: An entity that acts on behalf of another entity.
- **Proxy Data**: The way in which a Proxy can be described.
- **Monitoring Information**: The collection of different sources of information and statuses of an entity, including: the Proxy Data, incompatible QoS, connections, liveliness, deadlines missed, inconsistent topics and lost sample status.

Description

Enabling the service makes each *DomainParticipant* publish its local entities, each one with its related **Monitoring Information**.

The *Monitor Service* is disabled by default, as it may entail a performance cost. Further information on the *Monitor Service* topics and how to configure it is described in the following sections.

The *Monitor Service* is available in both the *DDS Layer* and *RTPS Layer*.

Note: If the service is activated within a *RTPS* context, not all the **Monitoring Information** may be published by the service.

Use Cases

The *Monitor Service* can be particularly useful in the following scenarios:

- Collecting the **Monitoring Information** of any local entity of a remote *DomainParticipant* in order to extend the default discovery information (see *Discovery*) about it.
- Troubleshooting issues regarding discovery or entity-matching, leveraging the information of the current locators in use, for example.
- Recreating an entity graph of a certain domain given that all participants are able to discover each other.

Monitor Service Topics

The following table depicts the properties of the topics within the *Monitor Service*:

Topic name	Topic Alias	TopicDataType
fastdds_monitor_service_status	MONITOR_SERVICE_TOPIC	<i>Monitor Service Status Data</i>

Monitor Service Status Topic

The Monitor Service Status Topic carries information about the *monitoring information* of the local entities of a particular *DomainParticipant*. The *monitoring information* can be divided into different statuses identified by a *StatusKind*. The possible values are described in the following table:

Value	Name	Description
0	ProxyInfo	Collection of Parameters describing the Proxy Data of that entity
1	ConnectionList	List of connections that this entity is using with its matched remote entities.
2	IncompatibleQoSInfo	Status of the Incompatible QoS of that entity.
3	InconsistentTopicInfo	Status of Inconsistent topics that the topic of that entity has.
4	LivelinessLostInfo	Tracks the status of the number of times a writer lost liveliness.
5	LivelinessChangedInfo	Tracks the status of the number of times the liveliness changed in a reader.
6	DeadlineMissedInfo	The Status of the number of deadlines missed of a sample for that entity.
7	SampleLostInfo	Tracks the status of the number of times this entity lost samples.

Note: If the service is enabled in a *RTPS layer* context, *not all the statuses will be published*, only the ProxyInfo and ConnectionList.

The Monitor Service Status Topic publishes new data when new updates are received from any of the *DomainParticipant*'s local entities (on-event driven) with a minimum waiting time between publications. In addition, it is in charge of notifying about any disposal or liveliness lost.

Monitor Service Status Data

The MonitorServiceStatusData data structure comprises the following fields:

- *local_entity*: Guid_t of the local entity.
- *status_kind*: *StatusKind* enumeration identifying the status.
- *value*: The value of the status.

```
MonitorServiceStatusData
@Key GUID local_entity
@Key StatusKind status_kind
Data value
```

Note: The *local_entity* and *status_kind* are keyed fields, hence making use of instances (see *Topics, keys and instances*). In this case, the pair <*local_entity*, *status_kind*> identifies a unique instance.

Each of the StatusKind enumeration values maps to a corresponding Data value. The actual field names for the different values are described below:

- *entity_proxy*: Collection of the serialized Quality of Service Parameters in the form of a *ParameterList*.
- *connection_list*: Defines how is this entity communicating with its matched entities. Each of the elements is of Connection type (depicted below).

```

Connection
uint32_t mode //INTRAPROCESS, DATASHARING, TRANSPORT
LocatorList announced_locators
LocatorList used_locators

```

- *incompatible_qos_status*: Status of the Incompatible QoS of that entity.
 - *DataWriter* Incompatible QoS Offered.
 - *DataReader* Incompatible QoS Requested.
- *inconsistent_topic_status*: Status of Inconsistent topics of the topic of that entity. Asked to the topic of the requested entity.
- *liveliness_lost_status*: Tracks the status of the number of times that liveliness was lost by a *DataWriter*.
- *liveliness_changed_status*: Tracks the status of the number of times that liveliness was lost by a *DataReader*.
- *deadline_status*: The Status of the number of deadlines missed that were registered in that entity.
- *sample_lost_status*: The number of samples that entity lost.

The following table depicts the relation between each of the StatusKind values and the Data field:

StatusKind Value	StatusKind Name	Data field Name	IDL Data field Type
0	ProxyInfo	entity_proxy	sequence<octet>
1	ConnectionList	connection_list	sequence<Connection>
2	IncompatibleQoSInfo	incompatible_qos_status	IncompatibleQoSStatus
3	InconsistentTopicInfo	inconsistent_topic_status	InconsistentTopicStatus
4	LivelinessLostInfo	liveliness_lost_status	LivelinessLostStatus
5	LivelinessLostInfo	liveliness_changed_status	LivelinessChangedStatus
6	DeadlineMissedInfo	deadline_missed_status	DeadlineMissedStatus
7	SampleLostInfo	sample_lost_status	SampleLostStatus

Monitor Service Configuration

The *Monitor Service* can be activated using the `-DFASTDDS_STATISTICS=ON` at CMake configuration step (for further information regarding the *Fast DDS* compilation, see [Linux installation from sources](#) and [Windows installation from sources](#)). Once the *Monitor Service* feature is activated, it can be programmatically enabled in both *DDS Layer* and *RTPS Layer* through the `enable_monitor_service()` and `disable_monitor_service()` calls. In addition, leveraging the *PropertyPolicyQos* there is new Property defined for the purpose: `fastdds.enable_monitor_service`.

The following table depicts the different ways in which the *Monitor Service* can be enabled or disabled:

C++ API

```

DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
↪ participant(0,
              PARTICIPANT_QOS_DEFAULT);

```

(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 Fast DDS Monitor Service through API
statistics_participant->enable_monitor_service();

// Disable Fast DDS Monitor Service through API
statistics_participant->disable_monitor_service();
```

C++ Property

```
DomainParticipantQos pqos;

// Enable Fast DDS Monitor Service through properties
pqos.properties().properties().emplace_back("fastdds.enable_monitor_service",
    "true");

// Enable Fast DDS Monitor Service through statistics properties (other way)
pqos.properties().properties().emplace_back("fastdds.statistics",
    "MONITOR_SERVICE_TOPIC");

DomainParticipant* participant_with_mon_srv = DomainParticipantFactory::get_instance()->
    ↪create_participant(0,
        pqos);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
<participant profile_name="monitor_service_participant_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Enable Monitor Service -->
                <property>
                    <name>fastdds.enable_monitor_service</name>
                    <value>true</value>
                </property>
                <!-- Include Monitor Service in the Fast DDS Statistics (other way to
    ↪enable it) -->
                <property>
                    <name>fastdds.statistics</name>
                    <value>MONITOR_SERVICE_TOPIC</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
</profiles>
```

Env. Variable Linux

```
export FASTDDS_STATISTICS="MONITOR_SERVICE_TOPIC"
```

Env. Variable Windows

```
set FASTDDS_STATISTICS=MONITOR_SERVICE_TOPIC
```

Endpoints QoS

For any consumer application of the *Monitor Service*, the following endpoint QoS of the *Monitor Service Status Topic* DataWriter should be taken into consideration:

Endpoint	QoS	Value
MONITOR_STATUS_WRITER	ReliabilityQosPolicyKind	RELIABLE_RELIABILITY_QOS
	HistoryQosPolicyKind	KEEP_LAST_HISTORY_QOS 1
	DurabilityQosPolicyKind	TRANSIENT_LOCAL_DURABILITY_QOS

7.29 XTypes

eProsima Fast DDS supports the OMG [Extensible and Dynamic Topic Types for DDS](#) specification (also known as XTypes). This specification defines the following concepts:

- DDS supported type system, including the concept of extensible types that might evolve in time.
- Type representation, including *IDL* and *TypeObject* representations.
- Data representation over the wire.
- Language binding, defining both a plain and a dynamic language binding. *eProsima Fast DDS-Gen* generates the plain language binding given an IDL type representation. [Dynamic Language Binding](#) section explains the required API to define and set/read the data types.
- DDS builtin mechanism to automatically discover remote data types. More information in [Remote Data Types Discovery](#) section.

7.29.1 Remote Data Types Discovery

[DDS-XTypes specification](#) defines an internal mechanism to discover the remote data types at runtime and match depending on the extensible types compatibility rules configured using the [TypeConsistencyEnforcementQosPolicy](#).

Note: *eProsima Fast DDS* does not support XTypes compatibility check yet.

The remote data type discovery mechanism is based on the exchange of the data type information optimized in order to reduce the required bandwidth. On the one hand, `TypeInformation` structure defined in the IDL below (extracted from Annex B of the [DDS-XTypes specification](#)), is used to communicate the Topic Data Type and its dependencies.

```
@extensibility(APPENDABLE) @nested
struct TypeIdentifierWithSize
{
```

(continues on next page)

(continued from previous page)

```

    TypeIdentifier      type_id;
    unsigned long      typeobject_serialized_size;
};

@extensibility(APPENDABLE) @nested
struct TypeIdentifierWithDependencies
{
    TypeIdentifierWithSize      typeid_with_size;
    long                        dependent_type_id_count;
    sequence<TypeIdentifierWithSize> dependent_typeids;
};

@extensibility(MUTABLE) @nested
struct TypeInformation
{
    @id(0x1001) TypeIdentifierWithDependencies minimal;
    @id(0x1002) TypeIdentifierWithDependencies complete;
};

```

TypeInformation includes the TypeIdentifier, the data type information hashed which identifies almost univocally the data type. The data type information is contained in the TypeObject union:

```

@extensibility(APPENDABLE) @nested
union TypeObject switch(octet)
{
    case EK_COMPLETE:
        CompleteTypeObject  complete;
    case EK_MINIMAL:
        MinimalTypeObject    minimal;
};

```

The CompleteTypeObject includes the data type full description. On the other hand, MinimalTypeObject only includes the minimum required information in order to check type compatibility.

Important: Current TypeObject representation implementation does not support forward declarations or recursive data types defined using the @external annotation. Please, remember to disable TypeObject generation code using -no-typeobjectsupport option when generating the code using Fast DDS-Gen.

Prerequisites

The remote data type discovery feature only works if some requisites are met:

1. The local data types must be registered into the ITypeObjectRegistry. The types are automatically registered when calling register_type() / register_type() if the code required for registration has been generated using *eProsima Fast DDS-Gen*. Fast DDS-Gen generates the required files (<IDLFileName>TypeObjectSupport.cxx/.hpp) by default.

Note: -no-typeobjectsupport option disables the generation of these files and effectively disables the discovery of remote types.

Note: Currently there is no support to register the `TypeObject` when the data type is defined using the *Dynamic Language Binding*. Consequently, local types defined using the dynamic language binding API, are not going to be discovered by remote `DomainParticipants`.

2. `TypeInformation` should be received with the `DomainParticipant`'s endpoint discovery information. A `DomainParticipant` that does not inform about its `TypeInformation` would not trigger the remote data type discovery mechanism.

If the prerequisites are not met, endpoint matching relies on type name and topic name in order to match the discovered endpoints.

Remote types discovery example

Please, refer to *Remote type discovery and endpoint matching* for more information about how to leverage this feature.

7.29.2 Dynamic Language Binding

The Dynamic Language Binding API allows to define data types at runtime instead of having the types predefined as it is required by the Plain Language Binding. This API includes both the type definition and, the getters and setters required to use the defined types. Type definition can also be done using a XML configuration file as explained in *Dynamic Types profiles* section.

This section presents first the Dynamic Language Binding API, and then the supported types and specific examples defining and using those types.

Dynamic Language Binding Interfaces

This section briefly presents the Dynamic Language Binding API. For more information, please refer both to the *DDS-XTypes specification* and the *API reference*.

TypeDescriptor

`TypeDescriptor` is in charge of describing the state of a type. Objects of this interface have value semantics allowing the `TypeDescriptor` data to be deeply copied and compared.

AnnotationDescriptor

`AnnotationDescriptor` is in charge of describing the user-defined applied annotation to a specific element. Objects of this interface have value semantics allowing the `AnnotationDescriptor` data to be deeply copied and compared.

MemberDescriptor

`MemberDescriptor` is in charge of describing the state of a specific member of a type. Objects of this interface have value semantics allowing the `MemberDescriptor` data to be deeply copied and compared.

VerbatimTextDescriptor

`VerbatimTextDescriptor` is in charge of describing the `@verbatim` builtin annotation application. Objects of this interface have value semantics allowing the `VerbatimTextDescriptor` data to be deeply copied and compared.

DynamicTypeBuilderFactory

The `DynamicTypeBuilderFactory` serves as a singleton which instance is responsible for both creating and deleting `DynamicTypeBuilder` objects. This class provides the generic `DynamicTypeBuilderFactory::create_type` API, and also specific APIs to define other basic types such as strings, sequences, etc. More information can be found in *Supported Types* section.

DynamicType

`DynamicType` objects represents a specific type definition. Once the `DynamicType` has been built, it cannot be modified. Objects of this interface have reference semantics, so the API receives a nil-reference which is then returned pointing to the correct `DynamicType` address.

DynamicTypeMember

`DynamicTypeMember` represents a data member of a `DynamicType`. Objects of this interface have reference semantics, so the API receives a nil-reference which is then returned pointing to the correct `DynamicTypeMember` address.

DynamicTypeBuilder

`DynamicTypeBuilder` interface allows the instantiation of concrete `DynamicType` objects and serves as a transitional state for configuring the `DynamicType` before its creation. Upon definition, `DynamicTypeBuilderFactory` leverages the information contained in the builder to create the `DynamicType`. `DynamicTypeBuilder::build` allows for creating the fully constructed `DynamicType`. Builders remain reusable after `DynamicType` creation, ensuring changes to the builder do not affect previously created types.

DynamicDataFactory

`DynamicDataFactory` serves as a singleton which instance is responsible for both creating and deleting `DynamicData` objects from a given `DynamicType` instance.

DynamicData

`DynamicData` represents a data instance of a `DynamicType`, providing functionalities to access and modify data values. Each `DynamicData` object corresponds to an object of the type represented by its `DynamicType`. Offering reflective getters and setters, `DynamicData` enables manipulation of individual data samples.

Supported Types

This section describes the supported Type System including examples of how to instantiate those specific types using the Dynamic Language Binding API and the XML configuration file. The C++ examples also include instantiating the corresponding `DynamicData` sample, and setting and reading a value.

Primitive Types

Primitive types are self-describing and can be created without configuration parameters. The `DynamicTypeBuilderFactory` interface exposes the method `DynamicTypeBuilderFactory::get_primitive_type` to allow users to directly get the corresponding primitive `DynamicType`. The `DynamicData` class provides specific getters and setters for each primitive data type.

The following table shows the supported primitive types and their corresponding `TypeKind`. The `TypeKind` is used to query the `DynamicTypeBuilderFactory` for the specific primitive `DynamicType`.

C++ Type	TypeKind
<code>bool</code>	<code>TK_BOOLEAN</code>
<code>char</code>	<code>TK_CHAR8</code>
<code>wchar_t</code>	<code>TK_CHAR16</code>
<code>uint8_t</code>	<code>TK_BYTE</code> / <code>TK_UINT8</code>
<code>int8_t</code>	<code>TK_INT8</code>
<code>int16_t</code>	<code>TK_INT16</code>
<code>uint16_t</code>	<code>TK_UINT16</code>
<code>int32_t</code>	<code>TK_INT32</code>
<code>uint32_t</code>	<code>TK_UINT32</code>
<code>int64_t</code>	<code>TK_INT64</code>
<code>uint64_t</code>	<code>TK_UINT64</code>
<code>float</code>	<code>TK_FLOAT32</code>
<code>double</code>	<code>TK_FLOAT64</code>
<code>long double</code>	<code>TK_FLOAT128</code>

The example below shows how to create an structure with primitive members.

IDL

```
struct PrimitivesStruct
{
    boolean my_bool;
    octet my_octet;
    char my_char;
    wchar my_wchar;
    long my_long;
    unsigned long my_ulong;
    int8 my_int8;
```

(continues on next page)

(continued from previous page)

```

uint8 my_uint8;
short my_short;
unsigned short my_ushort;
long long my_longlong;
unsigned long long my_ulonglong;
float my_float;
double my_double;
long double my_longdouble;
};

```

XML

```

<struct name="PrimitivesStruct">
  <member name="my_bool" type="boolean"/>
  <member name="my_octet" type="byte"/>
  <member name="my_char" type="char8"/>
  <member name="my_wchar" type="char16"/>
  <member name="my_long" type="int32"/>
  <member name="my_ulong" type="uint32"/>
  <member name="my_int8" type="int8"/>
  <member name="my_uint8" type="uint8"/>
  <member name="my_short" type="int16"/>
  <member name="my_ushort" type="uint16"/>
  <member name="my_longlong" type="int64"/>
  <member name="my_ulonglong" type="uint64"/>
  <member name="my_float" type="float32"/>
  <member name="my_double" type="float64"/>
  <member name="my_longdouble" type="float128"/>
</struct>

```

C++

```

// Define a struct type with various primitive members
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;

type_descriptor->kind(TK_STRUCTURE);
type_descriptor->name("PrimitivesStruct");
DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↪instance()->
                                create_type(type_
↪descriptor)}};

// Define and add primitive members to the struct
MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};
member_descriptor->name("my_bool");
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪type(TK_BOOLEAN));
struct_builder->add_member(member_descriptor);
member_descriptor = traits<MemberDescriptor>::make_shared();
member_descriptor->name("my_octet");
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_

```

(continues on next page)

(continued from previous page)

```

↪ type(TK_BYTE));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_char");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_CHAR8));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_wchar");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_CHAR16));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_long");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_INT32));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_ulong");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_UINT32));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_int8");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_INT8));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_uint8");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_UINT8));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_short");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_INT16));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_ushort");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_UINT16));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_longlong");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_INT64));
    struct_builder->add_member(member_descriptor);
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_ulonglong");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪ type(TK_UINT64));
    struct_builder->add_member(member_descriptor);

```

(continues on next page)

(continued from previous page)

```

        member_descriptor = traits<MemberDescriptor>::make_shared();
        member_descriptor->name("my_float");
        member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪type(TK_FLOAT32));
        struct_builder->add_member(member_descriptor);
        member_descriptor = traits<MemberDescriptor>::make_shared();
        member_descriptor->name("my_double");
        member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪type(TK_FLOAT64));
        struct_builder->add_member(member_descriptor);
        member_descriptor = traits<MemberDescriptor>::make_shared();
        member_descriptor->name("my_longdouble");
        member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_primitive_
↪type(TK_FLOAT128));
        struct_builder->add_member(member_descriptor);

        // Build the struct type
        DynamicType::_ref_type struct_type {struct_builder->build()};
        // Create dynamic data based on the struct type
        DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

        // Set and retrieve values for a member of type int32_t
        int32_t in_value {2};
        int32_t out_value {0};
        data->set_int32_value(data->get_member_id_by_name("my_long"), in_value);
        data->get_int32_value(out_value, data->get_member_id_by_name("my_long"));

```

For a detailed explanation about the XML definition of this type, please refer to *XML Primitive Types*.

Type promotions

The Dynamic Language Binding also supports type promotion, enabling implicit promotion of types during both `get()` and `set()` operations. This means that a smaller type can be implicitly promoted to a larger type, but not the other way around.

The following promotions are supported:

TypeKind	Allowed promotions
TK_INT8	TK_INT16, TK_INT32, TK_INT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128
TK_INT16	TK_INT32, TK_INT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128
TK_INT32	TK_INT64, TK_FLOAT64, TK_FLOAT128
TK_INT64	TK_FLOAT128
TK_UINT8	TK_INT16, TK_INT32, TK_INT64, TK_UINT16, TK_UINT32, TK_UINT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128
TK_UINT16	TK_INT32, TK_INT64, TK_UINT32, TK_UINT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128
TK_UINT32	TK_INT64, TK_UINT64, TK_FLOAT64, TK_FLOAT128
TK_UINT64	TK_FLOAT128
TK_FLOAT32	TK_FLOAT64, TK_FLOAT128
TK_FLOAT64	TK_FLOAT128
TK_FLOAT128	(none)
TK_CHAR8	TK_CHAR16, TK_INT16, TK_INT32, TK_INT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128
TK_CHAR16	TK_INT32, TK_INT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128
TK_BYTE	(any)
TK_BOOLEAN	TK_INT8, TK_INT16, TK_INT32, TK_INT64, TK_UINT8, TK_UINT16, TK_UINT32, TK_UINT64, TK_FLOAT32, TK_FLOAT64, TK_FLOAT128

String Types

String types are one-dimensional collections of characters (TK_CHAR8 or TK_CHAR16. The latest are also known as wide strings or wstring). The TypeKinds used to identify string types are TK_STRING8 and TK_STRING16. The string may be bounded, setting a maximum length, or unbounded. This is configured using TypeDescriptor bound property.

DynamicTypeBuilderFactory exposes the functions DynamicTypeBuilderFactory::create_string_type and DynamicTypeBuilderFactory::create_wstring_type that eases string creation providing the corresponding maximum length parameter (LENGTH_UNLIMITED is used for unbounded strings).

DynamicData class provides also with specific getters and setters: DynamicData::get_string_value, DynamicData::get_wstring_value, DynamicData::set_string_value, and DynamicData::set_wstring_value.

IDL

```
struct StringsStruct
{
    string my_string;
    wstring my_wstring;
    string<41925> my_bounded_string;
    wstring<20925> my_bounded_wstring;
};
```

XML

```
<struct name="StringsStruct">
  <member name="my_string" type="string"/>
  <member name="my_wstring" type="wstring"/>
  <member name="my_bounded_string" type="string" stringMaxLength="41925"/>
  <member name="my_bounded_wstring" type="wstring" stringMaxLength="20925"/>
</struct>
```

C++

```

// Define a struct type to contain the strings
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;

type_descriptor->kind(TK_STRUCTURE);
type_descriptor->name("StringsStruct");
DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↪instance()->
                                create_type(type_
↪descriptor)}};

// Define and add string members to the struct
MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};

member_descriptor->name("my_string");
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
                        create_string_type(static_cast<uint32_t>(LENGTH_UNLIMITED))->
↪build());

/* Alternative
   type_descriptor = traits<TypeDescriptor>::make_shared();
   type_descriptor->kind(TK_STRING8);
   type_descriptor->element_type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_CHAR8));
   type_descriptor->bound().push_back(static_cast<uint32_t>(LENGTH_UNLIMITED));
   member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_
↪type(type_descriptor)->build());
*/

struct_builder->add_member(member_descriptor);
member_descriptor->name("my_wstring");
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
                        create_wstring_type(static_cast<uint32_t>(LENGTH_UNLIMITED))->
↪build());

/* Alternative
   type_descriptor = traits<TypeDescriptor>::make_shared();
   type_descriptor->kind(TK_STRING16);
   type_descriptor->element_type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_CHAR16));
   type_descriptor->bound().push_back(static_cast<uint32_t>(LENGTH_UNLIMITED));
   member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_
↪type(type_descriptor)->build());
*/

struct_builder->add_member(member_descriptor);
member_descriptor->name("my_bounded_string");
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
                        create_string_type(41925)->build());

/* Alternative

```

(continues on next page)

(continued from previous page)

```

        type_descriptor = traits<TypeDescriptor>::make_shared();
        type_descriptor->kind(TK_STRING8);
        type_descriptor->element_type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_CHAR8));
        type_descriptor->bound().push_back(41925);
        member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_
↪type(type_descriptor)->build());
        */

    struct_builder->add_member(member_descriptor);
    member_descriptor->name("my_bounded_wstring");
    member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
        create_wstring_type(20925)->build());

    /* Alternative
        type_descriptor = traits<TypeDescriptor>::make_shared();
        type_descriptor->kind(TK_STRING16);
        type_descriptor->element_type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_CHAR16));
        type_descriptor->bound().push_back(20925);
        member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_
↪type(type_descriptor)->build());
        */

    struct_builder->add_member(member_descriptor);

    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

    // Set and retrieve values for a string member
    std::string in_value {"helloworld"};
    std::string out_value;
    data->set_string_value(data->get_member_id_by_name("my_string"), in_value);
    data->get_string_value(out_value, data->get_member_id_by_name("my_string"));

```

For a detailed explanation about the XML definition of this type, please refer to *XML String Types*.

Enumeration Types

An enumeration contains a set of supported values (enumeration literals) and a selected value among those supported. The `TypeKind` used to identify enumeration types is `TK_ENUM`.

The enumeration literals must be configured using the `DynamicTypeBuilder` by calling the `DynamicTypeBuilder::add_member` function for the respective supported values. The `MemberDescriptor` passed to the previous function must determine the enumeration literal name by using `name` property. The underlying primitive type related to the enumeration is configured using `MemberDescriptor` `type` property. This primitive type is determined when adding the first enumeration literal. For the enumeration type to be consistent, the remaining enumeration literals must be of the same primitive type. Additionally, the enumeration literal value might be set using `MemberDescriptor` `default_value` property. The behavior is the same as setting the `@value` *builtin annotation*.

Note: Currently, Fast DDS-Gen does not support @value builtin annotation.

As the enumeration type is basically a primitive type which might take only some specific values defined with the enumeration literals, the corresponding DynamicData getters and setters are the ones corresponding to the underlying primitive type (and any other method promotable to that specific primitive type).

IDL

```
enum MyEnum
{
    A,
    B,
    C
};

struct EnumStruct
{
    MyEnum my_enum;
};
```

XML

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

<struct name="EnumStruct">
  <member name="my_enum" type="nonBasic" nonBasicTypeName="MyEnum"/>
</struct>
```

C++

```
enum MyEnum : uint32_t
{
    A,
    B,
    C
};

// Define a struct type to contain an enum
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;
type_descriptor->kind(TK_STRUCTURE);
type_descriptor->name("EnumStruct");
DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↪instance()->
                                create_type(type_
↪descriptor)}};
// Define the enum type
TypeDescriptor::_ref_type enum_type_descriptor {traits<TypeDescriptor>::make_
↪shared()};
```

(continues on next page)

(continued from previous page)

```

enum_type_descriptor->kind(TK_ENUM);
enum_type_descriptor->name("MyEnum");
DynamicTypeBuilder::_ref_type enum_builder {DynamicTypeBuilderFactory::get_
↪instance()->
                                create_type(enum_type_
↪descriptor)}};
    // Add enum literals to the enum type
    MemberDescriptor::_ref_type enum_member_descriptor {traits<MemberDescriptor>
↪::make_shared()};
    enum_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_UINT32));
    enum_member_descriptor->name("A");
    enum_builder->add_member(enum_member_descriptor);
    enum_member_descriptor = traits<MemberDescriptor>::make_shared();
    enum_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_UINT32));
    enum_member_descriptor->name("B");
    enum_builder->add_member(enum_member_descriptor);
    enum_member_descriptor = traits<MemberDescriptor>::make_shared();
    enum_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_UINT32));
    enum_member_descriptor->name("C");
    enum_builder->add_member(enum_member_descriptor);
    // Build the enum type
    DynamicType::_ref_type enum_type = enum_builder->build();

    // Add an enum member to the struct
    MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};
    member_descriptor->name("my_enum");
    member_descriptor->type(enum_type);
    struct_builder->add_member(member_descriptor);
    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

    // Set and retrieve values for an enum member
    MyEnum in_value {MyEnum::C};

    /* Alternative
       uint32_t in_value {2}; // Selecting MyEnum::C
    */

    uint32_t out_value {0};
    data->set_uint32_value(data->get_member_id_by_name("my_enum"), in_value);
    data->get_uint32_value(out_value, data->get_member_id_by_name("my_enum"));

```

For a detailed explanation about the XML definition of this type, please refer to *XML Enumeration Types*.

Bitmask Types

Bitmask types are a collection of boolean flags (bitflags) that can be set individually. The `TypeKind` used to identify bitmask types is `TK_BITMASK`. The bitmasks bound, maximum number of bits, must be set using the `TypeDescriptor` bound property. The maximum bound allowed is 64 bits.

The bitflags must be configured using the `DynamicTypeBuilder` by calling the `DynamicTypeBuilder::add_member` function. Each bitflag is described using a `MemberDescriptor` defining the bitflag name using the `name` property. The underlying primitive type related to bitflags must be of kind `TK_BOOLEAN` and must be set in `MemberDescriptor` type property. The `MemberDescriptor` `id` property might be used to indicate the bitflag position within the bitmask. This behavior is the same as setting the `@position` *builtin annotation*. If the position is not specified, sequential order is followed.

The `DynamicTypeBuilderFactory` exposes the function `DynamicTypeBuilderFactory::create_bitmask_type` to facilitate the creation of bitmask types.

Bitmask types can be manipulated either using the `DynamicData::get_boolean_value/DynamicData::set_boolean_value` in order to set a specific bitflag, or by using the unsigned integer setter/getter corresponding to the bitmask bound. In this latest case, only bitflags are going to be set (bits not named are always unset).

IDL

```
@bit_bound(8)
bitmask MyBitMask
{
    @position(0) flag0,
    flag1,
    flag2,
    @position(5) flag5
};

struct BitmaskStruct
{
    MyBitMask my_bitmask;
};
```

XML

```
<bitmask name="MyBitMask" bit_bound="8">
  <bit_value name="flag0" position="0"/>
  <bit_value name="flag1"/>
  <bit_value name="flag2"/>
  <bit_value name="flag5" position="5"/>
</bitmask>

<struct name="BitmaskStruct">
  <member name="my_bitmask" type="nonBasic" nonBasicTypeName="MyBitMask"/>
</struct>
```

C++

```
// Define a struct type to contain a bitmask
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;

type_descriptor->kind(TK_STRUCTURE);
```

(continues on next page)

(continued from previous page)

```

        type_descriptor->name("BitmaskStruct");
        DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↳instance()->
                                create_type(type_
↳descriptor)}};

        // Define the bitmask type
        DynamicTypeBuilder::_ref_type bitmask_builder {DynamicTypeBuilderFactory::get_
↳instance()->create_bitmask_type(8)};

        /* Alternative
        TypeDescriptor::_ref_type bitmask_type_descriptor {traits<TypeDescriptor>
↳::make_shared()};
        bitmask_type_descriptor->kind(TK_BITMASK);
        bitmask_type_descriptor->name("MyBitMask");
        bitmask_type_descriptor->element_type(DynamicTypeBuilderFactory::get_
↳instance()->get_primitive_type(
                                TK_BOOLEAN));
        bitmask_type_descriptor->bound().push_back(8);
        DynamicTypeBuilder::_ref_type bitmask_builder
↳{DynamicTypeBuilderFactory::get_instance()->create_type(
                                bitmask_type_descriptor)}};
        */

        // Add bitfield members to the bitmask type
        MemberDescriptor::_ref_type bitfield_member_descriptor {traits<MemberDescriptor>
↳::make_shared()};
        bitfield_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↳primitive_type(TK_BOOLEAN));
        bitfield_member_descriptor->name("flag0");
        bitfield_member_descriptor->id(0);
        bitmask_builder->add_member(bitfield_member_descriptor);
        bitfield_member_descriptor = traits<MemberDescriptor>::make_shared();
        bitfield_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↳primitive_type(TK_BOOLEAN));
        bitfield_member_descriptor->name("flag1");
        bitmask_builder->add_member(bitfield_member_descriptor);
        bitfield_member_descriptor = traits<MemberDescriptor>::make_shared();
        bitfield_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↳primitive_type(TK_BOOLEAN));
        bitfield_member_descriptor->name("flag2");
        bitmask_builder->add_member(bitfield_member_descriptor);
        bitfield_member_descriptor = traits<MemberDescriptor>::make_shared();
        bitfield_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↳primitive_type(TK_BOOLEAN));
        bitfield_member_descriptor->name("flag5");
        bitfield_member_descriptor->id(5);
        bitmask_builder->add_member(bitfield_member_descriptor);
        // Build the bitmask type
        DynamicType::_ref_type bitmask_type = bitmask_builder->build();

        // Add a bitmask member to the struct

```

(continues on next page)

(continued from previous page)

```

    MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};
    member_descriptor->name("my_bitmask");
    member_descriptor->type(bitmask_type);
    struct_builder->add_member(member_descriptor);
    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

    // Set and retrieve values for bitmask member.
    uint8_t in_value {3}; // Setting both "flag0" and "flag1" simultaneously.
    uint8_t out_value {0};
    data->set_uint8_value(data->get_member_id_by_name("my_bitmask"), in_value);
    data->get_uint8_value(out_value, data->get_member_id_by_name("my_bitmask"));

    // Set and retrieve specific bitflag
    bool in_bitflag_value = true;
    bool out_bitflag_value = false;
    DynamicData::_ref_type bitmask_data = data->loan_value(data->get_member_id_by_
↪name("my_bitmask"));
    bitmask_data->set_boolean_value(bitmask_data->get_member_id_by_name("flag5"), in_
↪bitflag_value);
    bitmask_data->get_boolean_value(out_bitflag_value, bitmask_data->get_member_id_
↪by_name("flag5"));

```

For a detailed explanation about the XML definition of this type, please refer to [XML Bitmask Types](#).

Alias Types

Alias types provide an alternative name to an already existing type. The `TypeKind` used to identify aliases is `TK_ALIAS`. Besides defining the alias name, the underlying type must be set using `TypeDescriptor` `base_type` property. Alias recursion is supported by defining another alias type as the base type.

Once the `DynamicData` is created, information can be accessed as if working with the base type.

IDL

```

typedef MyEnum MyAliasedEnum;
typedef string<100> MyAliasedBoundedString;
typedef MyAliasedEnum MyRecursiveAlias;

struct AliasStruct
{
    MyAliasedEnum my_aliased_enum;
    MyAliasedBoundedString my_aliased_bounded_string;
    MyRecursiveAlias my_recursive_alias;
};

```

XML

```

<typedef name="MyAliasedEnum" type="nonBasic" nonBasicTypeName="MyEnum"/>
<!-- XSD does not allow to set bounds to aliased strings -->
<typedef name="MyAliasedBoundedString" type="string"/>
<typedef name="MyRecursiveAlias" type="nonBasic" nonBasicTypeName="MyAliasedEnum"/>

<struct name="AliasStruct">
  <member name="my_aliased_enum" type="nonBasic" nonBasicTypeName="MyAliasedEnum"/>
  <member name="my_aliased_bounded_string" type="nonBasic" nonBasicTypeName=
↪ "MyAliasedBoundedString"/>
  <member name="my_recursive_alias" type="nonBasic" nonBasicTypeName="MyRecursiveAlias
↪ "/>
</struct>

```

C++

```

// Define a struct type to contain the alias
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;

type_descriptor->kind(TK_STRUCTURE);
type_descriptor->name("AliasStruct");
DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↪ instance()->
                                create_type(type_
↪ descriptor)}};

// Define an alias type for the enum
TypeDescriptor::_ref_type aliasenum_type_descriptor {traits<TypeDescriptor>
↪ ::make_shared()};
aliasenum_type_descriptor->kind(TK_ALIAS);
aliasenum_type_descriptor->name("MyAliasedEnum");
aliasenum_type_descriptor->base_type(enum_type);
DynamicTypeBuilder::_ref_type aliasenum_builder {DynamicTypeBuilderFactory::get_
↪ instance()->
                                create_type(aliasenum_
↪ type_descriptor)}};

// Build the alias type
DynamicType::_ref_type aliasenum_type = aliasenum_builder->build();

// Define an alias type for a bounded string
TypeDescriptor::_ref_type alias_bounded_string_type_descriptor {traits
↪ <TypeDescriptor>::make_shared()};
alias_bounded_string_type_descriptor->kind(TK_ALIAS);
alias_bounded_string_type_descriptor->name("MyAliasedBoundedString");
alias_bounded_string_type_descriptor->base_type(DynamicTypeBuilderFactory::get_
↪ instance()->
                                create_string_type(100)->build());
DynamicTypeBuilder::_ref_type alias_bounded_string_builder
↪ {DynamicTypeBuilderFactory::get_instance()->
                                create_type(alias_
↪ bounded_string_type_descriptor)}};

// Build the alias type for the bounded string
DynamicType::_ref_type alias_bounded_string_type = alias_bounded_string_builder->
↪ build();

```

(continues on next page)

(continued from previous page)

```

    // Define a recursive alias
    TypeDescriptor::_ref_type recursive_alias_type_descriptor {traits<TypeDescriptor>
↪::make_shared()};
    recursive_alias_type_descriptor->kind(TK_ALIAS);
    recursive_alias_type_descriptor->name("MyRecursiveAlias");
    recursive_alias_type_descriptor->base_type(aliasenum_type);
    DynamicTypeBuilder::_ref_type recursive_alias_builder
↪{DynamicTypeBuilderFactory::get_instance()->
                                   create_type(recursive_
↪alias_type_descriptor)};
    // Build the recursive alias type
    DynamicType::_ref_type recursive_alias_type = recursive_alias_builder->build();

    // Add alias enum member to the structure
    MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};
    member_descriptor->name("my_aliased_enum");
    member_descriptor->type(aliasenum_type);
    struct_builder->add_member(member_descriptor);
    // Add alias bounded string member to the structure
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_aliased_bounded_string");
    member_descriptor->type(alias_bounded_string_type);
    struct_builder->add_member(member_descriptor);
    // Add recursive alias member to the structure
    member_descriptor = traits<MemberDescriptor>::make_shared();
    member_descriptor->name("my_recursive_alias");
    member_descriptor->type(recursive_alias_type);
    struct_builder->add_member(member_descriptor);

    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

    // Set and retrieve values for the alias enum member
    MyEnum in_value {MyEnum::C};
    int32_t out_value {0};
    data->set_int32_value(data->get_member_id_by_name("my_alias_enum"), in_value);
    data->get_int32_value(out_value, data->get_member_id_by_name("my_alias_enum"));

```

For a detailed explanation about the XML definition of this type, please refer to *XML Alias Types*.

Sequence types

Sequence types are one-dimensional collections of any type. The `TypeKind` used to identify sequences is `TK_SEQUENCE`. `TypeDescriptor` `element_type` property must be set with the collection's type. Additionally, `bound` property must also be configured with the sequence's maximum length, or `LENGTH_UNLIMITED` in case of unbounded sequences.

`DynamicTypeBuilderFactory` exposes the function `DynamicTypeBuilderFactory::create_sequence_type` to facilitate the creation of this type. This API requires the type stored in the collection and the collection's bound, using `LENGTH_UNLIMITED` in case of unbounded sequences.

`DynamicData` class provides specific `get_values()` and `set_values()` functions for each primitive type, allowing users to easily work with sequences of primitive types. Primitive type promotion is also applicable for these methods. For sequences of more complex types, please refer to *Managing Complex Types Data*.

If a specific range of values within the sequence are to be modified, passing the starting index to `get_values()` / `set_values()` would only manage data from that element forward until the length of the given input. Specific collection's element can be also be modified using the `get_value()` / `set_value()` passing the index of the element to be modified.

IDL

```
struct SequenceStruct
{
    sequence<MyBitMask> bitmask_sequence;
    sequence<short, 5> short_sequence;
};
```

XML

```
<struct name="SequenceStruct">
  <member name="bitmask_sequence" type="nonBasic" nonBasicTypeName="MyBitMask"
sequenceMaxLength="-1"/>
  <member name="short_sequence" sequenceMaxLength="5" type="int16"/>
</struct>
```

C++

```
// Define a struct type to contain the sequence
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;

type_descriptor->kind(TK_STRUCTURE);
type_descriptor->name("SequenceStruct");
DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↪instance()->
create_type(type_
↪descriptor)}};

// Define a member for the sequence
MemberDescriptor::_ref_type sequence_member_descriptor {traits<MemberDescriptor>
↪::make_shared()};
sequence_member_descriptor->name("bitmask_sequence");
sequence_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↪create_sequence_type(bitmask_type,
static_cast<uint32_t>(LENGTH_UNLIMITED))->build());
```

(continues on next page)

(continued from previous page)

```

    /* Alternative
       type_descriptor = traits<TypeDescriptor>::make_shared();
       type_descriptor->kind(TK_SEQUENCE);
       type_descriptor->element_type(bitmask_type);
       type_descriptor->bound().push_back(static_cast<uint32_t>(LENGTH_UNLIMITED));
       sequence_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↪create_type(
           type_descriptor)->build());
    */

    // Add the sequence member to the struct
    struct_builder->add_member(sequence_member_descriptor);

    sequence_member_descriptor = traits<MemberDescriptor>::make_shared();
    sequence_member_descriptor->name("short_sequence");
    sequence_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↪create_sequence_type(
        DynamicTypeBuilderFactory::get_instance()->get_primitive_type(TK_
↪INT16), 5)->build());

    /* Alternative
       type_descriptor = traits<TypeDescriptor>::make_shared();
       type_descriptor->kind(TK_SEQUENCE);
       type_descriptor->element_type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_INT16));
       type_descriptor->bound().push_back(5);
       sequence_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↪create_type(
           type_descriptor)->build());
    */

    // Add the sequence member to the struct
    struct_builder->add_member(sequence_member_descriptor);
    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

    // Set and retrieve values for the sequence member
    Int16Seq in_value = {1, 2};
    Int16Seq out_value;
    data->set_int16_values(data->get_member_id_by_name("short_sequence"), in_value);
    data->get_int16_values(out_value, data->get_member_id_by_name("short_sequence"));

    DynamicData::_ref_type sequence_data {data->loan_value(data->get_member_id_by_
↪name("short_sequence"))};
    // Set the two latest possible values on the sequence
    sequence_data->set_int16_values(3, in_value);
    // Read every sequence value from index 1 to the end
    sequence_data->get_int16_values(out_value, 1);

```

(continues on next page)

(continued from previous page)

```

int16_t in_simple_value = 8;
int16_t out_simple_value;
sequence_data->set_int16_value(2, in_simple_value);
sequence_data->get_int16_value(out_simple_value, 2);

data->return_loaned_value(sequence_data);

```

For a detailed explanation about the XML definition of this type, please refer to [XML Sequence Types](#).

Array types

Array types are multi-dimensional collections of any type. The `TypeKind` used to identify arrays is `TK_ARRAY`. `TypeDescriptor` `element_type` property must be set with the collection's type. Additionally, `bound` property must be configured with the sequence containing the size of each dimension. Bound sequence must have at least one dimension and it is not allowed for any dimension to have size 0.

`DynamicTypeBuilderFactory` exposes the function `DynamicTypeBuilderFactory::create_array_type` to facilitate the creation of this type. This API requires the type stored in the collection and the sequence with the collection's dimensions.

`DynamicData` class provides specific `get_values()` and `set_values()` functions for each primitive type, allowing users to easily work with arrays of primitives types. For arrays of more complex types, please refer to [Managing Complex Types Data](#).

Note: Multi-dimensional arrays *flatten* every dimension into a single-dimension array.

Primitive type promotion is also applicable for these methods.

If a specific range of values within the array are to be modified, passing the starting index to `get_values()` / `set_values()` would only manage data from that element forward until the length of the given input. Specific collection's element can be also be modified using the `get_value()` / `set_value()` passing the index of the element to be modified.

IDL

```

struct ArrayStruct
{
    long long_array[2][3][4];
};

```

XML

```

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

```

C++

```

// Define a struct type to contain the array
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;
type_descriptor->kind(TK_STRUCTURE);

```

(continues on next page)

(continued from previous page)

```

    type_descriptor->name("ArrayStruct");
    DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↪instance()->
                                                create_type(type_
↪descriptor)}};
    // Define a member for the array
    MemberDescriptor::_ref_type array_member_descriptor {traits<MemberDescriptor>
↪::make_shared()};
    array_member_descriptor->name("long_array");
    array_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_
↪array_type(
        DynamicTypeBuilderFactory::get_instance()->get_primitive_type(TK_
↪INT32), { 2, 3, 4 }->build());

    /* Alternative
       type_descriptor = traits<TypeDescriptor>::make_shared();
       type_descriptor->kind(TK_ARRAY);
       type_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_INT32));
       type_descriptor->bound().push_back(2);
       type_descriptor->bound().push_back(3);
       type_descriptor->bound().push_back(4);
       array_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↪create_type(
           type_descriptor)->build());
    */

    // Add the array member to the struct
    struct_builder->add_member(array_member_descriptor);
    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

    // Set and retrieve values for the array member
    Int32Seq in_value = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
↪18, 19, 20, 21, 23, 23, 24};
    Int32Seq out_value;
    data->set_int32_values(data->get_member_id_by_name("long_array"), in_value);
    data->get_int32_values(out_value, data->get_member_id_by_name("long_array"));

    DynamicData::_ref_type array_data {data->loan_value(data->get_member_id_by_name(
↪"long_array"))};
    // Set the two latest possible values on the array
    Int32Seq small_in_value = {0, 1};
    array_data->set_int32_values(22, small_in_value);
    // Read every array value from index 1 to the end
    array_data->get_int32_values(out_value, 1);

    int32_t in_simple_value = 8;
    int32_t out_simple_value;

```

(continues on next page)

(continued from previous page)

```
array_data->set_int32_value(2, in_simple_value);
array_data->get_int32_value(out_simple_value, 2);

data->return_loaned_value(array_data);
```

For a detailed explanation about the XML definition of this type, please refer to [XML Array Types](#).

Map Types

Map types are a collection of key/value pair types. Access to the value element is done through the key which is unique within the map type. The `TypeKind` used to identify maps is `TK_MAP`. `TypeDescriptor` `element_type` property must be set with the map value type. `TypeDescriptor` `key_type` property must be set with the map key type. Allowed key types are signed and unsigned integer types and string types.

Note: Currently, wide string keys are not supported as map keys.

Additionally, bound property must also be configured with the map's maximum length, or `LENGTH_UNLIMITED` in case of unbounded maps.

`DynamicTypeBuilderFactory` exposes the `DynamicTypeBuilderFactory::create_map_type` function to facilitate the creation of this type. This API requires the type of both the key and the value stored in the collection, and the collection's bound, using `LENGTH_UNLIMITED` in case of unbounded maps.

Manipulating map types data is more complex. First the `MemberId` corresponding to a specific key must be retrieved using `DynamicData::get_member_id_by_name` API. This API either returns the `MemberId` corresponding to the existing key or, if the key does not exist yet, it creates the key and returns the `memberId` associated to the just created key. In order to call this method, the correct string representation of the key value must be passed. The map value can now be set using the API corresponding to the map value type. For complex map values, please refer to [Managing Complex Types Data](#).

IDL

```
struct MapStruct
{
    map<string, MyAliasedBoundedString> string_alias_unbounded_map;
    map<short, long, 2> short_long_map;
};
```

XML

```
<struct name="MapStruct">
  <member name="string_alias_unbounded_map" type="nonBasic" nonBasicTypeName=
    ↪ "MyAliasedBoundedString" key_type="string" mapMaxLength="-1"/>
  <member name="short_long_map" type="int32" key_type="int16" mapMaxLength="2"/>
</struct>
```

C++

```
// Define a struct type to contain the map
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;

type_descriptor->kind(TK_STRUCTURE);
```

(continues on next page)

(continued from previous page)

```

    type_descriptor->name("MapStruct");
    DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↪instance()->
                                create_type(type_
↪descriptor)}};
    // Define a member for the map
    MemberDescriptor::_ref_type map_member_descriptor {traits<MemberDescriptor>
↪::make_shared()};
    map_member_descriptor->name("string_long_array_unbounded_map");
    map_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_
↪map_type(
        DynamicTypeBuilderFactory::get_instance()->create_string_type(static_cast
↪<uint32_t>(
            LENGTH_UNLIMITED))->build(), alias_bounded_string_type, static_cast
↪<uint32_t>(
            LENGTH_UNLIMITED))->build());

    /* Alternative
       type_descriptor = traits<TypeDescriptor>::make_shared();
       type_descriptor->kind(TK_MAP);
       type_descriptor->key_element_type(DynamicTypeBuilderFactory::get_instance()->
↪create_string_type(
           static_cast<uint32_t>(LENGTH_UNLIMITED))->build());
       type_descriptor->element_type(alias_bounded_string_type);
       type_descriptor->bound().push_back(static_cast<uint32_t>(LENGTH_UNLIMITED));
       map_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↪create_type(
           type_descriptor)->build());
    */

    // Add the map member to the struct
    struct_builder->add_member(map_member_descriptor);

    map_member_descriptor = traits<MemberDescriptor>::make_shared();
    map_member_descriptor->name("short_long_map");
    map_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_
↪map_type(
        DynamicTypeBuilderFactory::get_instance()->get_primitive_type(TK_
↪INT16),
        DynamicTypeBuilderFactory::get_instance()->get_primitive_type(TK_
↪INT32), 2)->build());

    /* Alternative
       type_descriptor = traits<TypeDescriptor>::make_shared();
       type_descriptor->kind(TK_MAP);
       type_descriptor->key_element_type(DynamicTypeBuilderFactory::get_instance()->
↪get_primitive_type(TK_INT16));
       type_descriptor->element_type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_INT32));
       type_descriptor->bound().push_back(2);
       map_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↪create_type(

```

(continues on next page)

(continued from previous page)

```

        type_descriptor)->build());
    */

    struct_builder->add_member(map_member_descriptor);
    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(struct_type)};

    // Get the data loan of the map member
    DynamicData::_ref_type map_data = data->loan_value(data->get_member_id_by_name(
↪"short_long_map"));

    // Set and retrieve values for the map member
    int32_t key {1};
    int32_t in_value {2};
    int32_t out_value;
    map_data->set_int32_value(map_data->get_member_id_by_name(std::to_string(key)), ↪
↪in_value);
    map_data->get_int32_value(out_value, map_data->get_member_id_by_name(std::to_
↪string(key)));

    // Return de data loan of the map member
    data->return_loaned_value(map_data);

```

For a detailed explanation about the XML definition of this type, please refer to *XML Map Types*.

Structure Types

Structure types are an aggregation of members of different types. The `TypeKind` used to identify structures is `TK_STRUCTURE`. Structure types have single inheritance support, so a structure type might extend one other already defined structure. The structure type which is extended should be configured in the `TypeDescriptor` `base_type` property. *Structure extensibility* might be configured using `TypeDescriptor` `extensibility_kind` property.

Note: Currently, `@nested` builtin annotation is not supported.

Structure members must be configured using the `DynamicTypeBuilder` by calling the `DynamicTypeBuilder::add_member` function with the corresponding `MemberDescriptor`.

Note: Empty structures, with no members, are allowed.

Member name is configured using `MemberDescriptor` `name` property and the member type is set using `type` property. Structure members might be keyed to create *topic instances* by setting the `MemberDescriptor` `is_key` property. The behavior is the same as setting the `@key` *builtin annotation*. Additionally, `MemberDescriptor` `default_value` property might be set to configure the member default value, and `MemberDescriptor` `id` property sets explicitly the member ID. This behavior is the same as setting the `@default` and `@id` *builtin annotations*.

Note: Currently, Fast DDS-Gen does not support `@default` builtin annotation.

Note: Currently, Dynamic Language Binding API implementation does not support the following builtin annotations:

- `@optional`
 - `@must_understand`
 - `@external`
 - `@try_construct`
-

Member data can be managed using the corresponding accessors for the underlying member type. Member ID might be retrieved using `DynamicData::get_member_id_by_name` API. For managing complex type members, please refer to *Managing Complex Types Data*.

IDL

```
struct InnerStruct
{
    @id(0x10) long first;
};

struct ParentStruct
{
    float first;
    long long second;
};

struct ComplexStruct : ParentStruct
{
    InnerStruct complex_member;
};
```

XML

```
<struct name="InnerStruct">
  <!-- XML does not support setting Member ID -->
  <member name="first" type="int32"/>
</struct>

<!-- TODO(XTypes: Fix inheritance loading from XML profile) Fast DDS#4626 -->
<!-- <struct name="ParentStruct">
  <member name="first" type="float32"/>
  <member name="second" type="int64"/>
</struct>

<struct name="ComplexStruct" baseType="ParentStruct">
  <member name="complex_member" type="nonBasic" nonBasicTypeName="InnerStruct"/>
</struct> -->
```

C++

```

// Create inner struct type
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;
type_descriptor->kind(eprosima::fastdds::dds::TK_STRUCTURE);
type_descriptor->name("InnerStruct");
DynamicTypeBuilder::_ref_type builder {DynamicTypeBuilderFactory::get_instance()-
↪create_type(type_descriptor)};
// Add members to the inner struct type
MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
get_primitive_type(eprosima::fastdds::dds::TK_INT32));
member_descriptor->name("first");
member_descriptor->id(16);
builder->add_member(member_descriptor);
// Build the inner struct type
DynamicType::_ref_type inner_struct_type {builder->build()};

// Create parent struct type
TypeDescriptor::_ref_type parentstruct_type_descriptor {traits<TypeDescriptor>
↪::make_shared()};
parentstruct_type_descriptor->kind(TK_STRUCTURE);
parentstruct_type_descriptor->name("ParentStruct");
DynamicTypeBuilder::_ref_type parentstruct_builder
↪{DynamicTypeBuilderFactory::get_instance()->
create_
↪type(parentstruct_type_descriptor)};
// Add members to the parent struct type
MemberDescriptor::_ref_type parentstruct_member {traits<MemberDescriptor>::make_
↪shared()};
parentstruct_member->name("first");
parentstruct_member->type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_FLOAT32));
parentstruct_builder->add_member(parentstruct_member);
parentstruct_member = traits<MemberDescriptor>::make_shared();
parentstruct_member->name("second");
parentstruct_member->type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_INT64));
parentstruct_builder->add_member(parentstruct_member);
// Build the parent struct type
DynamicType::_ref_type parentstruct_type = parentstruct_builder->build();

// Create complex struct type
TypeDescriptor::_ref_type complexstruct_type_descriptor {traits<TypeDescriptor>
↪::make_shared()};
complexstruct_type_descriptor->kind(TK_STRUCTURE);
complexstruct_type_descriptor->name("ComplexStruct");
complexstruct_type_descriptor->base_type(parentstruct_type);
DynamicTypeBuilder::_ref_type complexstruct_builder
↪{DynamicTypeBuilderFactory::get_instance()->
create_
↪type(complexstruct_type_descriptor)};
// Add members to the complex struct type

```

(continues on next page)

(continued from previous page)

```

MemberDescriptor::_ref_type complexstruct_member {traits<MemberDescriptor>::make_
↪shared()};
complexstruct_member->name("complex_member");
complexstruct_member->type(inner_struct_type);
complexstruct_builder->add_member(complexstruct_member);

// Build the complex struct type
DynamicType::_ref_type complexstruct_type = complexstruct_builder->build();
// Create dynamic data based on the struct type
DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(complexstruct_type)};

// Set and retrieve values for member of type float
float in_value {3.14};
float out_value {0.0};
data->set_float32_value(data->get_member_id_by_name("first"), in_value);
data->get_float32_value(out_value, data->get_member_id_by_name("first"));

```

For a detailed explanation about the XML definition of this type, please refer to *XML Structure Types*.

Union Types

Union types are a special type of structure type where only one member exists. The `TypeKind` used to identify unions is `TK_UNION`. Member selection is performed by setting another special member called discriminator. The discriminator type must be defined using `TypeDescriptor` `discriminator_type` property. Supported discriminator `TypeKind` are the following:

- `TK_BOOLEAN`
- `TK_BYTE`
- `TK_CHAR8`
- `TK_CHAR16`
- `TK_INT8`
- `TK_UINT8`
- `TK_INT16`
- `TK_UINT16`
- `TK_INT32`
- `TK_UINT32`
- `TK_INT64`
- `TK_UINT64`
- `TK_ENUM`
- `TK_ALIAS` that resolves, directly or indirectly to one of the aforementioned types.

Union extensibility might be configured using `TypeDescriptor` `extensibility_kind` property.

Note: Currently, `@nested` builtin annotation is not supported.

Union members must be configured using the `DynamicTypeBuilder` by calling the `DynamicTypeBuilder::add_member` function with the corresponding `MemberDescriptor`. At least one union member must be added to the union type. Union member name is configured using `MemberDescriptor` `name` property and the member type is set using `type` property. It is also mandatory to either set `MemberDescriptor` `is_default_label` property or configure the `label` property. This latest property indicates the discriminator values that select this specific member. If no labels are configured, then the flag indicating the member to be the default one, must be set. Only one union member must be configured as default.

Additionally, `MemberDescriptor` `default_value` property might be set to configure the member default value, and `MemberDescriptor` `id` property sets explicitly the member ID. This behavior is the same as setting the `@default` and `@id` *builtin annotations*.

Note: Currently, Fast DDS-Gen does not support `@default` builtin annotation.

Note: Currently, Dynamic Language Binding API implementation does not support the following builtin annotations:

- `@optional`
 - `@must_understand`
 - `@external`
 - `@try_construct`
-

Member data can be managed using the corresponding accessors for the underlying member type. Setting a member automatically changes the discriminator value selecting the set member. When reading a member, the discriminator must be selecting the member being read. Member ID might be retrieved using `DynamicData::get_member_id_by_name` API. For managing complex type members, please refer to *Managing Complex Types Data*.

IDL

```
union InnerUnion switch (short)
{
    case 0:
        @id(0x10) PrimitivesStruct first;
    case 1:
    default:
        long long second;
};

union ComplexUnion switch (long)
{
    case 0:
    case 1:
        long third;
    default:
        InnerUnion fourth;
};
```

XML

```
<union name="InnerUnion">
  <discriminator type="int16"/>
  <case>
```

(continues on next page)

(continued from previous page)

```

    <caseDiscriminator value="0"/>
    <member name="first" type="nonBasic" nonBasicTypeName="PrimitivesStruct"/>
  </case>
  <case>
    <caseDiscriminator value="1"/>
    <caseDiscriminator value="default"/>
    <member name="second" type="int64"/>
  </case>
</union>
<union name="ComplexUnion">
  <discriminator type="int32"/>
  <case>
    <caseDiscriminator value="0"/>
    <caseDiscriminator value="1"/>
    <member name="third" type="int32"/>
  </case>
  <case>
    <caseDiscriminator value="default"/>
    <member name="fourth" type="nonBasic" nonBasicTypeName="InnerUnion"/>
  </case>
</union>

```

C++

```

// Define a struct type to contain the union
// Create the inner union
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;

type_descriptor->kind(TK_UNION);
type_descriptor->name("InnerUnion");
type_descriptor->discriminator_type(DynamicTypeBuilderFactory::get_instance()->
    get_primitive_type(TK_INT16));
DynamicTypeBuilder::_ref_type builder {DynamicTypeBuilderFactory::get_instance()-
↪>

    create_type(type_descriptor)};

// Add members to the inner union type
MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};
member_descriptor->type(struct_type);
member_descriptor->name("first");
member_descriptor->id(16);
member_descriptor->label({0});
builder->add_member(member_descriptor);
member_descriptor = traits<MemberDescriptor>::make_shared();
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
    get_primitive_type(TK_INT64));
member_descriptor->name("second");
member_descriptor->label({1});
member_descriptor->is_default_label(true);
builder->add_member(member_descriptor);
// Build the inner union type
DynamicType::_ref_type inner_union_type {builder->build()};

```

(continues on next page)

(continued from previous page)

```

// Create a complex union type
type_descriptor = traits<TypeDescriptor>::make_shared();
type_descriptor->kind(TK_UNION);
type_descriptor->name("ComplexUnion");
type_descriptor->discriminator_type(DynamicTypeBuilderFactory::get_instance()->
    get_primitive_type(TK_INT32));
builder = DynamicTypeBuilderFactory::get_instance()->create_type(type_
↪descriptor);
// Add members to the complex union type
member_descriptor = traits<MemberDescriptor>::make_shared();
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
    get_primitive_type(TK_INT32));
member_descriptor->name("third");
member_descriptor->label({0, 1});
builder->add_member(member_descriptor);
member_descriptor = traits<MemberDescriptor>::make_shared();
member_descriptor->type(inner_union_type);
member_descriptor->name("fourth");
member_descriptor->is_default_label(true);
builder->add_member(member_descriptor);
// Build the complex union type
DynamicType::_ref_type union_type {builder->build()};

// Create dynamic data based on the union type
DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↪data(union_type)};
// Get the loan for the InnerUnion member
DynamicData::_ref_type union_data = data->loan_value(data->get_member_id_by_name(
↪"InnerUnion"));

// Set and retrieve values for the long long member within InnerUnion member
int64_t in_value {2};
int64_t out_value;
union_data->set_int64_value(union_data->get_member_id_by_name("second"), in_
↪value);
union_data->get_int64_value(out_value, union_data->get_member_id_by_name("second
↪"));
// Return de data loan of the member
data->return_loaned_value(union_data);

```

For a detailed explanation about the XML definition of this type, please refer to *XML Union Types*.

Bitset Types

Bitset types are an aggregation of bitfields. The `TypeKind` used to identify bitsets is `TK_BITSET`. `bound` property contains the sequence with the bitfield's bitcount (number of bits). In order to be consistent, the length of the bound sequence must agree with the number of bitfields. Bitset types have single inheritance support, so a bitset type might extend one other already defined bitset. The bitset type which is extended should be configured in the `TypeDescriptor` `base_type` property.

Bitfields must be configured using the `DynamicTypeBuilder` by calling the `DynamicTypeBuilder::add_member` function with the corresponding `MemberDescriptor`. At least one bitfield is required for the bitset to be consistent. Bitfield name is configured using `MemberDescriptor` `name` property, and the bitfield initial bit position is set using `MemberDescriptor` `id` property.

Note: For derived bitsets, the first bitfield initial position must be after the bits defined by the parent bitset type.

A bitfield manages exclusively a set of bits, so no bitfield superposition is allowed. Additionally, `MemberDescriptor` `type` property might be set to configure an integer type to access bitfield data. If not set, the minimum unsigned integer type is used instead:

Number of bits	C++ holder type
1	<code>bool</code>
2-8	<code>uint8_t</code>
9-16	<code>uint16_t</code>
17-32	<code>uint32_t</code>
33-64	<code>uint64_t</code>

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.

IDL

```
bitset ParentBitSet
{
    bitfield<3> a;
    bitfield<1> b;
    bitfield<4>;
    bitfield<10> c;
    bitfield<12, short> d;
};

bitset ChildBitSet : ParentBitSet
{
    bitfield<1> e;
    bitfield<20, unsigned long> f;
};

struct BitsetStruct
{
    ChildBitSet my_bitset;
};
```

XML

```

<bitset name="ParentBitset">
  <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>

<!-- TODO(XTypes: Fix inheritance loading from XML profile) Fast DDS#4626 -->
<!--<bitset name="ChildBitSet" baseType="ParentBitSet">
  <bitfield name="e" bit_bound="1"/>
  <bitfield name="f" bit_bound="20" type="uint32"/>
</bitset>

<struct name="BitsetStruct">
  <member name="my_bitset" type="nonBasic" nonBasicTypeName="ChildBitSet"/>
</struct>-->

```

C++

```

// Define a struct type to contain the bitset
TypeDescriptor::_ref_type struct_type_descriptor {traits<TypeDescriptor>::make_
↳ shared()};
struct_type_descriptor->kind(TK_STRUCTURE);
struct_type_descriptor->name("BitsetStruct");
DynamicTypeBuilder::_ref_type struct_builder {DynamicTypeBuilderFactory::get_
↳ instance()->create_type(
                                                    struct_type_descriptor)};

// Define type for parent bitset
TypeDescriptor::_ref_type bitset_type_descriptor {traits<TypeDescriptor>::make_
↳ shared()};
bitset_type_descriptor->kind(TK_BITSET);
bitset_type_descriptor->name("ParentBitSet");
bitset_type_descriptor->bound({3, 1, 10, 12});
DynamicTypeBuilder::_ref_type bitset_builder {DynamicTypeBuilderFactory::get_
↳ instance()->create_type(
                                                    bitset_type_descriptor)};

// Add members to the bitset type
MemberDescriptor::_ref_type bitset_member_descriptor {traits<MemberDescriptor>
↳ ::make_shared()};
bitset_member_descriptor->name("a");
bitset_member_descriptor->id(0);
bitset_builder->add_member(bitset_member_descriptor);
bitset_member_descriptor = traits<MemberDescriptor>::make_shared();
bitset_member_descriptor->name("b");
bitset_member_descriptor->id(3);
bitset_builder->add_member(bitset_member_descriptor);
bitset_member_descriptor = traits<MemberDescriptor>::make_shared();
bitset_member_descriptor->name("c");
bitset_member_descriptor->id(8);
bitset_builder->add_member(bitset_member_descriptor);
bitset_member_descriptor = traits<MemberDescriptor>::make_shared();

```

(continues on next page)

(continued from previous page)

```

    bitset_member_descriptor->name("d");
    bitset_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->get_
↳primitive_type(TK_INT16));
    bitset_member_descriptor->id(18);
    bitset_builder->add_member(bitset_member_descriptor);
    // Build the bitset type
    DynamicType::_ref_type parentbitset_type = bitset_builder->build();

    // Create child bitset type
    TypeDescriptor::_ref_type childbitset_type_descriptor {traits<TypeDescriptor>
↳::make_shared()};
    childbitset_type_descriptor->kind(TK_BITSET);
    childbitset_type_descriptor->name("ChildBitSet");
    childbitset_type_descriptor->base_type(parentbitset_type);
    childbitset_type_descriptor->bound({1, 20});
    DynamicTypeBuilder::_ref_type childbitset_builder
↳{DynamicTypeBuilderFactory::get_instance()->create_type(
                                                    childbitset_type_
↳descriptor)}};
    // Add members to the child bitset type
    MemberDescriptor::_ref_type childbitset_member_descriptor {traits
↳<MemberDescriptor>::make_shared()};
    childbitset_member_descriptor->name("e");
    childbitset_member_descriptor->id(30);
    childbitset_builder->add_member(childbitset_member_descriptor);
    childbitset_member_descriptor = traits<MemberDescriptor>::make_shared();
    childbitset_member_descriptor->name("d");
    childbitset_member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->
↳get_primitive_type(TK_UINT32));
    childbitset_member_descriptor->id(31);
    childbitset_builder->add_member(childbitset_member_descriptor);
    // Build the child bitset type
    DynamicType::_ref_type bitset_type = childbitset_builder->build();

    // Add the bitset member to the struct
    MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↳shared()};
    member_descriptor->name("my_bitset");
    member_descriptor->type(bitset_type);
    struct_builder->add_member(member_descriptor);
    // Build the struct type
    DynamicType::_ref_type struct_type {struct_builder->build()};
    // Create dynamic data based on the struct type
    DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↳data(struct_type)};
    // Get the loan for the bitset member
    DynamicData::_ref_type bitset_data = data->loan_value(data->get_member_id_by_
↳name("my_bitset"));

    // Set and retrieve bitfield values
    int16_t in_value {2};
    int16_t out_value;

```

(continues on next page)

(continued from previous page)

```
bitset_data->set_int16_value(bitset_data->get_member_id_by_name("d"), in_value);
bitset_data->get_int16_value(out_value, bitset_data->get_member_id_by_name("d"));
// Return de data loan of the member
data->return_loaned_value(bitset_data);
```

For a detailed explanation about the XML definition of this type, please refer to *XML Bitset Types*.

Annotations

Custom annotations

Both types and type members might be annotated using `DynamicTypeBuilder::apply_annotation` and `DynamicTypeBuilder::apply_annotation_to_member` API respectively.

Annotations are defined using an `AnnotationDescriptor` which provides two properties: `type` and `value`. The annotation type must be the `DynamicType` representing the annotation being applied. The `TypeKind` used to identify annotations is `TK_ANNOTATION`. The annotation name is set in `TypeDescriptor` `name` property.

The annotation type might have any number of parameters. Annotation parameters must be configured using the `DynamicTypeBuilder` by calling the `DynamicTypeBuilder::add_member` function with the corresponding `MemberDescriptor`. Annotation parameters must define the annotation parameter name in `MemberDescriptor` `name` property, and the parameter type using `type` property. Only the following types can be used to define an annotation parameter:

- *Primitive types*
- *String types*
- *Enumeration types*

Note: Currently, wide string types are not supported as annotation parameters.

Annotation parameter values are defined with `AnnotationDescriptor` `value` property using `AnnotationDescriptor::set_value()`. The annotation parameter name provided to the API must coincide with the one defined with the annotation type. The annotation parameter value must be converted to its string representation.

Note: Currently, custom annotations are not supported with *XML DynamicTypes*.

IDL

```
@annotation MyAnnotation
{
    short length;
};

@MyAnnotation(length = 5)
struct AnnotatedStruct
{
    @MyAnnotation(length = 10) string string_var;
};
```


XML

```
<!-- XML Types Profiles does not support defining/setting custom annotations -->
```

C++

```
// Create the structure to annotate
TypeDescriptor::_ref_type type_descriptor {traits<TypeDescriptor>::make_shared()}
↪;
type_descriptor->kind(TK_STRUCTURE);
type_descriptor->name("AnnotatedStruct");
DynamicTypeBuilder::_ref_type type_builder {DynamicTypeBuilderFactory::get_
↪instance()->
create_type(type_descriptor)}
↪;
MemberDescriptor::_ref_type member_descriptor {traits<MemberDescriptor>::make_
↪shared()};
member_descriptor->name("string_var");
member_descriptor->type(DynamicTypeBuilderFactory::get_instance()->create_string_
↪type(static_cast<uint32_t>(
LENGTH_UNLIMITED))->build());
type_builder->add_member(member_descriptor);

// Create the annotation type
AnnotationDescriptor::_ref_type annotation_descriptor {traits
↪<AnnotationDescriptor>::make_shared()};
TypeDescriptor::_ref_type annotation_type {traits<TypeDescriptor>::make_shared()}
↪;
annotation_type->kind(TK_ANNOTATION);
annotation_type->name("MyAnnotation");
DynamicTypeBuilder::_ref_type annotation_builder {DynamicTypeBuilderFactory::get_
↪instance()->
create_type(annotation_
↪type)};

// Add members to the annotation type
MemberDescriptor::_ref_type annotation_parameter {traits<MemberDescriptor>::make_
↪shared()};
annotation_parameter->name("length");
annotation_parameter->type(DynamicTypeBuilderFactory::get_instance()->get_
↪primitive_type(TK_INT16));
annotation_builder->add_member(annotation_parameter);
// Set the annotation type using the annotation descriptor
annotation_descriptor->type(annotation_builder->build());
// Set the value of the annotation
annotation_descriptor->set_value("length", std::to_string(5));
// Apply the annotation to the structure
type_builder->apply_annotation(annotation_descriptor);

// Reuse annotation descriptor to annotate struct member
annotation_descriptor = traits<AnnotationDescriptor>::make_shared();
annotation_descriptor->type(annotation_builder->build());
annotation_descriptor->set_value("length", std::to_string(10));
```

(continues on next page)

(continued from previous page)

```
DynamicTypeMember::_ref_type member;
type_builder->get_member_by_name(member, "string_var");
type_builder->apply_annotation_to_member(member->get_id(), annotation_
↪descriptor);
```

Builtin annotations

Beside the user-defined custom annotations, there are a number of builtin annotations that have already mentioned throughout this section.

The table below summarizes the builtin annotations that can be applied using the Dynamic Language Binding API. Please, refer to *builtin annotations* for the complete list and their behavior.

Builtin annotation	Dynamic Language Binding API	Dynamic language support	Lan-Binding	XML Type support	Dynamic profiles
@appendable	TypeDescriptor extensibility_kind property.				
@bit_bound	TypeDescriptor bound property for <i>Bitset Types</i> . MemberDescriptor type property for <i>Enumeration Types</i> .			(Enumeration types not configurable).	
@default	MemberDescriptor default_value property.				
default_label	MemberDescriptor is_default_label property.				
@extensible	TypeDescriptor extensibility_kind property.				
@external	MemberDescriptor is_shared property.				
@final	TypeDescriptor extensibility_kind property.				
@id	MemberDescriptor id property.				
@key / @Key	MemberDescriptor is_key property.				
@mutable	TypeDescriptor extensibility_kind property.				
@nested	TypeDescriptor is_nested property.				
@optional	MemberDescriptor is_optional property.				
@position	MemberDescriptor id property.				
@try_construct	MemberDescriptor try_construct_kind property.				
@value	MemberDescriptor default_value property.				
@verbatim	VerbatimTextDescriptor				

Managing Complex Types Data

Some DynamicData instances manage complex types that cannot be directly modified with the primitive getters and setters. Dynamic Language Binding provides two possible approaches for managing complex data types:

1. `DynamicData::get_complex_value` and `DynamicData::set_complex_value`: This API allows to get/set generic DynamicData. The main difference with the next approach is that this API performs always a copy.
2. `DynamicData::loan_value`: this API allows to loan a reference to a DynamicData to work with preventing the data copy. `DynamicData::return_loaned_value` must be called to return the loan. Calling `DynamicData::loan_value` for an already loaned value will fail.

The following snippet includes an example of managing complex data using the same structure as the one defined in *Structure types*:

```

// Create dynamic data based on the struct type
DynamicData::_ref_type data {DynamicDataFactory::get_instance()->create_
↳data(complexstruct_type)};

// Get/Set complex API (copy)
DynamicData::_ref_type complex_data;
data->get_complex_value(complex_data, data->get_member_id_by_name("complex_member
↳"));
// Set data
int32_t in_value {10};
complex_data->set_int32_value(complex_data->get_member_id_by_name("first"), in_
↳value);
data->set_complex_value(data->get_member_id_by_name("complex_member"), complex_
↳data);

// Loan API
DynamicData::_ref_type loan_data = data->loan_value(data->get_member_id_by_name(
↳"complex_member"));
loan_data->set_int32_value(loan_data->get_member_id_by_name("first"), in_value);
int32_t out_value;
loan_data->get_int32_value(out_value, loan_data->get_member_id_by_name("first"));
data->return_loaned_value(loan_data);

```

Note: Type compatibility rules among evolved types are still unsupported in Fast DDS.

7.30 Typical Use-Cases

Fast DDS is highly configurable, which allows for its use in a large number of scenarios. This section provides configuration examples for the following typical use cases when dealing with distributed systems:

- *Large Data mode and Fast DDS over TCP*. Describes how to configure *Fast DDS* to use the `LARGE_DATA` builtin transports mode. This mode enables efficient utilization of TCP transport without the need for constant reconfiguration during deployment changes. It optimizes communication performance for large data samples over lossy networks by employing a combination of UDP and TCP/SHM transports.
- *Fast DDS over WIFI*. Presents a case where *Discovery* through multicast communication is a challenge. This example shows how to:
 - Configure an initial list of peers with the address-port pairs of the remote participants (see *Configuring Initial Peers*).
 - Disable the multicast discovery mechanism (see *Disabling multicast discovery*).
 - Configure a SERVER discovery mechanism (see *Discovery Server*).
- *Well Known Network Deployments*. Describes a situation where the entire entity network topology (Participants, Publishers, Subscribers, and their addresses and ports) are known beforehand. In these kind of environments, *Fast DDS* allows to completely avoid the discovery phase configuring a `STATIC` discovery mechanism.
- *Topics with many subscribers*. In cases where there are many *DataReaders* subscribed to the same *Topic*, using multicast delivery can help reducing the overhead in the network and CPU.

- *Large Data Rates*. Presents configuration options that can improve the performance in scenarios where the amount of data exchanged between a *Publisher* and a *Subscriber* is large, either because of the data size or because the message rate. The examples describe how to:
 - Use TCP based communications (see *Large Data mode and Fast DDS over TCP*).
 - Configure the socket buffer size (see *increase the buffers size*).
 - Limit the publication rate (see *Flow Controllers*).
 - Tune the size of the socket buffers (see *Increasing socket buffers size*).
 - Tune the Heartbeat period (see *Tuning Heartbeat Period*).
 - Configure a non-strict reliable mode (see *Using Non-strict Reliability*).
- *Real-time behavior*. Describes the configuration options that allows using *Fast DDS* on a real-time scenario. The examples describe how to:
 - Configure memory management to avoid dynamic memory allocation (see *Tuning allocations*).
 - Limit the blocking time of API functions to have a predictable response time (see *Non-blocking calls*).
- *Reduce memory usage*. For use cases with memory consumption constraints, *Fast DDS* can be configured to reduce memory footprint to a minimum by adjusting different QoS policies.
- *Zero-Copy communication*. Under certain constraints, *Fast DDS* can provide application level communication between publishing and subscribing nodes avoiding any data copy during the process.
- *Unique network flows*. This use case illustrates the APIs that allow for the request of unique network flows, and for the identification of those in use.
- *Dynamic network interfaces*. If the network interfaces are expected to change while the application is running, *Fast DDS* provides an easy way of re-scanning the available interfaces and including them.
- *Statistics module*. This use case explains how to enable the Statistics module within the monitored application, and how to create a statistics monitoring application.
- *ROS 2 using Fast DDS middleware*. Since *Fast DDS* is the default middleware implementation in every *OSRF Robot Operation System 2 (ROS 2)* long term (LTS) releases and most of the non-LTS releases, this documentation includes a whole independent section to show the use of the library in ROS 2, and how to take full advantage of *Fast DDS* wide set of capabilities in a ROS 2 project.
- *How to use eProsima DDS Record and Replay (rosbag2 and DDS)*. Instructions on how to tune your application to be able to record and replay your DDS messages using ROS 2 rosbag2 package.
- *Request-Reply communication*. Although *Fast DDS* provides a *Publisher-Subscriber* communication protocol, it offers a mechanism to use *Request-Reply* communications. This use case explains how to use the APIs to communicate two application using *Request-Reply*.
- *Remote type discovery and endpoint matching*. This use case leverages the *Remote Data Types Discovery* to create endpoints and start communicating with remote discovered endpoints with previously unknown data types.

7.30.1 Large Data mode and Fast DDS over TCP

As explained in *TCP Transport*, Fast DDS offers the possibility to communicate nodes within distributed applications with DDS over a TCP transport layer. This has the advantage of leveraging the builtin flow control and reliability of the TCP protocol, which in the context of DDS applications is best suited for the transmission of large payloads, i.e. large data samples, over lossy networks. Examples of such cases would be transmitting video or large point clouds resulting from laser scanning over WiFi links.

The configuration of the TCP transport typically involves an *a priori* knowledge of the deployment in order to set *Initial peers* for *Discovery*, which may not always be possible and creates difficulties when reallocating nodes of the distributed applications, as the entire discovery configuration needs to be changed. To overcome this problem, Fast DDS presents the `LARGE_DATA` builtin transports configuration as an approach for leveraging the Fast DDS' TCP transport capabilities while at the same time not requiring configuration modifications when the deployment changes over time.

`LARGE_DATA` has been specifically designed to improve communication performance of large data samples over lossy networks. When configured, UDP transport will exclusively be used during the *PDP discovery* phase, taking advantage of the more reliable TCP/SHM for the remainder of the communication process. Fast DDS offers an extremely straightforward implementation for this mode through an environment variable, XML profiles or via code. Additionally, it is possible to use the builtin transports configuration options to customize the `LARGE_DATA` mode to better suit the specific requirements of each user.

For a video demonstration showcasing a practical example of this configuration, please refer to: [Large Data communication with ROS 2](#).

Also, it is possible to enable TCP communication while using *Discovery Server* to manage *Discovery*.

Large Data Mode

The following snippets show how to configure *Fast DDS DomainParticipants* to run the *PDP discovery* phase over UDP multicast and communicate application data over a *TCP Transport* or *Shared Memory Transport*, which is called the `LARGE_DATA` configuration (See *Managing the Builtin Transports*). With this approach, applications managing large samples can benefit from transmitting their data over TCP or SHM, while at the same time have the flexibility of automatic discovery. The `LARGE_DATA` mode can be set using the `FASTDDS_BUILTIN_TRANSPORTS` environment variable (see *FASTDDS_BUILTIN_TRANSPORTS*), XML profiles or via code.

Environment Variable

```
export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <!--
      UDP transport for PDP and SHM/TCPv4 transport for both EDP and application_
↪data
    -->
    <participant profile_name="large_data_builtin_transports" is_default_profile=
↪"true">
      <rtps>
        <builtinTransports>LARGE_DATA</builtinTransports>
      </rtps>
    </participant>
  </profiles>
</dds>
```

C++

```

eprosima::fastdds::dds::DomainParticipantQos pqos = PARTICIPANT_QOS_DEFAULT;

/* Transports configuration */
// UDPv4 transport for PDP over multicast and SHM / TCPv4 transport for EDP and
↪ application data
pqos.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_DATA);

/* Create participant as usual */
eprosima::fastdds::dds::DomainParticipant* participant =
    eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->create_
↪ participant(0, pqos);

```

Note: LARGE_DATA configuration of the builtin transports will also create a SHM transport along the UDP and TCP transports. Shared Memory will be used whenever it is possible. Manual configuration will be required if a TCP communication is required when SHM is feasible.

C++

```

eprosima::fastdds::dds::DomainParticipantQos pqos = PARTICIPANT_QOS_DEFAULT;

/* Transports configuration */
// UDPv4 transport for PDP over multicast
auto pdp_transport = std::make_shared<eprosima::fastdds::rtps::UDPv4TransportDescriptor>
↪ ();
pqos.transport().user_transports.push_back(pdp_transport);

// TCPv4 transport for EDP and application data (The listening port must to be unique for
// each participant in the same host)
constexpr uint16_t tcp_listening_port = 0;
auto data_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↪ ();
data_transport->add_listener_port(tcp_listening_port);
pqos.transport().user_transports.push_back(data_transport);

pqos.transport().use_builtin_transports = false;

/* Locators */
// Define locator for PDP over multicast
eprosima::fastrtps::rtps::Locator_t pdp_locator;
pdp_locator.kind = LOCATOR_KIND_UDPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(pdp_locator, "239.255.0.1");
pqos.wire_protocol().builtin.metatrafficMulticastLocatorList.push_back(pdp_locator);

// Define locator for EDP and user data
eprosima::fastrtps::rtps::Locator_t tcp_locator;
tcp_locator.kind = LOCATOR_KIND_TCPv4;
eprosima::fastrtps::rtps::IPLocator::setIPv4(tcp_locator, "0.0.0.0");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(tcp_locator, tcp_listening_port);
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(tcp_locator, tcp_listening_port);
pqos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(tcp_locator);
pqos.wire_protocol().default_unicast_locator_list.push_back(tcp_locator);

```

(continues on next page)

(continued from previous page)

```

/* Create participant as usual */
eprosima::fastdds::dds::DomainParticipant* participant =
    eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->create_
    ↪participant(0, pqos);

```

XML

```

<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <transport_descriptors>
      <!--
        UDP transport for PDP
      -->
      <transport_descriptor>
        <transport_id>pdp_transport</transport_id>
        <type>UD Pv4</type>
      </transport_descriptor>
      <!--
        TCP transport for both EDP and application data
      -->
      <transport_descriptor>
        <transport_id>data_transport</transport_id>
        <type>TCPv4</type>
      <!--
        Set listening port for the transport.
        This needs to be unique for each participant in the host
      -->
      <listening_ports>
        <port>0</port>
      </listening_ports>
    </transport_descriptor>
  </transport_descriptors>

  <participant profile_name="udp_multicast_pdp_w_tcp_data" is_default_profile="true
  ↪">
    <rtps>
      <!--
        Set participant transports
      -->
      <userTransports>
        <transport_id>pdp_transport</transport_id>
        <transport_id>data_transport</transport_id>
      </userTransports>
      <useBuiltinTransports>false</useBuiltinTransports>
      <!--
        Discovery configuration
      -->
      <builtin>
        <!--
          PDP over UDP multicast

```

(continues on next page)

(continued from previous page)

```

-->
<metatrafficMulticastLocatorList>
  <locator>
    <udp4>
      <address>239.255.0.1</address>
    </udp4>
  </locator>
</metatrafficMulticastLocatorList>
<!--
  EDP over TCP
-->
<metatrafficUnicastLocatorList>
  <locator>
    <tcpv4>
      <address>0.0.0.0</address>
      <physical_port>12345</physical_port>
      <port>12345</port>
    </tcpv4>
  </locator>
</metatrafficUnicastLocatorList>
</builtin>
<!--
  Application data over TCP
-->
<defaultUnicastLocatorList>
  <locator>
    <tcpv4>
      <address>0.0.0.0</address>
      <physical_port>12345</physical_port>
      <port>12345</port>
    </tcpv4>
  </locator>
</defaultUnicastLocatorList>
</rtps>
</participant>
</profiles>
</dds>

```

Large Data with configuration options

As it has been observed in *Large Data Mode*, `LARGE_DATA` builtin transports option offers an easy and efficient way to improve performance when working with large data. Nonetheless, custom configuration can help to enhance the performance even further.

Fast DDS provides configuration options to adjust the behavior of the builtin transports. This becomes particularly relevant when using the `LARGE_DATA` mode, as it enables increasing the maximum message size beyond 65500 KB and prevents fragmentation, leveraging the TCP and SHM transports.

All builtin transports can be configured by adjusting the following parameters:

- `max_msg_size`: Message maximum size that can be sent over the transport. Sending messages larger than this size will result in fragmentation. Its maximum value is $(2^{32})-1$ B for TCP and SHM and 65500 KB for UDP.

- **sockets_size**: Size of the send and receive socket buffers. This value must be higher or equal than the **max_msg_size** to obtain a valid configuration. It also defines the size of the shared memory segment, calculated as twice the value set. Its maximum value is $(2^{32})-1$ B.
- **non_blocking**: If set to true, the transport will use non-blocking sockets. This can be useful to avoid blocking the application if the socket buffers are full. However, some messages will be lost. Its default value is false.
- **tcp_negotiation_timeout**: It specifies the timeout duration for logical port negotiation. This parameter is useful for ensuring the availability of the logical port before data transmission, thus preventing message loss during the negotiation process. Conversely, it can delay the discovery process. The default value is 0, implying that discovery will occur as soon as possible, but the initial messages might be lost if reliability is set to **BEST_EFFORT_RELIABILITY_QOS**. This parameter is only valid for the **LARGE_DATA** mode.

Adjusting the maximum message size and the socket buffer sizes to a large enough value to accommodate the data to be sent can help improve the performance with large messages, as well as setting the transport to non-blocking mode.

In this way, it is possible to take advantage of the TCP transport to avoid fragmentation and use the non-blocking mode to avoid blocking the application when the socket buffers are full. This configuration can be used, for example, when streaming video, which will result in a significant increase in fluidity.

Note that even when using the **LARGE_DATA** mode within the same machine, the configuration options can prove useful for improving performance, as they affect the SHM transport. It is highly recommended to set a shared memory segment size large enough to accommodate the data to be sent. To achieve this, the **sockets_size** parameter must be set to a value at least half of the message size.

The following snippets show how to configure the **LARGE_DATA** mode:

Environment Variable

```
export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA?max_msg_size=1MB&sockets_size=1MB&non_
↪blocking=true&tcp_negotiation_timeout=50
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles">
  <profiles>
    <participant profile_name="large_data_builtin_transports_options" is_default_
↪profile="true">
      <rtps>
        <builtinTransports max_msg_size="200KB" sockets_size="200KB" non_
↪blocking="true" tcp_negotiation_timeout="50">LARGE_DATA</builtinTransports>
      </rtps>
    </participant>
  </profiles>
</dds>
```

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos = PARTICIPANT_QOS_DEFAULT;

/* Transports configuration */
// UDPv4 transport for PDP over multicast and SHM / TCPv4 transport for EDP and
↪application data
// Message Size and Sockets sizes of 200 KB + Non-blocking send + 50ms negotiation
↪timeout
eprosima::fastdds::rtps::BuiltinTransportsOptions large_data_options;
```

(continues on next page)

(continued from previous page)

```

large_data_options.maxMessageSize = 200000;
large_data_options.sockets_buffer_size = 200000;
large_data_options.non_blocking_send = true;
large_data_options.tcp_negotiation_timeout = 50;
pqos.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_DATA, large_data_
↪options);

/* Create participant as usual */
eprosima::fastdds::dds::DomainParticipant* participant =
    eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);

```

Note: To learn how to check and modify the default maximum system value for the socket buffers size, please refer to *Finding out system maximum values*.

Warning: Setting a `max_msg_size` value higher than 65500 KB with the DEFAULT or UDP modes will result in an error and the participant creation will fail.

TCP Communication with Discovery Server

Fast DDS Discovery Server consists on a client-server discovery mechanism, in which a server *DomainParticipant* operates as the central point of communication. It collects and processes the metatraffic sent by the client *DomainParticipants*, and then distributes the appropriate information among the rest of the clients. An extended description of the feature can be found at *Discovery Server Settings*.

To use TCP communication along with Discovery Server, both the server participant and the client participant need to use custom user transports. There exists several ways of configuring the server participant, being *Fast DDS CLI* the fastest solution:

Fast DDS CLI

It can be configured to work over a TCP transport layer by using the arguments `-t` and `-q` to set up the IP address and the TCP port, respectively. After sourcing the environment, the following command can be used to instantiate a server listening on localhost and port 12345 (see *CLI*).

```
fastdds discovery -i 0 -t 127.0.0.1 -q 12345
```

C++

The following snippet can be used to instantiate a server on IP 192.168.10.57 listening on port 12345.

```

eprosima::fastdds::dds::DomainParticipantQos qos = PARTICIPANT_QOS_DEFAULT;

// Configure the current participant as SERVER
qos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    eprosima::fastrtps::rtps::DiscoveryProtocol_t::SERVER;

// Add custom user transport with TCP port 12345
auto data_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↪();

```

(continues on next page)

(continued from previous page)

```

data_transport->add_listener_port(12345);
qos.transport().user_transports.push_back(data_transport);

// Define the listening locator to be on interface 192.168.10.57 and port 12345
constexpr uint16_t tcp_listening_port = 12345;
eprosima::fastrtps::rtps::Locator_t listening_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(listening_locator, "192.168.10.57");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(listening_locator, tcp_listening_
↳port);
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(listening_locator, tcp_listening_
↳port);
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(listening_locator);

// Set the GUID prefix to identify this server
std::istringstream("44.53.00.5f.45.50.52.4f.53.49.4d.41") >> qos.wire_protocol().prefix;

```

XML

The following snippet can be used to instantiate a server on IP 192.168.10.57 listening on port 12345.

```

<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>server_tcp_transport</transport_id>
        <type>TCPv4</type>
        <!--
          Set listening port for the transport.
          This port is where the clients will connect.
        -->
        <listening_ports>
          <port>12345</port>
        </listening_ports>
      </transport_descriptor>
    </transport_descriptors>
    <participant profile_name="TCP_SERVER" is_default_profile="true">
      <rtps>
        <builtin>
          <discovery_config>
            <discoveryProtocol>SERVER</discoveryProtocol>
          </discovery_config>
          <metatrafficUnicastLocatorList>
            <locator>
              <tcpv4>
                <address>192.168.10.57</address>
                <port>12345</port>
                <physical_port>12345</physical_port>
              </tcpv4>
            </locator>
          </metatrafficUnicastLocatorList>
        </builtin>
        <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
      </rtps>
    </participant>
  </profiles>
</dds>

```

(continues on next page)

(continued from previous page)

```

        <useBuiltinTransports>false</useBuiltinTransports>
        <userTransports>
            <transport_id>server_tcp_transport</transport_id>
        </userTransports>
    </rtps>
</participant>
</profiles>
</dds>

```

Fast DDS Discovery Server Example

It can be configured to work over a TCP transport layer by using the argument `--transport tcpv4`. The IP address and the TCP port can be set up with arguments `--listening-address` and `--listening-port`, respectively. From the *DiscoveryServerExample* folder, the following command can be used to instantiate a server listening on localhost and port 12345.

```
./DiscoveryServerExample server --transport tcpv4 --listening-address 127.0.0.1 --
↳listening-port 12345
```

The client participant can be configured by either using the `ROS_DISCOVERY_SERVER` environment variable (see *ROS_DISCOVERY_SERVER*) or by manually setting it.

Environment Variable

To configure a client participant to communicate over the TCP transport layer with the `ROS_DISCOVERY_SERVER` environment variable, the prefix `TCPv4` needs to be used. The following command can be used to configure the variable to set up a client using TCP communication and connecting to a server on localhost and port 12345.

```
export ROS_DISCOVERY_SERVER=TCPv4:[127.0.0.1]:12345
```

C++

The following snippet can be used to instantiate a client that will try to connect to a server on IP 192.168.10.57 and port 12345, that is, the server instantiated above.

```

eprosima::fastdds::dds::DomainParticipantQos qos = PARTICIPANT_QOS_DEFAULT;

// Configure the current participant as SERVER
qos.wire_protocol().builtin.discovery_config.discoveryProtocol =
    eprosima::fastrtps::rtps::DiscoveryProtocol_t::CLIENT;

// Add custom user transport with TCP port 0 (automatic port assignation)
auto data_transport = std::make_shared<eprosima::fastdds::rtps::TCPv4TransportDescriptor>
↳();
data_transport->add_listener_port(0);
qos.transport().user_transports.push_back(data_transport);

// Define the server locator to be on interface 192.168.10.57 and port 12345
constexpr uint16_t server_port = 12345;
eprosima::fastrtps::rtps::Locator_t server_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(server_locator, "192.168.10.57");
eprosima::fastrtps::rtps::IPLocator::setPhysicalPort(server_locator, server_port);
eprosima::fastrtps::rtps::IPLocator::setLogicalPort(server_locator, server_port);

// Define the server attributes

```

(continues on next page)

(continued from previous page)

```

eprosima::fastrtps::rtps::RemoteServerAttributes remote_server_att;
remote_server_att.metatrafficUnicastLocatorList.push_back(server_locator);

// Set the GUID prefix to identify this server
std::stringstream("44.53.00.5f.45.50.52.4f.53.49.4d.41") >> remote_server_att.
↳ guidPrefix;

// Add the server
qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(remote_server_
↳ att);

```

XML

The following snippet can be used to instantiate a client that will try to connect to a server on IP 192.168.10.57 and port 12345, that is, the server instantiated above.

```

<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <transport_descriptors>
      <transport_descriptor>
        <transport_id>client_tcp_transport</transport_id>
        <type>TCPv4</type>
        <!--
          Set listening port for the transport to 0.
          This automatically assigns a port.
        -->
        <listening_ports>
          <port>0</port>
        </listening_ports>
      </transport_descriptor>
    </transport_descriptors>
    <participant profile_name="TCP_CLIENT" is_default_profile="true">
      <rtps>
        <builtin>
          <discovery_config>
            <discoveryProtocol>CLIENT</discoveryProtocol>
            <discoveryServersList>
              <RemoteServer prefix="44.53.00.5f.45.50.52.4f.53.49.4d.41">
                <!--
                  Server locator specifying where it is listening
                -->
                <metatrafficUnicastLocatorList>
                  <locator>
                    <tcpv4>
                      <address>192.168.10.57</address>
                      <port>12345</port>
                      <physical_port>12345</physical_port>
                    </tcpv4>
                  </locator>
                </metatrafficUnicastLocatorList>
              </RemoteServer>
            </discoveryServersList>

```

(continues on next page)

(continued from previous page)

```
        </discovery_config>
    </builtin>
    <useBuiltinTransports>false</useBuiltinTransports>
    <userTransports>
        <transport_id>client_tcp_transport</transport_id>
    </userTransports>
</rtsp>
</participant>
</profiles>
</dds>
```

7.30.2 Fast DDS over WIFI

The [RTSPS 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 [RTSPS v2.2 standard](#). Refer to [Well Known Ports](#) to learn about these standard port numbers.

If the participant ID is not known, setting *TransportDescriptorInterface* `maxInitialPeersRange` to at least the maximum expected number of *DomainParticipants* will ensure discovery and communication.

C++

```
DomainParticipantQos qos;

// configure an initial peer on host 192.168.10.13.
// The port number corresponds to the well-known port for metatraffic unicast
// on participant ID `1` and domain `0`.
Locator_t initial_peer;
IPLocator::setIPv4(initial_peer, "192.168.10.13");
initial_peer.port = 7412;
qos.wire_protocol().builtin.initialPeersList.push_back(initial_peer);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="initial_peers_example_profile" is_default_profile="true"
  <rtps>
    <builtin>
      <initialPeersList>
        <locator>
          <udp4>
            <address>192.168.10.13</address>
            <port>7412</port>
          </udp4>
        </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 *initial 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">
  <participant profile_name="initial_peers_multicast_avoidance" is_default_profile=
↪"true" >
    <rtps>
      <builtin>
        <!-- Choosing a specific unicast address -->
        <metatrafficUnicastLocatorList>
          <locator>
            <udp4>
              <address>192.168.10.13</address>
              <port>7412</port>
            </udp4>
          </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.

- *UD Pv4 basic example setup*
- *UD Pv4 redundancy example*
- *UD Pv4 persistency example*
- *UD Pv4 partitioning using servers*

UD Pv4 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">
  <participant profile_name="UDP SERVER" is_default_profile="true">
    <rtps>
      <builtin>
        <discovery_config>
          <discoveryProtocol>SERVER</discoveryProtocol>
        </discovery_config>
        <metatrafficUnicastLocatorList>
          <locator>
            <udp4>
              <address>192.168.10.57</address>
              <port>56542</port>
            </udp4>
          </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">
  <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>
                  <udp4>
                    <address>192.168.10.57</address>
                    <port>56542</port>
                  </udp4>
                </locator>
              </metatrafficUnicastLocatorList>
              <metatrafficMulticastLocatorList>
                <locator>
                  <udp4>
                    <address>192.168.10.58</address>
                    <port>24565</port>
                  </udp4>
                </locator>
              </metatrafficMulticastLocatorList>
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
      </builtin>
    </rtps>
  </participant>
</profiles>

```

UDIPv4 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	UDIPv4 address-port
75.63.2D.73.76.72	192.168.10.57:56542
75.63.2D.73.76.72	192.168.10.60:56543

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">
  <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>
            <udp4>
              <address>192.168.10.57</address>
              <port>56542</port>
            </udp4>
          </locator>
        </metatrafficUnicastLocatorList>
      </builtin>
    </rtps>
  </participant>
```

```
7.30. <participant profile_name="UDP SERVER 2">
      <rtps>
```

```
        <prefix>75.63.2D.73.76.72.63.6C.6E.74.2D.32</prefix>
        <builtin>
          <discovery_config>
```

CLIENT**C++**

```
// Define a locator for the first SERVER Participant
Locator_t remote_server_locator_1;
IPLocator::setIPv4(remote_server_locator_1, "192.168.10.57");
remote_server_locator_1.port = 56542;

RemoteServerAttributes remote_server_attr_1;
remote_server_attr_1.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.31");
remote_server_attr_1.metatrafficUnicastLocatorList.push_back(remote_server_locator_1);

// Define a locator for the second SERVER Participant
Locator_t remote_server_locator_2;
IPLocator::setIPv4(remote_server_locator_2, "192.168.10.60");
remote_server_locator_2.port = 56543;

RemoteServerAttributes remote_server_attr_2;
remote_server_attr_2.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.32");
remote_server_attr_2.metatrafficUnicastLocatorList.push_back(remote_server_locator_2);

// Configure the current participant as CLIENT connecting to the SERVERS at the
↳previous locators
DomainParticipantQos client_qos;
client_qos.wire_protocol().builtin.discovery_config.discoveryProtocol =
↳DiscoveryProtocol_t::CLIENT;
client_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↳back(remote_server_attr_1);
client_qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↳back(remote_server_attr_2);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <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>
                  <udp4>
                    <address>192.168.10.57</address>
                    <port>56542</port>
                  </udp4>
                </locator>
              </metatrafficUnicastLocatorList>
            </RemoteServer>
            <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.32">
              <metatrafficUnicastLocatorList>
                <locator>
```

```
          <udp4>
            <address>192.168.10.60</address>
            <port>56543</port>
          </udp4>
        </locator>
```

UDIPv4 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.

UDIPv4 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. 13: Clients cannot see each other due to server isolation

However, it is possible to connect server isolated networks very much as physical networks can be connected through routers:

- *Option 1*: Connecting the clients to several servers, so that the clients belong to several networks.
- *Option 2*: Connecting one server to another, so that the networks are linked together.
- *Option 3*: Create a new server linked to the servers to which the clients are connected.

Options 1 and 2 can only be implemented by modifying QoS values or XML configuration files beforehand. In this regard they match the domain and partition strategy. Option 3, however, can be implemented at runtime, when the isolated networks are already up and running.

Option 1

Connect each client to both servers. This case matches the *redundancy use case* already introduced.

Option 2

Connect one server to the other. This means configuring one of the servers to act as a client of the other.

Consider two servers, each one managing an isolated network:

Nr	Prefix	UDIPv4 address
A	75.63.2D.73.76.7	192.168.10.60:56543
B	75.63.2D.73.76.7	192.168.10.57:56542

In order to communicate both networks we can set server A to act as a client of server B:

C++

```

DomainParticipantQos qos;

// Configure current Participant as SERVER on address 192.168.10.60
Locator_t server_locator;
IPLocator::setIPv4(server_locator, "192.168.10.60");
server_locator.port = 56543;

qos.wire_protocol().builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_
↳t::SERVER;
std::istringstream("75.63.2D.73.76.72.63.6C.6E.74.2D.31") >> qos.wire_protocol().
↳prefix;
qos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(server_locator);

// Add the connection attributes to the remote server.
Locator_t remote_server_locator;
IPLocator::setIPv4(remote_server_locator, "192.168.10.57");
remote_server_locator.port = 56542;

RemoteServerAttributes remote_server_attr;
remote_server_attr.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.32");
remote_server_attr.metatrafficUnicastLocatorList.push_back(remote_server_locator);

qos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_back(remote_
↳server_attr);

```

XML

```

<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <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>
                  <udp4>
                    <address>192.168.10.57</address>
                    <port>56542</port>
                  </udp4>
                </locator>
              </metatrafficUnicastLocatorList>
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
        <metatrafficUnicastLocatorList>
          <locator>
            <udp4>
              <address>192.168.10.60</address>
              <port>56543</port>
            </udp4>
          </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:

S€	Prefix	UDIPv4 address
A	75.63.2D.73.76.7	192.168.10.60:56543
B	75.63.2D.73.76.7	192.168.10.57:56542
C	75.63.2D.73.76.7	192.168.10.54:56541

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">
  <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>
                  <udp4>
                    <address>192.168.10.57</address>
                    <port>56542</port>
                  </udp4>
                </locator>
              </metatrafficUnicastLocatorList>

```

7.30.3 Well Known Network Deployments

It is often the case in industrial deployments, such as production lines, that the entire network topology (hosts, IP addresses, etc.) is known beforehand. Such scenarios are perfect candidates for *Fast DDS* *STATIC Discovery* mechanism, which drastically reduces the middleware setup time (time until all the entities are ready for information exchange), while at the same time limits the connections to those strictly necessary.

Knowing the complete network topology allows to:

- Minimize the PDP meta-traffic and avoid multicast communication with *Peer-to-Peer Participant Discovery Phase*.
- Completely avoid the EDP with *STATIC Endpoint Discovery Phase*.

Peer-to-Peer Participant Discovery Phase

The SIMPLE PDP discovery phase entails the *DomainParticipants* sending periodic PDP announcements over multicast, and answering to the announcements received from remote DomainParticipants. As a result, the number of PDP connections grows quadratically with the number of DomainParticipants, resulting in a large amount of meta traffic on the network.

However, if all DomainParticipants are known beforehand, they can be configured to send their announcements only to the unicast addresses of their peers. This is done by specifying a list of peer addresses, and by disabling the participant multicast announcements. As an additional advantage, with this method only the peers configured on the list are known to the DomainParticipant, allowing to arrange which participant will communicate with which. This reduces the amount of meta traffic if not all the DomainParticipants need to be aware of all the rest of the remote participants present in the network.

Use-case *Fast DDS over WIFI* provides a detailed explanation on how to configure *Fast DDS* for such cases.

STATIC Endpoint Discovery Phase

Users can manually configure which *Publisher* and *Subscriber* match with each other, so they can start sharing user data right away, avoiding the EDP phase.

A complete description of the feature can be found at *STATIC Discovery Settings*. There is also a fully functional helloworld example implementing STATIC EDP in the `examples/cpp/dds/StaticHelloWorldExample` folder.

The following subsections present an example configuration where a Publisher in Topic HelloWorldTopic from DomainParticipant HelloWorldPublisher is matched with a Subscriber from DomainParticipant HelloWorldSubscriber.

Create STATIC discovery XML files

HelloWorldPublisher.xml

```
<staticdiscovery>
  <participant>
    <name>HelloWorldPublisher</name>
    <writer>
      <userId>1</userId>
      <entityID>2</entityID>
      <topicName>HelloWorldTopic</topicName>
      <topicDataType>HelloWorld</topicDataType>
    </writer>
  </participant>
</staticdiscovery>
```

HelloWorldSubscriber.xml

```
<staticdiscovery>
  <participant>
    <name>HelloWorldSubscriber</name>
    <reader>
      <userId>3</userId>
      <entityID>4</entityID>
      <topicName>HelloWorldTopic</topicName>
      <topicDataType>HelloWorld</topicDataType>
    </reader>
  </participant>
</staticdiscovery>
```

Create entities and load STATIC discovery XML files

When creating the entities, the local writer/reader attributes must match those defined in the STATIC discovery XML file loaded by the remote entity.

PUBLISHER

C++

```

// Participant configuration
DomainParticipantQos participant_qos;
participant_qos.name("HelloWorldPublisher");
participant_qos.wire_protocol().builtin.discovery_config.use_SIMPLE_
↳EndpointDiscoveryProtocol = false;
participant_qos.wire_protocol().builtin.discovery_config.use_STATIC_
↳EndpointDiscoveryProtocol = true;
participant_qos.wire_protocol().builtin.discovery_config.static_edp_xml_config(
↳"HelloWorldSubscriber.xml");

// DataWriter configuration
DataWriterQos writer_qos;
writer_qos.endpoint().user_defined_id = 1;
writer_qos.endpoint().entity_id = 2;

// Create the DomainParticipant
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, participant_
↳qos);
if (nullptr == participant)
{
    // Error
    return;
}

// Create the Publisher
Publisher* publisher =
    participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr == publisher)
{
    // Error
    return;
}

// Create the Topic with the appropriate name and data type
std::string topic_name = "HelloWorldTopic";
std::string data_type = "HelloWorld";
Topic* topic =
    participant->create_topic(topic_name, data_type, TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create the DataWriter
DataWriter* writer =
    publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == writer)
{
    // Error
    return;
}
578 return;
}

```

XML

SUBSCRIBER

C++

```

// Participant configuration
DomainParticipantQos participant_qos;
participant_qos.name("HelloWorldSubscriber");
participant_qos.wire_protocol().builtin.discovery_config.use_SIMPLE_
↳EndpointDiscoveryProtocol = false;
participant_qos.wire_protocol().builtin.discovery_config.use_STATIC_
↳EndpointDiscoveryProtocol = true;
participant_qos.wire_protocol().builtin.discovery_config.static_edp_xml_config(
↳"HelloWorldPublisher.xml");

// DataWriter configuration
DataWriterQos writer_qos;
writer_qos.endpoint().user_defined_id = 3;
writer_qos.endpoint().entity_id = 4;

// Create the DomainParticipant
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, participant_
↳qos);
if (nullptr == participant)
{
    // Error
    return;
}

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

// Create the Topic with the appropriate name and data type
std::string topic_name = "HelloWorldTopic";
std::string data_type = "HelloWorld";
Topic* topic =
    participant->create_topic(topic_name, data_type, TOPIC_QOS_DEFAULT);
if (nullptr == topic)
{
    // Error
    return;
}

// Create the DataReader
DataReader* reader =
    subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr == reader)
{
    // Error
    return;
}

```

XML

7.30.4 Large Data Rates

When the amount of data exchanged between a *Publisher* and a *Subscriber* is large, some extra configuration may be required to compensate for side effects on the network and CPU load. This large amount of data can be a result of the data types being large, a high message rate, or a combination of both.

In this scenario, several approaches can be considered depending on the problem:

- For the cases in which the data samples are large (in the order of MB) such as transmitting raw video frames, point clouds, images, etc. between different hosts, TCP based communications may yield better reception rates with lower message loss, specially in the cases where a best effort transport layer is more susceptible to data loss, such as WiFi. To tackle these cases, *Large Data mode and Fast DDS over TCP* documents several ways to configure Fast DDS to communicate over TCP.
- Network packages could be dropped because the transmitted amount of data fills the socket buffer before it can be processed. The solution is to *increase the buffers size*.
- It is also possible to limit the rate at which the Publisher sends data using *Flow Controllers*, in order to limit the effect of message bursts, and avoid to flood the Subscribers faster than they can process the messages.
- On RELIABLE_RELIABILITY_QOS mode, the overall message rate can be affected due to the retransmission of lost packets. Selecting the Heartbeat period allows to tune between increased meta traffic or faster response to lost packets. See *Tuning Heartbeat Period*.
- Also on RELIABLE_RELIABILITY_QOS mode, with high message rates, the history of the *DataWriter* can be filled up, blocking the publication of new messages. A *non-strict reliable mode* can be configured to avoid this blocking, at the cost of potentially losing some messages on some of the Subscribers.

Warning: *eProsima Fast DDS* defines a conservative default message size of 64kB, which roughly corresponds to TCP and UDP payload sizes. If the topic data is bigger, it will automatically be fragmented into several transport packets.

Warning: The loss of a fragment means the loss of the entire message. This has the most impact on BEST_EFFORT_RELIABILITY_QOS mode, where the message loss probability increases with the number of fragments

Increasing socket buffers size

In high rate scenarios or large data scenarios, network packages can be dropped because the transmitted amount of data fills the socket buffer before it can be processed. Using RELIABLE_RELIABILITY_QOS mode, *Fast DDS* will try to recover lost samples, but with the penalty of retransmission. With BEST_EFFORT_RELIABILITY_QOS mode, samples will be definitely lost.

By default *eProsima Fast DDS* creates socket buffers with the system default size. However, these sizes can be modified using the *DomainParticipantQos*, as shown in the example below.

C++

```
DomainParticipantQos participant_qos;

// Increase the sending buffer size
participant_qos.transport().send_socket_buffer_size = 1048576;

// Increase the receiving buffer size
participant_qos.transport().listen_socket_buffer_size = 4194304;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="participant_xml_profile_qos_socketbuffers">
    <rtps>
      <sendSocketBufferSize>1048576</sendSocketBufferSize>
      <listenSocketBufferSize>4194304</listenSocketBufferSize>
    </rtps>
  </participant>
</profiles>
```

Finding out system maximum values

Operating systems set a maximum value for socket buffer sizes. If the buffer sizes are tuned with DomainParticipantQos, the values set cannot exceed the maximum value of the system.

Linux

The maximum buffer size values can be retrieved with the command `sysctl`. For socket buffers used to send data, use the following command:

```
$> sudo sysctl -a | grep net.core.wmem_max
net.core.wmem_max = 1048576
```

For socket buffers used to receive data the command is:

```
$> sudo sysctl -a | grep net.core.rmem_max
net.core.rmem_max = 4194304
```

However, these maximum values are also configurable and can be increased if needed. The following command increases the maximum buffer size of sending sockets:

```
$> sudo sysctl -w net.core.wmem_max=12582912
```

For receiving sockets, the command is:

```
$> sudo sysctl -w net.core.rmem_max=12582912
```

Windows

The following command changes the maximum buffer size of sending sockets:

```
C:\> reg add HKLM\SYSTEM\CurrentControlSet\services\AFD\Parameters /v DefaultSendWindow /  
↪t REG_DWORD /d 12582912
```

For receiving sockets, the command is:

```
C:\> reg add HKLM\SYSTEM\CurrentControlSet\services\AFD\Parameters /v  
↪DefaultReceiveWindow /t REG_DWORD /d 12582912
```

Increasing the Transmit Queue Length of an interface (Linux only)

The Transmit Queue Length (`txqueuelen`) is a TCP/UDP/IP stack network interface value. This value sets the number of packets allowed per kernel transmit queue of a network interface device. By default, the `txqueuelen` value for Ethernet interfaces is set to 1000 in Linux. This value is adequate for most Gigabit network devices. However, in some specific cases, the `txqueuelen` setting should be increased to avoid overflows that drop packets. Similarly, choosing a value that is too large can cause added overhead resulting in higher network latencies.

Note that this information only applies to the *sending* side, and not the *receiving* side. Also increasing the `txqueuelen` should go together with increasing the buffer sizes of the UDP and/or TCP buffers. (this must be applied for both the *sending* and *receiving* sides).

The settings for a specific network adapter can be viewed using the one of the following commands:

`ip`

```
ip link show ${interface}
```

`ifconfig`

```
ifconfig ${interface}
```

This will display the configuration of the adapter, and among the parameters the `txqueuelen`. This parameter can be a value between 1000 and 20000.

Important: If the `ip` command is used, the Transmit Queue Length parameter is called `qlen`.

The `txqueuelen` can be modified for the current session using either the `ifconfig` or `ip` commands. However, take into account that after rebooting the default values will be configured again.

`ip`

```
ip link set txqueuelen ${value} dev ${interface}
```

`ifconfig`

```
ifconfig ${interface} txqueuelen ${size}
```

Flow Controllers

eProsima Fast DDS provides a mechanism to limit the rate at which the data is sent by a `DataWriter`. These controllers should be registered on the creation of the `DomainParticipant` using *FlowControllersQos*, and then referenced on the creation of the `DataWriter` using *PublishModeQosPolicy*.

A new thread is spawned the first time a flow controller is referenced by an asynchronous `DataWriter`. This thread will be responsible for arbitrating the network output of the samples being transmitted by all the `DataWriters` referencing the same flow controller.

Flow controllers should be given a name so they can later on be referenced by the `DataWriters`. A default, unlimited, FIFO flow controller is always available with name `FASTDDS_FLOW_CONTROLLER_DEFAULT`.

Scheduling policy

There are different kinds of flow controllers, depending on the scheduling policy used. All of them will limit the number of bytes sent to the network to no more than `max_bytes_per_period` bytes during `period_ms` milliseconds. They only differ in the way they decide the order in which the samples are sent.

- `FIFO` will output samples on a first come, first served order.
- `ROUND_ROBIN` will output one sample from each `DataWriter` in circular order.
- `HIGH_PRIORITY` will output samples from `DataWriters` with the highest priority first. The priority of a `DataWriter` is configured using property `fastdds.sfc.priority`. Allowed values are from -10 (highest priority) to 10 (lowest priority). If the property is not present, it will be set to the lowest priority. Samples for `DataWriters` with the same priority are handled with `FIFO` order.
- `PRIORITY_WITH_RESERVATION` works as the previous one, but allows the `DataWriters` to reserve part of the output bandwidth. This is done with the property `fastdds.sfc.bandwidth_reservation`. Allowed values are from 0 to 100, and express a percentage of the total flow controller limit. If the property is not present, it will be set to 0 (no bandwidth is reserved for the `DataWriter`). After the reserved bandwidth has been consumed, the rest of the samples will be handled with the rules of `HIGH_PRIORITY`.

Example configuration

C++

```
// Limit to 300kb per second.
static const char* flow_controller_name = "example_flow_controller";
auto flow_control_300k_per_sec = std::make_shared
    <eprosima::fastdds::rtps::FlowControllerDescriptor>();
flow_control_300k_per_sec->name = flow_controller_name;
flow_control_300k_per_sec->scheduler =
    eprosima::fastdds::rtps::FlowControllerSchedulerPolicy::FIFO;
flow_control_300k_per_sec->max_bytes_per_period = 300 * 1000;
flow_control_300k_per_sec->period_ms = 1000;

// [OPTIONAL] Configure sender thread settings
flow_control_300k_per_sec->sender_thread = eprosima::fastdds::rtps::ThreadSettings{-1, 0,
    0, -1};

// Register flow controller on participant
DomainParticipantQos participant_qos;
```

(continues on next page)

(continued from previous page)

```
participant_qos.flow_controllers().push_back(flow_control_300k_per_sec);

// .... create participant and publisher

// Link writer to the registered flow controller.
// Note that ASYNCHRONOUS_PUBLISH_MODE must be used
DataWriterQos qos;
qos.publish_mode().kind = ASYNCHRONOUS_PUBLISH_MODE;
qos.publish_mode().flow_controller_name = flow_controller_name;
```

XML

There is currently no way of configuring flow controllers with XML. This will be added in future releases of the product.

Warning: Specifying a flow controller with a size smaller than the transport buffer size can cause the messages to never be sent.

Tuning Heartbeat Period

On `RELIABLE_RELIABILITY_QOS` (*ReliabilityQosPolicy*), RTPS protocol can detect which messages have been lost and retransmit them. This mechanism is based on meta-traffic information exchanged between DataWriters and DataReaders, namely, Heartbeat and Ack/Nack messages.

A smaller Heartbeat period increases the CPU and network overhead, but speeds up the system response when a piece of data is lost. Therefore, users can customize the Heartbeat period to match their needs. This can be done with the `DataWriterQos`.

```
DataWriterQos qos;
qos.reliable_writer_qos().times.heartbeatPeriod.seconds = 0;
qos.reliable_writer_qos().times.heartbeatPeriod.nanosec = 500000000;    //500 ms
```

Using Non-strict Reliability

When *HistoryQosPolicyKind* is set as `KEEP_ALL_HISTORY_QOS`, all samples have to be received (and acknowledged) by all subscribers before they can be overridden by the DataWriter. If the message rate is high and the network is not reliable (i.e., lots of packets get lost), the history of the DataWriter can be filled up, blocking the publication of new messages until any of the old messages is acknowledged by all subscribers.

If this strictness is not needed, *HistoryQosPolicyKind* can be set as `KEEP_LAST_HISTORY_QOS`. In this case, when the history of the DataWriter is full, the oldest message that has not been fully acknowledged yet is overridden with the new one. If any subscriber did not receive the discarded message, the publisher will send a GAP message to inform the subscriber that the message is lost forever.

Practical Examples

Example: Sending a large file

Consider the following scenario:

- A Publisher needs to send a file with a size of 9.9 MB.
- The Publisher and Subscriber are connected through a network with a bandwidth of 100 MB/s

With a fragment size of 64 kB, the Publisher has to send about 1100 fragments to send the whole file. A possible configuration for this scenario could be:

- Using `RELIABLE_RELIABILITY_QOS`, since a losing a single fragment would mean the loss of the complete file.
- Decreasing the heartbeat period, in order to increase the reactivity of the Publisher.
- Limiting the data rate using a *Flow Controller*, to avoid this transmission cannibalizing the whole bandwidth. A reasonable rate for this application could be 5 MB/s, which represents only 5% of the total bandwidth.

Note: Using *Shared Memory Transport* the only limit to the fragment size is the available memory. Therefore, all fragmentation can be avoided in SHM by increasing the size of the shared buffers.

Example: Video streaming

In this scenario, the application transmits a video stream between a Publisher and a Subscriber, at 50 fps. In real-time audio or video transmissions, it is usually preferred to have a high stable datarate feed, even at the cost of losing some samples. Losing one or two samples per second at 50 fps is more acceptable than freezing the video waiting for the retransmission of lost samples. Therefore, in this case `BEST_EFFORT_RELIABILITY_QOS` can be appropriate.

7.30.5 Topics with many subscribers

By default, every time a *DataWriter* publishes a data change on a *Topic*, it sends a unicast message for every *DataReader* that is subscribed to the Topic. If there are several DataReaders subscribed, it is recommendable to use multicast instead of unicast. By doing so, only one network package will be sent for each sample. This will improve both CPU and network usage.

This solution can be implemented with *UDP Transport* or *Shared Memory Transport* (SHM). SHM transport is multicast by default, but is only available between DataWriters and DataReaders on the same machine. UDP transport needs some extra configuration. The example below shows how to set a *DataReaderQos* to configure a DataReader to use a multicast transport on UDP. More information about configuring local and remote locators on endpoints can be found in *RTPSEndpointQos*.

Note: Multicast over UDP can be problematic on some scenarios, mainly WiFi and complex networks with multiple network links.

C++

```
DataReaderQos qos;

// Add new multicast locator with IP 239.255.0.4 and port 7900
eprosima::fastdds::rtps::Locator_t new_multicast_locator;
eprosima::fastdds::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">
  <data_reader profile_name="reader_xml_conf_multicast_locators_profile">
    <multicastLocatorList>
      <locator>
        <udp4>
          <address>239.255.0.4</address>
          <port>7900</port>
        </udp4>
      </locator>
    </multicastLocatorList>
  </data_reader>
</profiles>
```

7.30.6 Real-time behavior

Real-time applications have very tight constraints on data processing times. In order to comply with these constraints, *Fast DDS* can be configured to guarantee responses within a specified time. This is achieved with the following restraints:

- Allocating all the required memory during entity initialization, so that all the data processing tasks are heap allocation free (see [Tuning allocations](#)).
- Returning from blocking functions if the provided timeout is reached (see [Non-blocking calls](#)).

This section explains how to configure *Fast DDS* to achieve this behavior.

Tuning allocations

Allocating and deallocating memory implies some non-deterministic time consuming operations. Therefore, most real-time systems need to operate in a way that all dynamic memory is allocated during the application initialization, avoiding memory management operations in the main loop.

If users provide maximum sizes for the data and collections that *Fast DDS* keeps internally, memory for these data and collections can be preallocated during entity initialization. In order to choose the correct size values, users must be aware of the topology of the whole domain. Specifically, the number of [DomainParticipants](#), [DataWriters](#), and [DataReaders](#) must be known when setting their configuration.

The following sections describe how to configure allocations to be done during the initialization of the entities. Although some examples are provided on each section as reference, there is also a [complete example use case](#).

Parameters on the participant

Every `DomainParticipant` holds an internal collection with information about every local and remote peer `DomainParticipant`s 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 `DomainParticipant`s.
- `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">
  <participant profile_name="participant_profile_qos_entity_resource_limit">
    <rtps>
      <allocation>
        <!-- Limit to 3 participants -->
        <total_participants>
          <initial>3</initial>
          <maximum>3</maximum>
          <increment>0</increment>
        </total_participants>

        <!-- Limit to 3 readers per participant -->
        <total_readers>
          <initial>3</initial>
          <maximum>3</maximum>
          <increment>0</increment>
        </total_readers>

        <!-- Limit to 1 writer per participant -->
        <total_writers>
          <initial>1</initial>
          <maximum>1</maximum>
          <increment>0</increment>
        </total_writers>
      </allocation>
    </rtps>
  </participant>
</profiles>
```


Warning: Configuring a collection as fixed in size effectively limits the number of peer entities that can be discovered. Once the configured limit is reached, any new entity will be ignored. In the given example, if a fourth peer `DomainParticipant` appears, it will not be discovered, as the collection of discovered `DomainParticipants` is already full.

Limiting the size of custom parameters

`data_limits` inside *ParticipantResourceLimitsQos* provides three data members to configure the allocation behavior of custom parameters:

- `max_user_data` limits the size of *UserDataQosPolicy* to the given number of octets.
- `max_properties` limits the size of *PartitionQosPolicy* to the given number of octets.
- `max_partitions` limits the size of *PropertyPolicyQos* to the given number of octets.

If these sizes are configured to something different than zero, enough memory will be allocated for them for each participant and endpoint. A value of zero implies no size limitation, and memory will be dynamically allocated as needed. By default, a full dynamic behavior is used.

`content_filter` inside *ParticipantResourceLimitsQos* provides members to configure the allocation behavior of content filter discovery information:

- `expression_initial_size` sets the preallocated size of the filter expression.
- `expression_parameters` controls the allocation behavior for the list of expression parameters. Refer to *ResourceLimitedContainerConfig* for a complete description of the alternatives. Receiving information about a content filter with more parameters than the maximum configured here, will make the filtering happen on the reader side.

C++

```
DomainParticipantQos qos;

// Fix the size of the complete user data field to 256 octets
qos.allocation().data_limits.max_user_data = 256u;
// Fix the size of the complete partitions field to 256 octets
qos.allocation().data_limits.max_partitions = 256u;
// Fix the size of the complete properties field to 512 octets
qos.allocation().data_limits.max_properties = 512u;
// Set the preallocated filter expression size to 512 characters
qos.allocation().content_filter.expression_initial_size = 512u;
// Set the maximum number of expression parameters to 4 and its allocation_
↪configuration to fixed size
qos.allocation().content_filter.expression_parameters =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(4u);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="participant_profile_qos_parameter_resource_limit">
    <rtps>
      <allocation>
        <max_partitions>256</max_partitions>
        <max_user_data>256</max_user_data>
        <max_properties>512</max_properties>

        <!-- content_filter cannot be configured using XML (yet) -->
      </allocation>
    </rtps>
  </participant>
</profiles>
```

Warning: If the data fields announced by the remote peer do not fit on the preallocated memory, an error will be triggered during the processing of the announcement message. This usually means that the discovery messages of a remote peer with too large data fields will be discarded, i.e., peers with too large data fields will not be discovered.

Parameters on the DataWriter

Every DataWriter holds internal collections with information about every DataReader to which it matches. By default, these collections are fully dynamic, meaning that new memory is allocated when a new DataReader is matched. However, *DataWriterQos* has a data member `writer_resource_limits()`, of type *WriterResourceLimitsQos*, that allows configuring the memory allocation behavior on the DataWriter.

WriterResourceLimitsQos provides data members `matched_subscriber_allocation` and `reader_filters_allocation` of type *ResourceLimitedContainerConfig* that allow configuring the maximum expected size of the collection of matched DataReader, and the collection of writer side content filters, so they

can be preallocated during the initialization of the DataWriter, as shown in the example below. Please, refer to [ResourceLimitedContainerConfig](#) for a complete description of additional configuration alternatives given by these data members.

C++

```
DataWriterQos qos;

// Fix the size of matched DataReaders to 3
// This will effectively preallocate the memory during initialization
qos.writer_resource_limits().matched_subscriber_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    configuration(3u);
// Fix the size of writer side content filters to 1
// This will effectively preallocate the memory during initialization
qos.writer_resource_limits().reader_filters_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    configuration(1u);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <data_writer profile_name="writer_profile_qos_resource_limit">
    <!-- Limit to 3 matching readers -->
    <matchedSubscribersAllocation>
      <initial>3</initial>
      <maximum>3</maximum>
      <increment>0</increment>
    </matchedSubscribersAllocation>

    <!-- reader_filters_allocation cannot be configured using XML (yet) -->
  </data_writer>
</profiles>
```

Warning: Configuring the collection of matched DataReaders as fixed in size effectively limits the number of DataReaders to be matched. Once the configured limit is reached, any new DataReader will be ignored. In the given example, if a fourth (potentially matching) DataReader appears, it will not be matched, as the collection is already full.

Parameters on the DataReader

Every DataReader holds an internal collection with information about every *ReaderResourceLimitsQos* to which it matches. By default, this collection is fully dynamic, meaning that new memory is allocated when a new DataWriter is matched. However, *DataReaderQos* has a data member `reader_resource_limits()`, of type *ReaderResourceLimitsQos*, that allows configuring the memory allocation behavior on the DataReader.

ReaderResourceLimitsQos provides a data member `matched_publisher_allocation` of type *ResourceLimitedContainerConfig* that allows configuring the maximum expected size of the collection of matched DataWriters, so that it can be preallocated during the initialization of the DataReader, as shown in the example below. Please, refer to *ResourceLimitedContainerConfig* for a complete description of additional configuration alternatives given by this data member.

C++

```
DataReaderQos qos;

// Fix the size of matched DataWriters to 1
// This will effectively preallocate the memory during initialization
qos.reader_resource_limits().matched_publisher_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    configuration(1u);
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <data_reader profile_name="reader_profile_qos_resource_limit">
    <!-- Limit to 1 matching writer -->
    <matchedPublishersAllocation>
      <initial>1</initial>
      <maximum>1</maximum>
      <increment>0</increment>
    </matchedPublishersAllocation>
  </data_reader>
</profiles>
```

Warning: Configuring the collection of matched DataWriters as fixed in size effectively limits the number of DataWriters to be matched. Once the configured limit is reached, any new DataWriter will be ignored. In the given example, if a fourth (potentially matching) DataWriter appears, it will not be matched, as the collection is already full.

Full example

Given a system with the following topology:

Table 14: **Allocation tuning example topology**

Participant P1	Participant P2	Participant P3
Topic 1 publisher	Topic 1 subscriber	Topic 2 subscriber
Topic 1 subscriber		Topic 2 publisher
Topic 1 subscriber		Topic 2 subscriber

- The total number of DomainParticipants is 3.
- The maximum number of DataWriters per DomainParticipant is 1
- The maximum number of DataReaders per DomainParticipant is 2.
- The DataWriter for topic 1 matches with 3 DataReaders.
- The DataWriter for topic 2 matches with 2 DataReaders.
- All the DataReaders match exactly with 1 DataWriter.

We will assume that content filtering is not being used, and will also limit the size of the parameters:

- Maximum *PartitionQosPolicy* size: 256
- Maximum *UserDataQosPolicy* size: 256
- Maximum *PropertyPolicyQos* size: 512

The following piece of code shows the set of parameters needed for the use case depicted in this example.

C++

```

// DomainParticipant configuration
////////////////////////////////////
DomainParticipantQos participant_qos;

// We know we have 3 participants on the domain
participant_qos.allocation().participants =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(3u);
// We know we have at most 2 readers on each participant
participant_qos.allocation().readers =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(2u);
// We know we have at most 1 writer on each participant
participant_qos.allocation().writers =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(1u);

// We know the maximum size of partition data
participant_qos.allocation().data_limits.max_partitions = 256u;
// We know the maximum size of user data
participant_qos.allocation().data_limits.max_user_data = 256u;
// We know the maximum size of properties data
participant_qos.allocation().data_limits.max_properties = 512u;

// Content filtering is not being used
participant_qos.allocation().content_filter.expression_initial_size = 0u;
participant_qos.allocation().content_filter.expression_parameters =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(0u);

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

// We know we will only have three matching subscribers, and no content filters
writer1_qos.writer_resource_limits().matched_subscriber_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(3u);
writer1_qos.writer_resource_limits().reader_filters_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(0u);

// DataWriter configuration for Topic 2
////////////////////////////////////
DataWriterQos writer2_qos;

// We know we will only have two matching subscribers
writer2_qos.writer_resource_limits().matched_subscriber_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(2u);
writer2_qos.writer_resource_limits().reader_filters_allocation =
    eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
    ↪configuration(0u);

```

```

// DataReader configuration for both Topics
////////////////////////////////////

```

Non-blocking calls

Note: As OSX does not support necessary POSIX Real-time features, this feature is not fully supported on OSX. In that case, the feature is limited by the implementation of `std::timed_mutex` and `std::condition_variable_any`.

Several functions on the *Fast DDS API* can be blocked for an undefined period of time when operations compete for the control of a resource. The blocked function cannot continue until the operation that gained the control finishes, thus blocking the calling thread.

Real-time applications need a predictable behavior, including a predictable maximum time since a function is called until it returns control. In order to comply with this restriction, *Fast DDS* can be configured to limit the maximum blocking time of these functions. If the blocking time limit is exceeded, the requested operation is aborted and function terminated, returning the control to the caller.

This configuration needs two steps:

- Set the CMake option `-DSTRICT_REALTIME=ON` during the compilation of the application.
- Configure the maximum blocking times for the functions.

Table 15: **Fast RTPS non-blocking API**

Method	Configuration attribute	Default value
<code>DataWriter::write()</code>	<code>reliability().max_blocking_time</code> <i>DataWriterQos</i> .	on 100 millisec- onds.
<code>DataReader::take_next_sample()</code>	<code>reliability().max_blocking_time</code> <i>DataReaderQos</i> .	on 100 millisec- onds.
<code>DataReader::read_next_sample()</code>	<code>reliability().max_blocking_time</code> <i>DataReaderQos</i> .	on 100 millisec- onds.
<code>DataReader::wait_for_unread_mes</code>	The method accepts an argument with the maximum blocking time.	

7.30.7 Reduce memory usage

A great number of modern systems have tight constraints on available memory, making the reduction of memory usage to a minimum critical. Reducing memory consumption of a *Fast DDS* application can be achieved through various approaches, mainly through architectural restructuring of the application, but also by limiting the resources the middleware utilizes, and by avoiding static allocations.

Limiting Resources

The *ResourceLimitsQosPolicy* controls the resources that the service can use in order to meet the requirements imposed. It limits the amount of allocated memory per *DataWriter* or *DataReader*, as per the following parameters:

- `max_samples`: Configures the maximum number of samples that the *DataWriter* or *DataReader* can manage across all the instances associated with it, i.e. it represents the maximum samples that the middleware can store for a *DataReader* or *DataWriter*.
- `max_instances`: Configures the maximum number of instances that the *DataWriter* or *DataReader* can manage.
- `max_samples_per_instance`: Controls the maximum number of samples within an instance that the *DataWriter* or *DataReader* can manage.
- `allocated_samples`: States the number of samples that will be allocated on initialization.

All these parameters may be lowered as much as needed to reduce memory consumption, limit the resources to the application's needs. Below is an example of a configuration for the minimum resource limits possible.

Warning:

- The value of `max_samples` must be higher or equal to the value of `max_samples_per_instance`.
- The value established for the *HistoryQosPolicy* `depth` must be lower or equal to the value stated for `max_samples_per_instance`.

C++

```
ResourceLimitsQosPolicy resource_limits;

// The ResourceLimitsQosPolicy is default constructed with max_samples = 5000
// Change max_samples to the minimum
resource_limits.max_samples = 1;

// The ResourceLimitsQosPolicy is default constructed with max_instances = 10
// Change max_instances to the minimum
resource_limits.max_instances = 1;

// The ResourceLimitsQosPolicy is default constructed with max_samples_per_instance = 400
// Change max_samples_per_instance to the minimum
resource_limits.max_samples_per_instance = 1;

// The ResourceLimitsQosPolicy is default constructed with allocated_samples = 100
// No allocated samples
resource_limits.allocated_samples = 0;
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <data_writer profile_name="data_writer_min_samples">
    <topic>
      <historyQos>
        <kind>KEEP_LAST</kind>
        <depth>1</depth>
      </historyQos>
      <resourceLimitsQos>
        <max_samples>1</max_samples>
        <max_instances>1</max_instances>
        <max_samples_per_instance>1</max_samples_per_instance>
        <allocated_samples>0</allocated_samples>
      </resourceLimitsQos>
    </topic>
  </data_writer>

  <data_reader profile_name="data_reader_min_samples">
    <topic>
      <historyQos>
        <kind>KEEP_LAST</kind>
        <depth>1</depth>
      </historyQos>
      <resourceLimitsQos>
        <max_samples>1</max_samples>
        <max_instances>1</max_instances>
        <max_samples_per_instance>1</max_samples_per_instance>
        <allocated_samples>0</allocated_samples>
      </resourceLimitsQos>
    </topic>
  </data_reader>
</profiles>
```

Set Dynamic Allocation

By default *MemoryManagementPolicy* is set to `PREALLOCATED_WITH_REALLOC_MEMORY_MODE`, meaning that the amount of memory required by the configured *ResourceLimitsQosPolicy* will be allocated at initialization. If some more memory has to be allocated at run time, it is reallocated.

Using the dynamic settings of the *RTPSEndpointQos* will prevent unnecessary allocations. Lowest footprint is achieved with `DYNAMIC_RESERVE_MEMORY_MODE` at the cost of higher allocation counts, in this mode memory is allocated when needed and freed as soon as it stops being used. For higher determinism at a small memory cost the `DYNAMIC_REUSABLE_MEMORY_MODE` option is available, this option is similar but once more memory is allocated it is not freed and is reused for future messages.

C++

```
RTPSEndpointQos endpoint;
endpoint.history_memory_policy = eprosima::fastrtps::rtps::DYNAMIC_REUSABLE_MEMORY_
↪MODE;
```

XML

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

7.30.8 Zero-Copy communication

This section explains how to configure a Zero-Copy communication in *Fast DDS*. The Zero-Copy communication allows the transmission of data between applications without copying data in memory, saving time and resources. In order to achieve this, it uses Data-sharing delivery between the *DataWriter* and the *DataReader*, and data buffer loans between the application and *Fast DDS*.

- *Overview*
- *Getting started*
- *Writing and reading in Zero-Copy transfers*
- *Caveats*
- *Constraints*
- *Next steps*

Overview

Data-sharing delivery provides a communication channel between a DataWriter and a DataReader using shared memory. Therefore, it does not require copying the sample data to transmit it.

DataWriter sample loaning is a *Fast DDS* extension that allows the application to borrow a buffer for a sample in the publishing DataWriter. The sample can be constructed directly on this buffer, eliminating the need to copy it to the DataWriter afterwards. This prevents the copying of the data between the publishing application and the DataWriter. If Data-sharing delivery is used, the loaned data buffer will be in the shared memory itself.

Reading the data on the subscriber side can also be done with *loans from the DataReader*. The application gets the received samples as a reference to the receive queue itself. This prevents the copying of the data from the DataReader to the receiving application. Again, if Data-sharing delivery is used, the loaned data will be in the shared memory, and will indeed be the same memory buffer used in the DataWriter history.

Combining these three features, we can achieve Zero-Copy communication between the publishing application and the subscribing application.

Getting started

To enable Zero-Copy perform the following steps:

1. Define a plain and bounded type in an IDL file and generate the corresponding source code for further processing with the *Fast DDS-Gen* tool.

```
struct LoanableHelloWorld
{
    unsigned long index;
    char message[256];
};
```

2. On the DataWriter side:
 - a) Create a DataWriter for the previous type. Make sure that the DataWriter does not have DataSharing disabled.
 - b) Get a loan on a sample using `loan_sample()`.
 - c) Write the sample using `write()`.
3. On the DataReader side:
 - a) Create a DataReader for the previous type. Make sure that the DataReader does not have DataSharing disabled.
 - b) Take/read samples using the available functions in the DataReader. Please refer to section *Loaning and Returning Data and SampleInfo Sequences* for further detail on how to access to loans of the received data.
 - c) Return the loaned samples using `DataReader::return_loan()`.

Writing and reading in Zero-Copy transfers

The following is an example of how to publish and receive samples with DataWriters and DataReaders respectively that implement Zero-Copy.

DataWriter

When the DataWriter is created, *Fast DDS* will pre-allocate a pool of `max_samples + extra_samples` samples that reside in a shared memory mapped file. This pool will be used to loan samples when the `loan_sample()` function is called.

An application example of a DataWriter that supports Zero-Copy using the *Fast DDS* library is presented below. There are several points to note in the following code:

- Not disabling the *DataSharingQosPolicy*. AUTO kind automatically enables Zero-Copy when possible.
- The use of the `loan_sample()` function to access and modify data samples.
- The writing of data samples.

```
// CREATE THE PARTICIPANT
DomainParticipantQos pqos;
pqos.name("Participant_pub");
DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
    participant(0, pqos);

// REGISTER THE TYPE
TypeSupport type(new LoanableHelloWorldPubSubType());
type.register_type(participant);

// CREATE THE PUBLISHER
Publisher* publisher = participant->create_publisher(PUBLISHER_QOS_DEFAULT, nullptr);

// CREATE THE TOPIC
Topic* topic = participant->create_topic(
    "LoanableHelloWorldTopic",
    type.get_type_name(),
    TOPIC_QOS_DEFAULT);

// CREATE THE WRITER
DataWriterQos wqos = publisher->get_default_datawriter_qos();
wqos.history().depth = 10;
wqos.durability().kind = TRANSIENT_LOCAL_DURABILITY_QOS;
// DataSharingQosPolicy has to be set to AUTO (the default) or ON to enable Zero-Copy
wqos.data_sharing().on("shared_directory");

DataWriter* writer = publisher->create_datawriter(topic, wqos);

std::cout << "LoanableHelloWorld DataWriter created." << std::endl;

int msgsent = 0;
void* sample = nullptr;
// Always call loan_sample() before writing a new sample.
// This function will provide the user with a pointer to an internal buffer where the
```

(continues on next page)

(continued from previous page)

```

↪data type can be
// prepared for sending.
if (RETCODE_OK == writer->loan_sample(sample))
{
    // Modify the sample data
    LoanableHelloWorld* data = static_cast<LoanableHelloWorld*>(sample);
    data->index() = msgsent + 1;
    memcpy(data->message().data(), "LoanableHelloWorld ", 20);

    std::cout << "Sending sample (count=" << msgsent
                << ") at address " << &data << std::endl
                << "   index=" << data->index() << std::endl
                << "   message=" << data->message().data() << std::endl;

    // Write the sample.
    // After this function returns, the middleware owns the sample.
    writer->write(sample);
}

```

DataReader

The following is an application example of a DataReader that supports Zero-Copy using the *Fast DDS* library. As shown in this code snippet, the configuration in the DataReader is similar to the DataWriter. Be sure not to disable the *DataSharingQosPolicy*. AUTO kind automatically enables Zero-Copy when possible.

```

// CREATE THE PARTICIPANT
DomainParticipantQos pqos;
pqos.name("Participant_sub");
DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
↪participant(0, pqos);

// REGISTER THE TYPE
TypeSupport type(new LoanableHelloWorldPubSubType());
type.register_type(participant);

// CREATE THE SUBSCRIBER
Subscriber* subscriber = participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT, nullptr);

// CREATE THE TOPIC
Topic* topic = participant->create_topic(
    "LoanableHelloWorldTopic",
    type.get_type_name(),
    TOPIC_QOS_DEFAULT);

// CREATE THE READER
DataReaderQos rqos = subscriber->get_default_datareader_qos();
rqos.history().depth = 10;
rqos.reliability().kind = RELIABLE_RELIABILITY_QOS;
rqos.durability().kind = TRANSIENT_LOCAL_DURABILITY_QOS;
// DataSharingQosPolicy has to be set to AUTO (the default) or ON to enable Zero-Copy
rqos.data_sharing().automatic();

```

(continues on next page)

(continued from previous page)

```
DataReader* reader = subscriber->create_datareader(topic, rqos, &datareader_listener);
```

Finally, the code snippet below implements the `on_data_available()` *DataReaderListener* callback. The key points to be noted in this function are:

- The declaration and handling of `LoanableSequence`.
- Checking `DataReader::is_sample_valid()` for verifying that the sample was not replaced. Refer to *DataReader and DataWriter history coupling* for further information.
- The use of the `DataReader::return_loan()` function to indicate to the `DataReader` that the application has finished accessing the sequence.

```
void on_data_available(
    eprosima::fastdds::dds::DataReader* reader) override
{
    // Declare a LoanableSequence for a data type
    FASTDDS_SEQUENCE(DataSeq, LoanableHelloWorld);

    DataSeq data;
    SampleInfoSeq infos;
    // Access to the collection of data-samples and its corresponding collection of
    ↪ SampleInfo structures
    while (RETCODE_OK == reader->take(data, infos))
    {
        // Iterate over each LoanableCollection in the SampleInfo sequence
        for (LoanableCollection::size_type i = 0; i < infos.length(); ++i)
        {
            // Check whether the DataSample contains data or is only used to communicate
            ↪ of a
            // change in the instance
            if (infos[i].valid_data)
            {
                // Print the data.
                const LoanableHelloWorld& sample = data[i];

                ++samples;
                std::cout << "Sample received (count=" << samples
                    << ") at address " << &sample
                    << (reader->is_sample_valid(&sample,
                        &infos[i]) ? " is valid" : " was replaced" ) << std::endl
                    << " index=" << sample.index() << std::endl
                    << " message=" << sample.message().data() << std::endl;
            }
        }
        // Indicate to the DataReader that the application is done accessing the
        ↪ collection of
        // data values and SampleInfo, obtained by some earlier invocation of read or
        ↪ take on the
        // DataReader.
        reader->return_loan(data, infos);
    }
}
```

Caveats

- After calling `write()`, *Fast DDS* takes ownership of the sample and therefore it is no longer safe to make changes to that sample.
- If function `loan_sample()` is called first and the sample is never written, it is necessary to use function `discard_loan()` to return the sample to the DataWriter. If this is not done, the subsequent calls to `loan_sample()` may fail if DataWriter has no more `extra_samples` to loan.
- The current maximum supported sample size is the maximum value of an `uint32_t`.

Constraints

Although Zero-Copy can be used for one or several *Fast DDS* application processes running on the same machine, it has some constraints:

- Only plain types are supported.

A plain type is a type whose CDR representation matches its in-memory representation. This requirement avoids the copy between the CDR buffer and the user buffer because the data representation is the same. Consequently, only *primitive types* (except `string`), arrays of these primitive types, and structures with FINAL *extensibility* and members of these primitive types, are considered to be plain (Fast DDS also provides an API to check if a defined type is plain: `TypeSupport::is_plain()`).

- Constraints for *datasharing delivery* also apply.

Note: Zero-Copy transfer support for non-plain types may be implemented in future releases of *Fast DDS*.

Next steps

The *eProsima Fast DDS* Github repository contains the complete example discussed in this section, as well as multiple other examples for different use cases. The example implementing Zero-Copy transfers can be found [here](#).

7.30.9 Unique network flows

This section explains which APIs should be used on Fast DDS in order to have unique network flows on specific topics.

- *Background*
- *Identifying a flow*
- *Requesting unique flows*
- *Example*

Background

IP networking is the pre-dominant inter-networking technology used nowadays. Ethernet, WiFi, 4G/5G telecommunication, all of them rely on IP networking.

Streams of IP packets from a given source to destination are called *packet flows* or simply *flows*. The network QoS of a flow can be configured when using certain networking equipment (routers, switches). Such pieces of equipment typically support 3GPP/5QI protocols to assign certain Network QoS parameters to specific flows. Requesting a specific Network QoS is usually done on the endpoint sending the data, as it is the one that usually has complete information about the network flow.

Applications may need to use specific Network QoS parameters on different topics.

This means an application should be able to:

- a) Identify the flows being used in the communications, so they can correctly configure the networking equipment.
- b) Use specific flows on selected topics.

Identifying a flow

The *5-tuple* is a traditional unique identifier for flows on 3GPP enabled equipment. The 5-tuple consists of five parameters: source IP address, source port, destination IP address, destination port, and the transport protocol (example, TCP/UDP).

Definitions

Network flow: A tuple of networking resources selected by the middleware for transmission of messages from a DataWriter to a DataReader, namely:

- Transport protocol: UDP or TCP
- Transport port
- Internet protocol: IPv4 or IPv6
- IP address

Network Flow Endpoint (NFE): The portion of a network flow specific to the DataWriter or the DataReader. In other words, each network flow has two NFEs; one for the DataWriter, and the other for the DataReader.

APIs

Fast DDS provides the APIs needed to get the list of NFEs used by a given DataWriter or a DataReader.

- On the DataWriter, `get_sending_locators()` allows the application to obtain the list of locators from which the writer may send data.
- On the DataReader, `get_listening_locators()` allows the application to obtain the list of locators on which the reader is listening.

Requesting unique flows

A unique flow can be created by ensuring that at least one of the two NFEs are unique. On Fast DDS, there are two ways to select unique listening locators on the DataReader:

- The application can specify on which locators the DataReader should be listening. This is done using *RTPSEndpointQos* on the *DataReaderQos*. In this case it is the responsibility of the application to ensure the uniqueness of the locators used.
- The application can request the reader to be created with unique listening locators. This is done using a *PropertyPolicyQos* including the property "fastdds.unique_network_flows". In this case, the reader will listen on a unique port outside the range of ports typically used by RTPS.

Example

The following snippet demonstrates all the APIs described on this page:

```
// Create the DataWriter
DataWriter* writer = publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr == writer)
{
    // Error
    return;
}

// Create DataReader with unique flows
DataReaderQos drqos = DATAREADER_QOS_DEFAULT;
drqos.properties().properties().emplace_back("fastdds.unique_network_flows", "");
DataReader* reader = subscriber->create_datareader(topic, drqos);

// Print locators information
eprosima::fastdds::rtps::LocatorList locators;
writer->get_sending_locators(locators);
std::cout << "Writer is sending from the following locators:" << std::endl;
for (const auto& locator : locators)
{
    std::cout << " " << locator << std::endl;
}

reader->get_listening_locators(locators);
std::cout << "Reader is listening on the following locators:" << std::endl;
for (const Locator_t& locator : locators)
{
    std::cout << " " << locator << std::endl;
}
```

7.30.10 Statistics module

eProsima Fast DDS Statistics Module allows the user to monitor the data being exchanged by its application. In order to use this module, the user must enable it in the monitored application, and create another application that receives the data being published by the statistics DataWriters. The user can also use for the latter the *eProsima Fast DDS Statistics Backend* which already implements the collection and aggregation of the data coming from the statistics topics.

- *Enable Statistics module*
- *Create monitoring application*

Enable Statistics module

The Statistics module has to be enabled both at build and runtime. On the one hand, *CMake option* `FASTDDS_STATISTICS` must be enabled when building the library (since Fast DDS v2.9.0 this CMake option is enabled by default). On the other hand, the desired statistics DataWriters should be enabled using the *Statistics Module DDS Layer*.

The statistics DataWriters can be enabled automatically using the *PropertyPolicyQos* `fastdds.statistics` and the `FASTDDS_STATISTICS` environment variable. They can also be enabled manually following the next example:

```
// Create a DomainParticipant
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↪DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// Obtain pointer to child class
eprosima::fastdds::statistics::dds::DomainParticipant* statistics_participant =
    eprosima::fastdds::statistics::dds::DomainParticipant::narrow(participant);

// Enable statistics DataWriter
if (statistics_participant->enable_statistics_
    ↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC,
    eprosima::fastdds::statistics::dds::STATISTICS_DATAWRITER_QOS) != RETCODE_OK)
{
    // Error
    return;
}

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

// Disable statistics DataWriter
if (statistics_participant->disable_statistics_
    ↪datawriter(eprosima::fastdds::statistics::GAP_COUNT_TOPIC) !=
    RETCODE_OK)
{

```

(continues on next page)

(continued from previous page)

```

    // Error
    return;
}

// Delete DomainParticipant
if (DomainParticipantFactory::get_instance()->delete_participant(participant) != 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.

7.30.11 Dynamic network interfaces

DDS *Simple Discovery* relies on well-known multicast addresses and ports to relay the Participant announcement messages (see *Discovery phases*). Such Participant announcement includes information about the unicast address-port pairs (a.k.a locators) where the Participant is expecting to receive incoming metatraffic data. The list with these unicast locators is automatically initialized taking into account the network interfaces that are available when the *Fast DDS* DomainParticipant is enabled. Consequently, any network interface that is added after enabling the DomainParticipant should be notified to *Fast DDS* in order to initialize an unicast locator in said network, so communication can be established over that new interface.

Dynamic network interface addition at run-time

In case that the user wants to include new network interfaces at run-time, some prerequisites have to be fulfilled. Then, once the interfaces are available, the user may notify Fast DDS so these interfaces are also used in the communication.

Prerequisites

This feature is intended to be used when *Fast DDS* automatically sets the listening unicast locators. Consequently, both `metatrafficUnicastLocatorList` and `metatrafficMulticastLocatorList` lists must be empty. These attributes are set within the *builtin* member of `wire_protocol()` contained in the `DomainParticipantQos` (please refer to *DomainParticipantQos*).

Note: Be aware of the remote locators' collections limits set within the *DomainParticipantQos* (please refer to *RemoteLocatorsAllocationAttributes*). It is recommended to use the highest number of local addresses found on all the

systems belonging to the same domain.

Notify *Fast DDS*

Once a new network interface has been enabled, *Fast DDS* has to be manually notified. This is done calling `DomainParticipant::set_qos()`. The `DomainParticipantQoS` that is passed to the method can either change one of the mutable `DomainParticipant` QoS or it can simply be the current `DomainParticipant` QoS (obtained with `DomainParticipant::get_qos()`).

Using `DomainParticipant::set_qos()` is the reason for the previous prerequisites: once the `DomainParticipant` is enabled, there are several QoS policies that are immutable and cannot be changed at run-time. [WireProtocolConfigQos](#) where the aforementioned lists are defined is among these immutable policies.

Find below a brief snippet of how to use this feature:

```
// Create the DomainParticipant
DomainParticipant* participant =
    DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_QOS_
    ↳DEFAULT);
if (nullptr == participant)
{
    // Error
    return;
}

// User application

// Notify Fast DDS a new network interface is available
participant->set_qos(PARTICIPANT_QOS_DEFAULT);
```

Important: This feature is still under development and only officially supported for UDPv4 Transport without whitelisting.

7.30.12 How to use eProsima DDS Record and Replay (rosbag2 and DDS)

eProsima DDS Record and Replay allows the user to continuously monitor the ROS 2 traffic in real time, and to play it back at any given time. This highly contributes to facilitating simulation of real life conditions, application testing, optimizing data analysis and general troubleshooting. [rosbag2](#) is a ROS 2 application that can be used to capture DDS messages and store them on an SQLite database which allows inspecting and replaying said messages at a later time.

rosbag2 interactions with a native Fast DDS application

Using rosbag2 to capture traffic between ROS 2 talkers and listeners is straightforward. However, recording and re-playing messages sent by Fast DDS participants outside ROS 2 ecosystem requires some modifications.

Prerequisites

A Fast DDS installation, either binary or from sources is required. Fast DDS-Gen is also required for generating the examples and Fast DDS TypeSupport from the IDL file. A ROS 2 installation with the rosbag2 package is needed as well.

DDS IDL interoperability with ROS 2 messages

DDS uses IDLs to define the data model being exchanged by the applications. While ROS 2 can use IDL files to define the messages, there are some rules that these IDL files must follow so compatibility between ROS 2 and Fast DDS native applications can be achieved. Specifically, the type definition must be nested inside the type module name and then the generator to be used. For ROS 2 messages, the generator would be `msg`, whereas in this case, the `idl` generator must be used. Assuming that the type module name selected is `fastdds_record_typesupport` the following `HelloWorld.idl` file could be defined. This IDL file will be the one used in the following steps.

```
module fastdds_record_typesupport
{
    module idl
    {
        struct HelloWorld
        {
            unsigned long index;
            string message;
        };
    };
};
```

By default, rosbag2 can only recognize those Topics which types ROS 2 has already defined in its different TypeSupport libraries. Therefore, a new ROS 2 TypeSupport module library generated with the previously defined types must be created, so rosbag2 would be able to parse the message contents coming from the Fast DDS application. First, the new ROS 2 TypeSupport package should be created. Follow the instructions below, after having sourced your ROS 2 installation:

```
ros2 pkg create --build-type ament_cmake fastdds_record_typesupport
```

This command will create a new ROS 2 package named `fastdds_record_typesupport` with the following folder structure:

```
.
├── fastdds_record_typesupport
│   ├── include
│   │   └── fastdds_record_typesupport
│   ├── src
│   ├── CMakeLists.txt
│   └── package.xml
```

ROS 2 TypeSupport code generators expect IDL files inside their own `idl` folder, so the final folder structure would be like this:

```
.
├── fastdds_record_typesupport
│   ├── idl
│   │   └── HelloWorld.idl
│   ├── include
│   │   └── fastdds_record_typesupport
│   ├── src
│   ├── CMakeLists.txt
│   └── package.xml
```

In order to generate the TypeSupport interfaces required, the CMakeLists.txt file should be modified accordingly so the ROS 2 TypeSupport generator is called. Please add the following lines to the CMakeLists.txt file before calling `ament_package()`:

```
find_package(rosidl_default_generators REQUIRED)

set(idl_files
  "idl/HelloWorld.idl"
)

rosidl_generate_interfaces(${PROJECT_NAME}
  ${idl_files}
)
```

Similarly, the package.xml file should be modified adding the ROS 2 TypeSupport generator dependency. Add the following lines to the package.xml file after the `buildtool_depend` tags:

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

The last step would be to build the package. Run the following command within the `fastdds_record_typesupport` folder:

```
RMW_IMPLEMENTATION=rmw_fastrtps_cpp colcon build
```

The build process will create inside the install folder a new ROS 2 overlay with all the required libraries and scripts for the ROS 2 applications to use the type defined in the IDL file.

Fast DDS Application tuning

ROS 2 adds special tokens to the topic names depending on the ROS 2 subsystem the topic belongs to. More information on this topic can be found [on ROS 2 design documentation](#).

Using the same IDL file defined earlier, Fast DDS-Gen can generate the required code to handle the new type in Fast DDS. The changes required in the Fast DDS application so `roscpp` can communicate with it are going to be illustrated via the Publisher/Subscriber example generated automatically from an IDL using Fast DDS-Gen. An in-depth guide to Fast DDS-Gen can be found [here](#).

In the case of plain topics, the namespace “`rt/`” is added by ROS 2 to the DDS topic name. Data Type names for ROS 2 generated types are structured concatenating the modules names. For the IDL being used in this example the data type name would be “`fastdds_record_typesupport::idl::HelloWorld`”.

Create a new workspace different from the ROS 2 one used previously. Copy inside the same IDL file and run Fast DDS-Gen to generate the TypeSupport and the example source files.

Note: Fast DDS-Gen `-no-typeobjectsupport` option is advisable to be enabled as this feature does not ensure vendor interoperability for the moment.

```
mkdir HelloWorldExample
cd HelloWorldExample
cp <PATH_TO_ROS2_WORKSPACE>/fastdds_record_typesupport/idl/HelloWorld.idl .
fastddsgen -example CMake -typeros2 -no-typeobjectsupport HelloWorld.idl
```

This command will populate the current folder with the required header and source files to build the TypeSupport, and the Publisher and Subscriber applications.

```
├─ HelloWorldExample
│  ├── CMakeLists.txt
│  ├── HelloWorldCdrAux.hpp
│  ├── HelloWorldCdrAux.ipp
│  ├── HelloWorld.hpp
│  ├── HelloWorld.idl
│  ├── HelloWorldPublisher.cxx
│  ├── HelloWorldPublisher.h
│  ├── HelloWorldPubSubMain.cxx
│  ├── HelloWorldPubSubTypes.cxx
│  ├── HelloWorldPubSubTypes.h
│  ├── HelloWorldSubscriber.cxx
│  └── HelloWorldSubscriber.h
```

The Fast DDS-Gen example should be modified taking into account the topic and type name mangling applied by ROS 2 so communication can be established with rosbag2. Having used the `-typeros2` Fast DDS-Gen option when generating the TypeSupport, the generated type name would already include the ROS 2 naming rule mangling. However, the topic name must be modified manually both in the Publisher and Subscriber applications. Look for the `create_topic` command in both the `HelloWorldPublisher.cxx` and the `HelloWorldSubscriber.cxx` files and modify the topic name:

```
topic_ = participant->create_topic(
    "rt/HelloWorldTopic",
    type_.get_type_name(),
    TOPIC_QOS_DEFAULT);
if (topic_ == nullptr)
{
    return false;
}
```

To build this example run the following commands:

```
mkdir build && cd build
cmake ..
make
```

This will create a HelloWorld binary file inside the build directory that can be used to launch both the Publisher and the Subscriber applications. Run each application in a terminal and confirm that the communication is established.

```
./HelloWorld publisher|subscriber
```

eProsima DDS Record and Replay

In order to use the generated ROS 2 TypeSupport package, the ROS 2 workspace should be sourced besides the ROS 2 installation. This allows rosbag2 to record the data types used in this example. To start recording the traffic being exchanged between the Publisher/Subscriber applications the corresponding ROS 2 Topic name has to be passed to rosbag2 (not to be mistaken with the DDS Topic name). Remember also to ensure that Fast DDS is the ROS 2 middleware being used by setting the environment variable `RMW_IMPLEMENTATION`.

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
source <PATH_TO_ROS2_WORKSPACE>/fastdds_record_typesupport/install/setup.bash
ros2 bag record /HelloWorldTopic
```

Having the Publisher application running already, the following rosbag2 log discovery info would be shown:

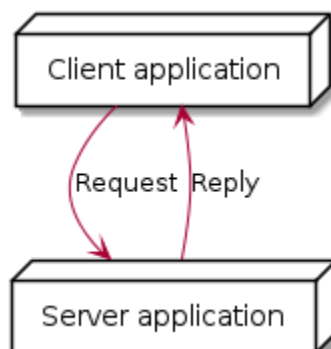
```
[INFO] [1644320308.422161532] [rosbag2_recorder]: Subscribed to topic '/HelloWorldTopic'
[INFO] [1644320308.422292205] [rosbag2_recorder]: All requested topics are subscribed.
↔ Stopping discovery...
```

rosbag2 will proceed to create a folder named `rosbag2_<DATE>` with an SQLite database inside (db3 extension) where the received messages will be recorded. Within the folder a YAML file provides metadata information about the record: type and topic name, number of messages recorded, record duration, etc. The path to this database file can be used to replay the recorded messages. Having the Subscriber application running, the previously recorded traffic will be replayed. After stopping the rosbag2 application, rerun it in replay mode running the following command. The recorded messages will be published by rosbag2 at their original publishing rate and the Subscriber application will receive them:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
source <PATH_TO_ROS2_WORKSPACE>/fastdds_record_typesupport/install/setup.bash
ros2 bag play <path-to-db-file>
```

7.30.13 Request-Reply communication

This section explains how to configure a Request-Reply communication in *Fast DDS* between two applications. A *Client application* will send a *Request* to the *Server application* and this one, after processing the request, will send a *Reply* to the *Client application*.

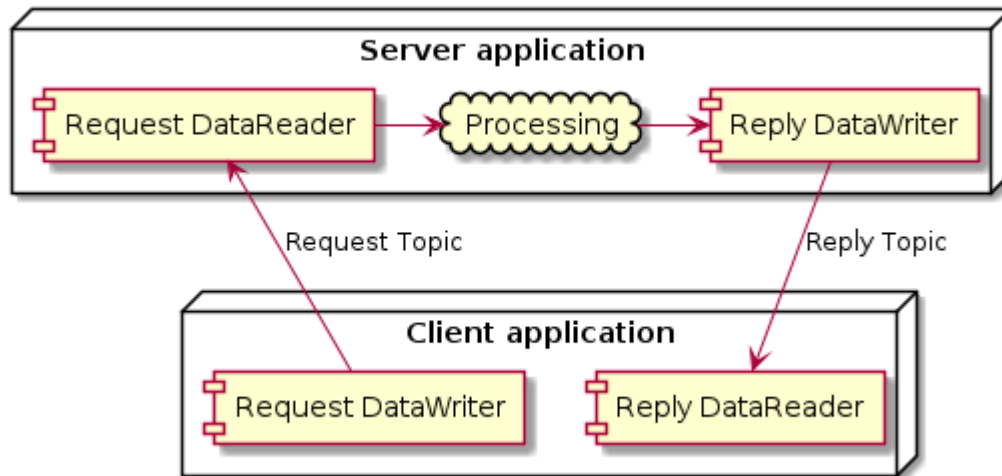


- [Overview](#)
- [Getting started](#)

- *Sending the request and storing the assigned identifier*
- *Receiving the request and sending a reply associated with it*
- *Identifying the reply for the client*

Overview

This kind of communication involves in the DDS paradigm the usage of two *Topics*: one for sending requests (Request Topic) and the other one for sending replies (Reply Topic). For managing these *Topics* four DDS entities are involved: a *DataReader* and a *DataWriter* for each *Topic*. The DDS communication schema will be:



The key for making *Request-Reply* work is relating the Request with the Reply in the client's side. *Fast DDS* API provides `SampleIdentity` to achieve this.

A full example can be found in [Fast DDS repository](#).

Getting started

For *Request-Reply* communication perform the following steps:

1. Define two structures in an IDL file. One structure will be used as Request Topic's data type and the other one as Reply Topic's data type.

```

enum OperationType
{
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION
};

struct RequestType
{
    OperationType operation;
  
```

(continues on next page)

(continued from previous page)

```

    long x;
    long y;
};

struct ReplyType
{
    long long z;
};

```

2. In the client application, create a `DataWriter` for the request and a `DataReader` for the reply.

```

participant->register_type(request_type);
participant->register_type(reply_type);

Topic* request_topic = participant->create_topic("CalculatorRequest",
        request_type.get_type_name(), TOPIC_QOS_DEFAULT);

Topic* reply_topic =
    participant->create_topic("CalculatorReply", reply_type.get_type_
↪name(), TOPIC_QOS_DEFAULT);

DataWriter* request_writer = publisher->create_datawriter(request_topic, ↪
↪DATAWRITER_QOS_DEFAULT);

DataReader* reply_reader = subscriber->create_datareader(reply_topic, ↪
↪DATAREADER_QOS_DEFAULT, &listener);

```

3. In the server application, create a `DataWriter` for the reply and a `DataReader` for the request.

```

participant->register_type(request_type);
participant->register_type(reply_type);

Topic* request_topic = participant->create_topic("CalculatorRequest",
        request_type.get_type_name(), TOPIC_QOS_DEFAULT);

Topic* reply_topic =
    participant->create_topic("CalculatorReply", reply_type.get_type_
↪name(), TOPIC_QOS_DEFAULT);

DataWriter* reply_writer = publisher->create_datawriter(reply_topic, ↪
↪DATAWRITER_QOS_DEFAULT);

DataReader* request_reader = subscriber->create_datareader(request_topic, ↪
↪DATAREADER_QOS_DEFAULT, &listener);

```

Sending the request and storing the assigned identifier

For sending the request, the client application should retrieve and store the internal identifier assigned to the published sample. Therefore the sample should be published using the overloaded `write()` function which second argument is a reference to a `WriteParams` object. The assigned identifier will be stored in the `WriteParams`'s attribute `sample_identity()`.

```
eprosima::fastrtps::rtps::SampleIdentity my_request_sample_identity;
RequestType request;

// Fill the request

// Publish request
eprosima::fastrtps::rtps::WriteParams write_params;
request_writer->write(static_cast<void*>(&request), write_params);

// Store sample identity
my_request_sample_identity = write_params.sample_identity();
```

Receiving the request and sending a reply associated with it

When the server application receives the request (for example through `on_data_available()`), it has to retrieve the request's identifier using `sample_identity`.

```
void on_data_available(
    eprosima::fastdds::dds::DataReader* reader) override
{
    RequestType request;
    eprosima::fastdds::dds::SampleInfo sample_info;

    reader->take_next_sample(&request, &sample_info);

    if (eprosima::fastdds::dds::InstanceStateKind::ALIVE_INSTANCE_STATE == sample_
    ↪info.instance_state)
    {
        // Store the request identity.
        eprosima::fastrtps::rtps::SampleIdentity client_request_identity = sample_
    ↪info.sample_identity;
    }
}
```

After processing the request, the server should send the reply to the client with the related request attached. This is done assigning the stored identifier in `related_sample_identity()`.

```
ReplyType reply;

// Fill reply

// Send reply associating it with the client request.
eprosima::fastrtps::rtps::WriteParams write_params;
write_params.related_sample_identity() = client_request_identity;
reply_writer->write(reinterpret_cast<void*>(&reply), write_params);
```

Identifying the reply for the client

When the client application receives a reply (for example through `on_data_available()`), the client application should identify if the received reply is the one expected for its request. For this the client application has to compare the stored `SampleIdentity` with the incoming `related_sample_identity`.

```
void on_data_available(
    eprosima::fastdds::dds::DataReader* reader) override
{
    ReplyType reply;
    eprosima::fastdds::dds::SampleInfo sample_info;

    reader->take_next_sample(&reply, &sample_info);

    if (eprosima::fastdds::dds::InstanceStateKind::ALIVE_INSTANCE_STATE == sample_
    ↪info.instance_state)
    {
        if (sample_info.related_sample_identity == my_request_sample_identity)
        {
            // Work to do
        }
    }
}
```

7.30.14 Remote type discovery and endpoint matching

This section explains how to create an endpoint using a remotely discovered data type that was previously unknown to the local participant.

Prerequisites

This use case focuses on the strategy to follow in order to create an endpoint at runtime in a previously unknown topic using the information provided by the remote endpoint discovery information. Therefore, the prerequisites are:

- Two participants, A and B, running in different process (type information is shared within the same `DomainParticipantFactory`).
- Participant A must not know the data type registered in participant B.
- Participant B data type must be registered using the code generated by *eProsima Fast DDS-Gen* without disabling the `TypeObjectSupport` code generation.
- Participant B must create an endpoint using the data type unknown by participant A.
- Participant A must be attached to a *DomainParticipantListener*.

Remote type discovery

Following the *participant discovery phase*, the endpoint information is exchanged. The appropriate `on_data_reader_discovery()` or `on_data_writer_discovery()` callback is called, depending on the kind of endpoint created on the remote participant. The endpoint discovery callback provides access to the remotely discovered information including the *TypeInformation*.

Provided the *TypeInformation*, *ITypeObjectRegistry* singleton can be queried for the corresponding *TypeObject representation* calling `ITypeObjectRegistry::get_type_object` API.

Register remote type

`DynamicTypeBuilderFactory` provides a specific API that given a *TypeObject representation* returns the corresponding `DynamicTypeBuilder`: `DynamicTypeBuilderFactory::create_type_w_type_object`. The `DynamicType` can then be obtained and registered using `DynamicPubSubType`.

Create local endpoint

Once the remote type has been locally registered, a *Topic* can be created within the `DomainParticipant` and endpoints using this *Topic* might be also created.

Note: Endpoint matching takes into consideration QoS consistency. Consequently, for the local endpoint to match, the remote QoS has to be taken into account. The remote endpoint discovery information provided by the discovery callback includes also this data.

Example

The following snippet shows the previously explained steps:

```
class RemoteDiscoveryDomainParticipantListener : public DomainParticipantListener
{
    /* Custom Callback on_data_reader_discovery */
    void on_data_reader_discovery(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info,
        bool& should_be_ignored) override
    {
        should_be_ignored = false;
        // Get remote type information
        xtypes::TypeObject remote_type_object;
        if (RETCODE_OK != DomainParticipantFactory::get_instance()->type_object_
↪registry().get_type_object(
        info.info.type_information().type_information.complete().typeid_with_
↪size().type_id(),
        remote_type_object))
        {
            // Error
            return;
        }
        // Register remotely discovered type
```

(continues on next page)

(continued from previous page)

```

        DynamicType::_ref_type remote_type = DynamicTypeBuilderFactory::get_instance()->
↪create_type_w_type_object(
            remote_type_object)->build();
        TypeSupport dyn_type_support(new DynamicPubSubType(remote_type));
        dyn_type_support.register_type(participant);

        // Create a Topic with the remotely discovered type.
        Topic* topic =
            participant->create_topic(info.info.topicName().to_string(), dyn_type_
↪support.get_type_name(),
            TOPIC_QOS_DEFAULT);
        if (nullptr == topic)
        {
            // Error
            return;
        }

        // Create endpoint
        Publisher* publisher = participant->create_publisher(PUBLISHER_QOS_DEFAULT);
        if (nullptr == publisher)
        {
            // Error
            return;
        }

        DataWriter* data_writer = publisher->create_datawriter(topic, DATAWRITER_QOS_
↪DEFAULT);
        if (nullptr == data_writer)
        {
            // Error
            return;
        }
    }

    /* Custom Callback on_data_writer_discovery */
    void on_data_writer_discovery(
        DomainParticipant* participant,
        eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info,
        bool& should_be_ignored) override
    {
        should_be_ignored = false;
        // Get remote type information
        xtypes::TypeObject remote_type_object;
        if (RETCODE_OK != DomainParticipantFactory::get_instance()->type_object_
↪registry().get_type_object(
            info.info.type_information().type_information.complete().typeid_with_
↪size().type_id(),
            remote_type_object))
        {
            // Error
            return;
        }
    }

```

(continues on next page)

(continued from previous page)

```

    // Register remotely discovered type
    DynamicType::_ref_type remote_type = DynamicTypeBuilderFactory::get_instance()->
    ↪create_type_w_type_object(
        remote_type_object)->build();
    TypeSupport dyn_type_support(new DynamicPubSubType(remote_type));
    dyn_type_support.register_type(participant);

    // Create a Topic with the remotely discovered type.
    Topic* topic =
        participant->create_topic(info.info.topicName().to_string(), dyn_type_
    ↪support.get_type_name(),
        TOPIC_QOS_DEFAULT);
    if (nullptr == topic)
    {
        // Error
        return;
    }

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

    // The QoS depends on the remote endpoint QoS. For simplicity, default QoS have_
    ↪been assumed.
    DataReader* data_reader = subscriber->create_datareader(topic, DATAREADER_QOS_
    ↪DEFAULT);
    if (nullptr == data_reader)
    {
        // Error
        return;
    }
}
};

```

7.31 ROS 2 using Fast DDS middleware

Fast DDS is the default middleware implementation in the [Open Source Robotic Fundation \(OSRF\) Robot Operating System ROS 2](#) in every long term (LTS) releases and most of the non-LTS releases.

ROS 2 is a state-of-the-art software for robot engineering which consists of a set of [free software libraries](#) and tools for building robot applications. This section presents some use cases and shows how to take full advantage of Fast DDS wide set of capabilities in a ROS 2 project.

The interface between the ROS 2 stack and *Fast DDS* is provided by a ROS 2 package `rmw_fastrtps`. This package is available in all ROS 2 distributions, both from binaries and from sources. `rmw_fastrtps` actually provides not one but two different ROS 2 middleware implementations, both of them using *Fast DDS* as middleware layer: `rmw_fastrtps_cpp` and `rmw_fastrtps_dynamic_cpp`. The main difference between the two is that `rmw_fastrtps_dynamic_cpp` uses introspection type support at run time to decide on the serialization/deserialization

mechanism, while `rmw_fastrtps_cpp` uses its own type support, which generates the mapping for each message type at build time. The default ROS 2 RMW implementation until *Foxy* is `rmw_fastrtps_cpp`. For *Galactic* the environment variable `RMW_IMPLEMENTATION` has to be set to select `rmw_fastrtps_cpp` in order to use *Fast DDS* as the middleware layer. This environment variable can also be used to select the `rmw_fastrtps_dynamic_cpp` implementation:

1. Exporting `RMW_IMPLEMENTATION` environment variable:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
```

or

```
export RMW_IMPLEMENTATION=rmw_fastrtps_dynamic_cpp
```

2. When launching your ROS 2 application:

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

or

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

Note: Since *Galactic* you may have to install the `rmw_fastrtps_cpp` package:

```
sudo apt install ros-galactic-rmw-fastrtps-cpp
```

7.31.1 Configuring *Fast DDS* in ROS 2

ROS 2 only allows for the configuration of certain middleware QoS (see [ROS 2 QoS policies](#)). However, `rmw_fastrtps` offers extended configuration capabilities to take full advantage of the features in *Fast DDS*. This section describes how to specify this extended configuration.

- *Changing publication mode*
- *XML configuration*
 - *XML configuration file location*
 - *Applying different profiles to different entities*
- *Example*

Changing publication mode

`rmw_fastrtps` in ROS 2 uses asynchronous publication by default. This can be easily changed setting the environment variable `RMW_FASTRTPS_PUBLICATION_MODE` to one of the following allowed values:

- **ASYNCHRONOUS**: asynchronous publication mode. Setting this mode implies that when the publisher invokes the write operation, the data is copied into a queue, a background thread (asynchronous thread) is notified about the addition to the queue, and control of the thread is returned to the user before the data is actually sent. The background thread is in charge of consuming the queue and sending the data to every matched reader.

- **SYNCHRONOUS**: synchronous publication mode. Setting this mode implies that the data is sent directly within the context of the user thread. This entails that any blocking call occurring during the write operation would block the user thread, thus preventing the application from continuing its operation. It is important to note that this mode typically yields higher throughput rates at lower latencies, since there is no notification nor context switching between threads.
- **AUTO**: let Fast DDS select the publication mode. This implies using the publication mode set in the *XML file*, or otherwise, the default value set in Fast DDS (see *PublishModeQosPolicy*).

rmw_fastrtps defines two configurable parameters in addition to **ROS 2 QoS policies**. Said parameters, and their default values under ROS 2, are:

Parameter	Description	Default value	ROS 2
<i>Memory-ManagementPolicy</i>	<i>Fast DDS</i> preallocates memory for the publisher and subscriber histories. When those histories fill up, a reallocation occurs to reserve more memory.	PREALLOCATED_WITH_REALLOC_MEMORY_MODE	
<i>Publish-Mode-QosPolicy</i>	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	rmw_qos_profile_t	Fast DDS XML QoS	Fast DDS XML history memory policy and publication mode
0 (default)	Default values	Overridden by <code>rmw_qos_profile_t</code>	Overridden by <code>rmw_fastrtps</code> default value
0 (default)	Non system default	Overridden by <code>rmw_qos_profile_t</code>	Overridden by <code>rmw_fastrtps</code> default value
0 (default)	System default	Used	Overridden by <code>rmw_fastrtps</code> default value
1	Default values	Overridden by <code>rmw_qos_profile_t</code>	Used
1	Non system default	Overridden by <code>rmw_qos_profile_t</code>	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 `FASTDDS_DEFAULT_PROFILES_FILE` to contain the path to the XML configuration file (see [Environment variables](#)).

```
export FASTDDS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
```

- **Alternative:** Placing the XML file in the running application directory under the name `DEFAULT_FASTDDS_PROFILES.xml`.

For example:

```
export FASTDDS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run <package> <application>
```

Applying different profiles to different entities

`rmw_fastrtps` allows for the configuration of different entities with different QoS using the same XML file. For doing so, `rmw_fastrtps` locates profiles in the XML based on topic names.

Creating publishers/subscribers with different profiles

- To configure a publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic prepended by the node namespace (which defaults to "" if not specified), i.e. the node's namespace followed by topic name used to create the publisher. Mind that topic names always start with / (it is added when creating the topic if not present), and that namespace and topic name are always separated by one /. If such profile is not defined, `rmw_fastrtps` attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.
- To configure a subscriber, define a `<data_reader>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the topic prepended by the node namespace (which defaults to "" if not specified), i.e. the node's namespace followed by topic name used to create the subscriber. Mind that topic names always

start with / (it is added when creating the topic if not present), and that namespace and topic name are always separated by one /. If such profile is not defined, *rmw_fastrtps* attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.

The following table presents different combinations of node namespaces and user specified topic names, as well as the resulting topic names and the suitable profile names:

User specified topic name	Node namespace	Final topic name	Profile name
chatter	DEFAULT ("")	/chatter	/chatter
chatter	test_namespace	/test_namespace/chatter	/test_namespace/chatter
chatter	/test_namespace	/test_namespace/chatter	/test_namespace/chatter
/chatter	test_namespace	/chatter	/chatter
/chatter	/test_namespace	/chatter	/chatter

Important: Node namespaces are NOT prepended to user specified topic names starting with /, a.k.a Fully Qualified Names (FQN). For a complete description of topic name remapping please refer to [Remapping Names](#).

Creating services with different profiles

ROS 2 services contain a subscriber for receiving requests, and a publisher to reply to them. *rmw_fastrtps* allows for configuring each of these endpoints separately in the following manner:

- To configure the request subscriber, define a `<data_reader>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. For more information on name mangling, please refer to [Topic and Service name mapping to DDS](#). If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_reader>` profile with attribute `profile_name="service"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.
- To configure the reply publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_writer>` profile with attribute `profile_name="service"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.

Creating clients with different profiles

ROS 2 clients contain a publisher to send requests, and a subscription to receive the service's replies. *rmw_fastrtps* allows for configuring each of these endpoints separately in the following manner:

- To configure the requests publisher, define a `<data_writer>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_writer>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_writer>` profile with attribute `is_default_profile="true"`.
- To configure the reply subscription, define a `<data_reader>` profile with attribute `profile_name=topic_name`, where `topic_name` is the name of the service after mangling. If such profile is not defined, *rmw_fastrtps* attempts to load a `<data_reader>` profile with attribute `profile_name="client"`. If neither of the previous profiles exist, *rmw_fastrtps* attempts to load the `<data_reader>` profile with attribute `is_default_profile="true"`.

Creating ROS contexts and nodes

ROS *context* and *node* entities are mapped to *Fast DDS* Participant entity, according to the following table:

ROS entity	<i>Fast DDS</i> entity since <i>Foxy</i>	<i>Fast DDS</i> entity in <i>Eloquent</i> & below
Context	Participant	<i>Not DDS direct mapping</i>
Node	<i>Not DDS direct mapping</i>	Participant

This means that on *Foxy* and later releases, contexts can be configured using a `<Participant>` profile with attribute `is_default_profile="true"`. The same profile will be used in *Eloquent* and below to configure nodes.

For example, a profile for a ROS 2 context on *Foxy* and later releases would be specified as:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <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 version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com">
  <participant profile_name="participant_profile_ros2" is_default_profile="true"
  ↪">
    <rtps>
      <name>profile_for_ros2_context</name>
    </rtps>
  </participant>

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

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

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

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

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

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

```

2. Open one terminal and run:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export FASTDDS_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 FASTDDS_DEFAULT_PROFILES_FILE=path/to/xml/ros_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp listener
```

7.31.2 Use ROS 2 with Fast-DDS Discovery Server

This section explains how to run some ROS 2 examples using the Discovery Servers as discovery communication. In order to get more information about the specific use of this configuration, please check the [Discovery Server Documentation](#) or read the *common use cases* for this configuration.

The following tutorial gathers the steps to check this functionality and learn how to use it with ROS 2.

- *Discovery Server v2*
- *Prerequisites*
- *Run the demo*
 - *Setup Discovery Server*
 - *Launch node listener*
 - *Launch node talker*
 - *Demonstrate Discovery Server execution*
- *Advance user cases*
 - *Server Redundancy*
 - *Backup Server*
 - *Discovery partitions*
- *ROS 2 Introspection*
 - *Daemon's related commands*
 - *No Daemon commands*
- *Compare Discovery Server with Simple Discovery*

The *Simple Discovery Protocol* is the standard protocol defined in the [DDS standard](#). However, it has certain known disadvantages in some scenarios, mainly:

- It does not **Scale** efficiently, as the number of exchanged packets highly increases as new nodes are added.
- It requires **Multicasting** capabilities that may not work reliably in some scenarios, e.g. WiFi.

The **Discovery Server** provides a Client-Server Architecture that allows the nodes to connect with each other using an intermediate server. Each node will work as a *Client*, sharing its info with the *Discovery Server* and receiving the

discovery information from it. This means that the network traffic is highly reduced in big systems, and it does not require *Multicasting*.

These **Discovery Servers** can be independent, duplicated or connected with each other in order to create redundancy over the network and avoid having a *Single-Point-Of-Failure*.

Discovery Server v2

The new version **v2** of Discovery Server, available from *Fast DDS* v2.0.2, implements a new filter feature that allows to further reduce the number of discovery messages sent. This version uses the *topic* of the different nodes to decide if two nodes must be connected, or they could be left unmatched. The following schema represents the decrease of the discovery packages:

This architecture reduces the number of packages sent between the server and the different clients dramatically. In the following graph, the reduction in traffic network over the discovery phase for a RMF Clinic demo use case, is shown:

In order to use this functionality, **Fast-DDS Discovery Server** can be set using the XML configuration for Participants. Furthermore, Fast DDS provides an easier way to set a **Discovery Server** communication using the `fastdds` *CLI tool* and an *environment variable*, which are going to be used along this tutorial. For a more detailed explanation about the configuration of the Discovery Server, visit *Discovery Server Settings*.

Prerequisites

This tutorial assumes you have at least a [working Foxy ROS 2 installation](#). In case your installation is using a Fast DDS version lower than v2.0.2 you could not use the `fastdds` tool. You could update your repository to use a different Fast DDS version, or *set the discovery server by Fast-DDS XML QoS configuration*.

Note: This tutorial can also be run in *Galactic* exporting the environment variable that selects *Fast DDS* as the middleware layer:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
```

Run the demo

The `talker-listener` ROS 2 demo allows to create a *talker* node that publishes a *Hello World* message every second, and a *listener* node that listens to these messages.

By [Sourcing ROS 2](#) you will get access to the CLI of *Fast DDS*: `fastdds`. This CLI gives access to the *discovery tool*, which allows to launch a server. This server will manage the discovery process for the nodes that connect to it.

Important: Do not forget to [source ROS 2](#) in every new terminal opened.

Setup Discovery Server

Start by launching a server with id 0, with port 11811 and listening on all available interfaces.

Open a new terminal and run:

```
fastdds discovery -i 0
```

Launch node listener

Execute the listener demo, that will listen in `/chatter` topic.

In a new terminal, set the environment variable `ROS_DISCOVERY_SERVER` to use *Discovery Server*. (Do not forget to source ROS 2 in every new terminal)

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
```

Afterwards, launch the listener node. Use the argument `--remap __node:=listener_discovery_server` to change the node's name for future purpose.

```
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener_discovery_server
```

This process will create a ROS 2 node, that will automatically create a client for the *Discovery Server* and use the server created previously to run the discovery protocol.

Launch node talker

Open a new terminal and set the environment variable as before, so the node raises a client for the discovery protocol.

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker_discovery_server
```

Now, we should see the talker publishing *Hello World* messages, and the listener receiving these messages.

Demonstrate Discovery Server execution

So far, there is not proof that this example and the standard talker-listener example run differently. For this purpose, run another node that is not connected to our Discovery Server. Just run a new listener (listening in `/chatter` topic by default) in a new terminal and check that it is not connected to the talker already running.

```
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=simple_listener
```

In this case, we should not see the listener receiving the messages.

To finally verify that everything is running correctly, a new talker can be created using the *simple discovery protocol*.

```
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=simple_talker
```

Now we should see the listener *simple_listener* receiving the messages from *simple_talker* but not the other messages from *talker_discovery_server*.

Advance user cases

The following paragraphs are going to show different features of the Discovery Server that allows to hold a robust structure over the node's network.

Server Redundancy

By using the Fast DDS tool, several servers can be created, and the nodes can be connected to as many servers as desired. This allows to have a safe redundancy network that will work even if some servers or nodes shut down unexpectedly. Next schema shows a simple architecture that will work with server redundancy:

In different terminals, run the next code to establish a communication over redundant servers.

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

```
fastdds discovery -i 1 -l 127.0.0.1 -p 11888
```

-i N means server with id N. When referencing the servers with ROS_DISCOVERY_SERVER, server 0 must be in first place and server 1 in second place.

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

Now, if one of these servers fails, there would still be discovery communication between nodes.

Backup Server

Fast DDS Discovery Server allows to easily build a server with a **backup** functionality. This allows the server to retake the last state it saved in case of a shutdown.

In different terminals, run the next code to establish a communication over a backup server.

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811 -b
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener
```

Several backup files are created in the path the server has run. Two SQLite files and two json files that contains the information required to raise a new server in case of failure, avoiding the whole discovery process to happen again and without losing information.

Discovery partitions

The **Discovery Server** communication could be used with different servers to split in virtual partitions the discovery info. This means that two endpoints only would know each other if there is a server or a server network between them. We are going to execute an example with two different independent servers. The following image shows a schema of the architecture desired:

With this schema *Listener 1* will be connected to *Talker 1* and *Talker 2*, as they share *Server 1*. *Listener 2* will connect with *Talker 1* as they share *Server 2*. But *Listener 2* will not hear the messages from *Talker 2* because they do not share any server or servers' network that connect them.

Run the first server listening in localhost in default port 11811.

```
fastdds discovery -i 0 -l 127.0.0.1 -p 11811
```

In another terminal run the second server listening in localhost in port another port, in this case 11888.

```
fastdds discovery -i 1 -l 127.0.0.1 -p 11888
```

Now, run each node in a different terminal. Use the *environment variable* `ROS_DISCOVERY_SERVER` to decide which server they are connected to. Be aware that the ids must match (*Environment variables*).

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker_1
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811;127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener_1
```

```
export ROS_DISCOVERY_SERVER="127.0.0.1:11811"
ros2 run demo_nodes_cpp talker --ros-args --remap __node:=talker_2
```

```
export ROS_DISCOVERY_SERVER="";127.0.0.1:11888"
ros2 run demo_nodes_cpp listener --ros-args --remap __node:=listener_2
```

We should see how *Listener 1* is receiving double messages, while *Listener 2* is in a different partition from *Talker 2* and so it does not listen to it.

Note: Once two endpoints know each other, they do not need the server network between them to listen to each other messages.

ROS 2 Introspection

ROS 2 Command Line Interface (CLI) implements several introspection features to analyze the behaviour of a ROS 2 execution. These features (i.e. *rosviz*, *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 FASTDDS_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 FASTDDS_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 in 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 FASTDDS_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 generate 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 generate a graph similar to the one below:

```
$ export FASTDDS_DEFAULT_PROFILES_FILE="no_intraprocess_configuration.xml"
$ sudo bash generate_discovery_packages.bash ~/ros2_foxy/install/local_setup.bash
```

(continues on next page)

(continued from previous page)

```
$ sudo bash generate_discovery_packages.bash ~/ros2_foxy/install/local_setup.bash SERVER
$ python3 discovery_packets.py
```

This graph is the result of a 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.

7.32 Troubleshooting

This section offers hints and pointers to help users with navigating through the documentation while troubleshooting issues.

- Although UDP/SHM default transports of Fast DDS are designed to work in most network environments, they may encounter certain limitations when operating over WiFi or within lossy network conditions. In these cases, it is advisable to set up the `LARGE_DATA` configuration, which has been specifically optimized for these scenarios. The `LARGE_DATA` profile limits the use of UDP solely to the *PDP discovery* phase, employing the more reliable TCP/SHM for the remainder of the communication process. Its implementation can be accomplished by simply configuring the `FASTDDS_BUILTIN_TRANSPORTS` environment variable, or alternatively through XML profiles or via code. For more information, please refer to *Large Data Mode*.

Environment Variable

```
export FASTDDS_BUILTIN_TRANSPORTS=LARGE_DATA
```

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<dds xmlns="http://www.eprosima.com">
  <profiles>
    <!--
      UDP transport for PDP and SHM/TCPv4 transport for both EDP and
      ↪ application data
    -->
    <participant profile_name="large_data_builtin_transports" is_default_
      ↪ profile="true">
      <rtps>
        <builtinTransports>LARGE_DATA</builtinTransports>
      </rtps>
    </participant>
  </profiles>
</dds>
```

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos = PARTICIPANT_QOS_DEFAULT;

/* Transports configuration */
// UDPv4 transport for PDP over multicast and SHM / TCPv4 transport for EDP and
↪ application data
pqos.setup_transports(eprosima::fastdds::rtps::BuiltinTransports::LARGE_DATA);

/* Create participant as usual */
eprosima::fastdds::dds::DomainParticipant* participant =
    eprosima::fastdds::dds::DomainParticipantFactory::get_instance()->create_
↪ participant(0, pqos);
```

- If having problems with transmitting **large samples** when using the LARGE_DATA mode, try to use the builtin transports configuration options to adjust LARGE_DATA to your specific use case. Please refer to [Large Data with configuration options](#) for more information.
- If having problems with transmitting **large samples such as video or point clouds**, please refer to [Large Data Rates](#).

7.33 C++ API Reference

Fast DDS, as a Data Distribution Service (DDS) standard implementation, exposes the DDS Data-Centric Publish-Subscribe (DCPS) Platform Independent Model (PIM) API, as specified in the [DDS specification](#). Furthermore, is also gives the user the possibility to directly interact with the underlying Real-time Publish-Subscribe (RTPS) API that DDS implements for wired communications, as specified in the [RTPS standard](#).

This section presents the most commonly used APIs provided by *Fast DDS*. For more information about the API reference, please refer to [Fast DDS API reference](#).

7.33.1 DDS DCPS PIM

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

Core

Entity

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DomainEntity

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Policy

DataRepresentationId

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataRepresentationQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataSharingQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataSharingKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DeadlineQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DestinationOrderQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DestinationOrderQosPolicyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DisablePositiveACKsQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DurabilityQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DurabilityQosPolicyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DurabilityServiceQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EntityFactoryQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GenericDataQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GroupDataQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

HistoryQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

HistoryQosPolicyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LatencyBudgetQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LifespanQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LivelinessQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LivelinessQosPolicyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

OwnershipQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

OwnershipQosPolicyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

OwnershipStrengthQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ParticipantResourceLimitsQos

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Partition_t

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PartitionQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PresentationQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PresentationQosPolicyAccessScopeKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PropertyPolicyQos

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PublishModeQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PublishModeQosPolicyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

QosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

QosPolicyId_t

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReaderDataLifecycleQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReliabilityQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReliabilityQosPolicyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ResourceLimitsQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSEndpointQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TimeBasedFilterQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TopicDataQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TransportConfigQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TransportPriorityQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TypeConsistencyEnforcementQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TypeConsistencyKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

UserDataQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WireProtocolConfigQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterDataLifecycleQosPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterResourceLimitsQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Status

BaseStatus

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DeadlineMissedStatus

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

IncompatibleQosStatus

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

InconsistentTopicStatus

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LivelinessChangedStatus

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

MatchedStatus

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

OfferedDeadlineMissedStatus

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

OfferedIncompatibleQosStatus

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PublicationMatchedStatus

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

QosPolicyCount

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

QosPolicyCountSeq

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RequestedDeadlineMissedStatus

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RequestedIncompatibleQosStatus

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LivelinessLostStatus

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SampleLostStatus

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SampleRejectedStatus

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SampleRejectedStatusKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

StatusMask

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SubscriptionMatchedException

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Condition

Condition

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ConditionSeq

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GuardCondition

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

StatusCondition

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Wait-set

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LoanableArray

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LoanableCollection

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LoanableSequence

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

StackAllocatedSequence

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Domain

DomainParticipant

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DomainParticipantFactory

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DomainParticipantFactoryQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DomainParticipantListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DomainParticipantQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Publisher

DataWriter

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataWriterListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataWriterQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Publisher

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PublisherListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PublisherQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSReliableWriterQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Subscriber

DataReader

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataReaderListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataReaderQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

InstanceStateKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReadCondition

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReaderResourceLimitsQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSReliableReaderQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SampleInfo

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SampleStateKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Subscriber

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SubscriberListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SubscriberQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TypeConsistencyQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ViewStateKind

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Topic

TopicDataType

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TypeSupport

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TopicDescription

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Topic

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ContentFilteredTopic

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ContentFilter

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ContentFilterFactory

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TopicListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TopicQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TypeInformationParameter

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

XTypes

Dynamic Language Binding

AnnotationDescriptor

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DynamicData

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DynamicDataFactory

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DynamicPubSubType

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DynamicType

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DynamicTypeBuilder

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DynamicTypeBuilderFactory

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DynamicTypeMember

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

MemberDescriptor

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TypeDescriptor

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

VerbatimTextDescriptor

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

InvalidArgumentError

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Type Representation

TypeObjectRegistry

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TypeObjectUtils

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

7.33.2 RTPS

eProsima Fast DDS Real-Time Publish-Subscribe (RTPS) layer API.

Attributes

BuiltinAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

c_default RTPSParticipantAllocationAttributes

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DiscoveryProtocol

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DiscoverySettings

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EndpointAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ExternalLocators

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

HistoryAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

InitialAnnouncementConfig

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ParticipantFilteringFlags

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PropertyPolicy

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PropertyPolicyHelper

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReaderAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReaderTimes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RemoteLocatorsAllocationAttributes

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RemoteServerAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RemoteServerList_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSParticipantAllocationAttributes

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSParticipantAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSWriterPublishMode

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SendBuffersAllocationAttributes

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SimpleEDPAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ThreadSettings

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

VariableLengthDataLimits

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterAttributes

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterTimes

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Builtin data

ContentFilterProperty

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Common

BinaryProperty

BinaryProperty

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

BinaryPropertyHelper

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

BinaryPropertySeq

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

CacheChange

CacheChange_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ChangeForReader_t

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ChangeForReaderCmp

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ChangeForReaderStatus_t

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ChangeKind_t

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

CDRMessage

CDRMessage_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Macro definitions (#define)

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

CDRSerialization

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EntityId

Const values

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-
rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Macro definitions (#define)

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EntityId_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EntityId_t Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

FragmentNumber

FragmentNumber_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

FragmentNumberSet_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Guid

c_Guid_Unknown

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GUID_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GUID_t Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GuidPrefix

c_GuidPrefix_Unknown

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GuidPrefix_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

GuidPrefix_t Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

InstanceHandle

c_InstanceHandle_Unknown

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

InstanceHandle_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

InstanceHandle_t Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Locator

Macro definitions (#define)

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

IsAddressDefined

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

IsLocatorValid

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Locator_t

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LocatorList

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LocatorListConstIterator

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LocatorListIterator

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LocatorsIterator

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Locators

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Locator Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LocatorSelectorEntry

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LocatorSelector

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LocatorWithMask

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

MatchingInfo

MatchingInfo

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

MatchingStatus

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PortParameters

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Property

Property

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PropertyHelper

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PropertySeq

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RemoteLocators

RemoteLocators Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RemoteLocatorList

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SampleIdentity

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SequenceNumber

c_SequenceNumber_Unknown

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SequenceNumber_t Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SequenceNumber_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SequenceNumberDiff

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SequenceNumberHash

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SequenceNumberSet_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

sort_seqNum

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SerializedPayload

Macro definitions (#define)

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SerializedPayload_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Time_t

Const values

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Macro definitions (#define)

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

eprosima::fastrtps::Duration_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

eprosima::fastrtps::Time_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Time_t Operators

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Time_t

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Token

AuthenticatedPeerCredentialToken

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataHolder

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataHolderHelper

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataHolderSeq

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

IdentityStatusToken

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

IdentityToken

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PermissionsCredentialToken

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PermissionsToken

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Token

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Types

BuiltinEndpointSet_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Const values

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Count_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Macro definitions (#define)

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DurabilityKind_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Endianness_t

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EndpointKind_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

octet

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ProtocolVersion_t

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReliabilityKind_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SubmessageFlag

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TopicKind_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

VendorId_t

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriteParams

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Endpoint

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Exceptions

Exception

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Flow control

FlowControllerDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

FlowControllerSchedulerPolicy

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

History

History

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

IChangePool

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

IPayloadPool

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReaderHistory

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterHistory

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSParticipant

ParticipantDiscoveryInfo

ParticipantAuthenticationInfo

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenfunction: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ParticipantDiscoveryInfo

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ParticipantProxyData

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReaderDiscoveryInfo

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ReaderProxyData

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterDiscoveryInfo

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterProxyData

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSParticipant

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSParticipantListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSReader

ReaderListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSReader

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Resources

MemoryManagementPolicy

Warning: doxygenenum: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSDomain

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

RTPSWriter

RTPSWriter

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

WriterListener

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

7.33.3 Transport

eProsima Fast DDS Transport Layer API.

Transport Generic Interfaces

TransportDescriptorInterface

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TransportInterface

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygentypedef: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TransportReceiverInterface

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

PortBasedTransportDescriptor

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

SocketTransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Chaining of transports

ChainingTransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

ChainingTransport

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

UDP Transport

UDPTransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

UDIPv4TransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

UDIPv6TransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TCP Transport

TCPTransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TCPv4TransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

TCPv6TransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Shared Memory Transport

SharedMemTransportDescriptor

Warning: doxygenstruct: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

7.33.4 LOG

eProsima Fast DDS Logging Module API

Colors

A collection of macros for ease the stream coloring.

Color Blue

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright Blue

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright Cyan

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright Green

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright Magenta

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright Red

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright White

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Bright Yellow

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Cyan

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Def

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Green

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Magenta

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Red

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color White

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Color Yellow

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

FileConsumer

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Log

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

LogConsumer

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtsp/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EPROSIMA_LOG_ERROR

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EPROSIMA_LOG_INFO

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

EPROSIMA_LOG_WARNING

Warning: doxygendefine: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

OStreamConsumer

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

StdoutConsumer

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

StdoutErrConsumer

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

7.33.5 Statistics

eProsima Fast DDS Statistics Module extension API.

DomainParticipant

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataWriterQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

DataReaderQos

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenclass: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Topic names

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

Warning: doxygenvariable: Cannot find file: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/3.0.x-devel/build/doxygen/xml/index.xml

7.34 Python API Reference

This section presents the most commonly used Python APIs provided by *Fast DDS*.

7.34.1 DDS DCPS PIM

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

Core

Entity

DomainEntity

Policy

DataRepresentationId

DataRepresentationQosPolicy

DataSharingQosPolicy

DataSharingKind

DeadlineQosPolicy

DestinationOrderQosPolicy

DestinationOrderQosPolicyKind

DisablePositiveACKsQosPolicy

DurabilityQosPolicy

DurabilityQosPolicyKind

DurabilityServiceQosPolicy

EntityFactoryQosPolicy

GenericDataQosPolicy

GroupDataQosPolicy

HistoryQosPolicy

HistoryQosPolicyKind

LatencyBudgetQosPolicy

LifespanQosPolicy

LivelinessQosPolicy

LivelinessQosPolicyKind

OwnershipQosPolicy

OwnershipQosPolicyKind

OwnershipStrengthQosPolicy

ParticipantResourceLimitsQos

Partition_t

PartitionQosPolicy

PresentationQosPolicy

PresentationQosPolicyAccessScopeKind

PropertyPolicyQos

PublishModeQosPolicy

PublishModeQosPolicyKind

QosPolicy

QosPolicyId_t

ReaderDataLifecycleQosPolicy

ReliabilityQosPolicy

ReliabilityQosPolicyKind

ResourceLimitsQosPolicy

RTPSEndpointQos

TimeBasedFilterQosPolicy

TopicDataQosPolicy

TransportConfigQos

TransportPriorityQosPolicy

TypeConsistencyEnforcementQosPolicy

TypeConsistencyKind

UserDataQosPolicy

WireProtocolConfigQos

WriterDataLifecycleQosPolicy

WriterResourceLimitsQos

Status

BaseStatus

DeadlineMissedStatus

IncompatibleQosStatus

InconsistentTopicStatus

LivelinessChangedStatus

MatchedStatus

OfferedDeadlineMissedStatus

OfferedIncompatibleQosStatus

PublicationMatchedStatus

QosPolicyCount

QosPolicyCountSeq

RequestedDeadlineMissedStatus

RequestedIncompatibleQosStatus

LivelinessLostStatus

SampleLostStatus

SampleRejectedStatus

SampleRejectedStatusKind

StatusMask

SubscriptionMatchedStatus

LoanableArray

LoanableCollection

LoanableSequence

StackAllocatedSequence

Domain

DomainParticipant

DomainParticipantFactory

DomainParticipantFactoryQos

DomainParticipantListener

DomainParticipantQos

Publisher

DataWriter

DataWriterListener

DataWriterQos

Publisher

PublisherListener

PublisherQos

RTPSReliableWriterQos

Subscriber

DataReader

DataReaderListener

DataReaderQos

InstanceStateKind

ReaderResourceLimitsQos

RTPSReliableReaderQos

SampleInfo

SampleStateKind

Subscriber

SubscriberListener

SubscriberQos

TypeConsistencyQos

ViewStateKind

Topic

Topic

TopicDataType

TopicDescription

TopicListener

TopicQos

TypeldV1

TypeInformation

TypeObjectV1

TypeSupport

7.35 Introduction

eProsima Fast DDS-Gen is a Java application that generates *eProsima Fast DDS* source code using the data types defined in an IDL (Interface Definition Language) file. This generated source code can be used in any *Fast DDS* application in order to define the data type of a topic, which will later be used to publish or subscribe. *eProsima Fast DDS* defines the data type exchanged in a Topic through two classes: the `TypeSupport` and the `TopicDataType`. `TopicDataType` describes the data type exchanged between a publication and a subscription, i.e. the data corresponding to a Topic; while `TypeSupport` encapsulates an instance of `TopicDataType`, providing the functions needed to register the type and interact with the publication and subscription. Please refer to [Definition of data types](#) for more information on data types.

To declare the structured data, the IDL format must be used. IDL is a specification language, made by [OMG](#) (Object Management Group), which describes an interface in a language independent manner, allowing communication between software components that do not share the same language. The *eProsima Fast DDS-Gen* tool reads the IDL files and parses a subset of the [OMG IDL specification](#) to generate source code for data serialization. This subset includes the data type descriptions included in [Defining a data type via IDL](#). The rest of the file content is ignored.

eProsima Fast DDS-Gen generated source code uses [Fast CDR](#), a C++11 library that provides the data serialization and codification mechanisms. Therefore, as stated in the [RTPS standard](#), when the data are sent, they are serialized and encoded using the corresponding Common Data Representation (CDR). The CDR transfer syntax is a low-level representation for inter-agents transfer, mapping from OMG IDL data types to byte streams. Please refer to the official [CDR specification](#) for more information on the CDR transfer syntax (see PDF section 15.3).

The main feature of *eProsima Fast DDS-Gen* is to facilitate the implementation of DDS applications without the knowledge of serialization or deserialization mechanisms. With *Fast DDS-Gen* it is also possible to generate the C++ source code of a DDS application with a publisher and a subscriber that uses the *eProsima Fast DDS* library (see [Building](#)

a *publish/subscribe application*). *Fast DDS-Gen* can also generate Python bindings for the data types in order to use them within a Python-based *Fast DDS* application (see *Building Python auxiliary libraries*).

For installing *Fast DDS-Gen*, please refer to *Linux installation of Fast DDS-Gen* or to *Windows installation of Fast DDS-Gen*.

7.36 Usage

This section explains the usage of *Fast DDS-Gen* tool and briefly describes the generated files.

7.36.1 Running the *Fast DDS-Gen* Java application

First, the steps outlined in *Linux installation of Fast DDS-Gen* or *Window installation of Fast DDS-Gen* must be accomplished for the installation of *Fast DDS-Gen*. According to this section, an executable file for Linux and Windows that runs the Java *Fast DDS-Gen* application is available in the `scripts` folder. If the `scripts` folder path is added to the `PATH` environment variable, *Fast DDS-Gen* can be executed running the following commands:

- Linux:

```
$ fastddsgen
```

- Windows:

```
> fastddsgen.bat
```

Note: In case the `PATH` has not been modified, these scripts can be found in the `<fastddsgen_directory>/scripts` directory.

7.36.2 Supported options

The expected argument list of the application is:

```
fastddsgen [<options>] <IDL file> [<IDL file> ...]
```

Where the options are:

Option	Description
-cs	Enables case sensitivity in variable names.
-d <directory>	Sets the output directory for the generated files.
-default-container-prealloc-size	Sets the default preallocated size for unbounded collections (sequences and maps) Default value: 0 (empty collection).
-default_extensibility <extensibility> -de <extensibility>	Sets the default extensibility for types without the @extensibility annotation. Values: -final - appendable (default) - mutable
-example <platform>	Generates an example and a solution to compile the generated source code for a specific platform. The help command shows the supported platforms.
-extrastg <template> <output>	Specifies a custom template used for generating source code. This option expects the location of the template and the location of the file where source code output will be stored. A custom template example can be found in this link
-flat-output-dir	Ignores input files relative paths and place all generated files in the specified output directory.
-help	Shows the help information
-I <directory>	Adds directory to preprocessor include paths.
-no-typesupport	Avoids generating the type support files.
-no-typeobjectsupport	Avoids generating the TypeObject support specific files. Enabled automatically if -no-typesupport argument is used.
-no-dependencies	Avoids processing the dependent IDL files.
-ppDisable	Disables the preprocessor.
-ppPath	Specifies the preprocessor path.
-python	Generates source code and a CMake solution to compile a library containing the data types Python bindings required to run a <i>Fast DDS</i> Python-based application. This option is incompatible with <i>-example</i> .
-replace	Replaces the generated source code files even if they exist.
-t <directory>	Sets a specific directory as a temporary directory.
-typeros2	Generates type naming compatible with ROS 2.
-version	Shows the current version of eProsima <i>Fast DDS-Gen</i> .

Please refer to *XTypes* for more information on TypeObject representation.

7.37 Building a publish/subscribe application

Fast DDS-Gen can be used to build a fully functional publication/subscription application from an IDL file that defines the Topic under which messages are published and received. The application generated allows for the creation of as many publishers and subscribers as desired, all belonging to the same Domain and communicating using the same Topic.

- *Background*
- *Prerequisites*
- *Create the application workspace*
- *Import linked libraries and its dependencies*
 - *Installation from binaries*
 - *Colcon installation*

- *Creating the IDL file with the data type*
- *Generating a minimal functional example*
 - *Generate the Fast DDS source code*
 - *Build the Fast DDS application*
 - *Run the Fast DDS application*
- *Summary and next steps*

7.37.1 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.

7.37.2 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.

7.37.3 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.



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
```

7.37.4 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/fastdds/` 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\fastdds <major>.<minor>.<patch>\include\fastdds` and `C:\Program Files\eProsima\fastdds <major>.<minor>.<patch>\include\fastcdr\` for *Fast DDS* and *Fast CDR* respectively. The compiled libraries of both can be found in the directory `C:\Program Files\eProsima\fastdds <major>.<minor>.<patch>\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`.

7.37.5 Creating the IDL file with the data type

To build a minimal application, the Topic must be defined by means of an IDL file. For this example the Topic data type defined by IDL is just a `string` message. Topics are explained in more detail in [Topic](#), while the Topic data types to be defined using IDL are presented in [Definition of data types](#). In the preferred text editor, create the *HelloWorld.idl* file with the following content and save it in the *FastDDSGenHelloWorld* directory.

```
// HelloWorld.idl
struct HelloWorld
{
    string message;
};
```

Then, this file is translated to something *Fast DDS* understands. For this, use the *Fast DDS-Gen* code generation tool, which can do two different things:

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

The second option is the one used to create this publish/subscribe application, while the first option is applied in this other tutorial: [Writing a simple C++ publisher and subscriber application](#).

7.37.6 Generating a minimal functional example

If the steps outlined in the Installation Manual have been followed, then *Fast DDS*, *Fast CDR*, and *Fast-RTPS-Gen* should be installed in the system.

Generate the Fast DDS source code

The application files are generated using the following command. The `-example` option creates an example application, and the CMake files needed to build it. In the workspace directory (*FastDDSGenHelloWorld* directory), execute one of the following commands according to the installation followed and the operating system.

- On Linux:

- For an **installation from binaries** or a **colcon installation**:

```
<path-to-Fast-DDS-workspace>/src/fastddsgen/scripts/fastddsgen -example
CMake HelloWorld.idl
```

- For a **stand-alone installation**, run:

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

- On Windows:

- For a **colcon installation**:

```
<path-to-Fast-DDS-workspace>/src/fastddsgen/scripts/fastddsgen.bat -example
CMake HelloWorld.idl
```

- For a **stand-alone installation**, run:

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


- For an **installation from binaries**, run:

```
fastddsgen.bat -example CMake HelloWorld.idl
```

Warning: The colcon installation does not build the `fastddsgen.jar` file although it does download the Fast DDS-Gen repository. The following commands must be executed to build the Java executable:

```
cd <path-to-Fast-DDS-workspace>/src/fastddsgen
gradle assemble
```

Build the Fast DDS application

Then, compile the generated code executing the following commands from the *FastDDSGenHelloWorld* directory.

- On Linux:

```
cd build
cmake ..
make
```

- On Windows:

```
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
cmake --build .
```

Run the Fast DDS application

The application build can be used to spawn any number of publishers and subscribers associated with the topic.

- On Linux:

```
./HelloWorld publisher
./HelloWorld subscriber
```

- On Windows:

```
HelloWorld.exe publisher
HelloWorld.exe subscriber
```

Each time <Enter> is pressed on the Publisher, a new datagram is generated, sent over the network and received by Subscribers currently online. If more than one subscriber is available, it can be seen that the message is equally received on all listening nodes.

The values on the custom IDL-generated data type can also be modified as indicated below.

```
HelloWorld sample;      //Auto-generated container class for topic data from Fast DDS-Gen
sample.msg("Hello there!");    // Add contents to the message
data_writer->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.

7.37.7 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.

7.38 Building Python auxiliary libraries

eProsima Fast DDS-Gen can generate the required source files and CMake project to build the Python modules that allow the use of the IDL defined data types within a *Fast DDS* Python-based application. Each IDL file will result in a new Python module that will contain all the data types defined in the file. The Python binding is generated building the provided solution using [SWIG](#).

Calling *eProsima Fast DDS-Gen* with the option `-python` will generate these files. *eProsima Fast DDS-Gen* will generate a `.i` file which will be processed by [SWIG](#) and a CMake project to call [SWIG](#) first generating C++ files (for connecting C++ and Python) and Python files (Python module for your type) and then compiling the C++ sources.

Note: The Python bindings does not support the use of namespaces.

Before calling CMake, the *Building* process needs several *Dependencies* to be met.

7.38.1 Dependencies

SWIG

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

Note: More recent [SWIG](#) releases are not yet supported. Please, ensure to be using [SWIG](#) 4.0.

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

```
sudo apt install swig
```

Header files and static library for Python

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

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

7.38.2 Building

Call CMake:

```
mkdir build
cd build
cmake ..
cmake --build .
```

This will create the Python files (.py) with the modules (one per each IDL file) that have to be imported within the Python script.

Note: The python bindings does not support using different modules in the same idl file. Split them in different files for expected behavior.

7.39 Defining a data type via IDL

This section describes the data types that can be defined using IDL files, as well as other mechanisms for building data types using IDL files.

- *Supported IDL types*
 - *Primitive types*
 - *Arrays*
 - *Sequences*
 - *Maps*
 - *Structures*
 - *Unions*
 - *Bitsets*
 - *Enumerations*
 - *Bitmasks*
 - *Modules*
 - *Data types with a key*
- *Including other IDL files*
- *Annotations*

- *Forward declaration*
- *IDL 4.2 aliases*
- *IDL 4.2 comments*

7.39.1 Supported IDL types

Be aware that *Fast DDS-Gen* is not case sensitive as it is specified in the [IDL specification](#). To activate case sensitivity use option `-cs` when running *Fast DDS-Gen* (see [Supported options](#)).

Primitive types

The following table shows the basic IDL types supported by *Fast DDS-Gen* and how they are mapped to C++11.

IDL	C++11
char	char
octet	uint8_t
short	int16_t
unsigned short	uint16_t
long	int32_t
unsigned long	uint32_t
long long	int64_t
unsigned long long	uint64_t
float	float
double	double
long double	long double
boolean	bool
string	std::string

Arrays

Fast DDS-Gen supports unidimensional and multidimensional arrays. Arrays are always mapped to `std::array` containers. The following table shows the array types supported and their mapping.

IDL	C++11
char a[5]	std::array<char,5> a
octet a[5]	std::array<uint8_t,5> a
short a[5]	std::array<int16_t,5> a
unsigned short a[5]	std::array<uint16_t,5> a
long a[5]	std::array<int32_t,5> a
unsigned long a[5]	std::array<uint32_t,5> a
long long a[5]	std::array<int64_t,5> a
unsigned long long a[5]	std::array<uint64_t,5> a
float a[5]	std::array<float,5> a
double a[5]	std::array<double,5> a

Sequences

Fast DDS-Gen supports sequences, which map into the `std::vector` container. The following table represents how the map between IDL and C++11 is handled.

IDL	C++11
<code>sequence<char></code>	<code>std::vector<char></code>
<code>sequence<octet></code>	<code>std::vector<uint8_t></code>
<code>sequence<short></code>	<code>std::vector<int16_t></code>
<code>sequence<unsigned short></code>	<code>std::vector<uint16_t></code>
<code>sequence<long></code>	<code>std::vector<int32_t></code>
<code>sequence<unsigned long></code>	<code>std::vector<uint32_t></code>
<code>sequence<long long></code>	<code>std::vector<int64_t></code>
<code>sequence<unsigned long long></code>	<code>std::vector<uint64_t></code>
<code>sequence<float></code>	<code>std::vector<float></code>
<code>sequence<double></code>	<code>std::vector<double></code>

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
<code>map<char, unsigned long long></code>	<code>std::map<char, uint64_T></code>

Note: Only *Primitive types* are currently supported.

Structures

You can define an IDL structure with a set of members with multiple types. It will be converted into a C++ class in which the members of the structure defined via IDL are mapped to private data members of the class. Furthermore, `set()` and `get()` member functions are created to access these private data members.

The following IDL structure:

```
struct Structure
{
    octet octet_value;
    long long_value;
    string string_value;
};
```

Would be converted to:

```
class Structure
{
public:
```

(continues on next page)

(continued from previous page)

```

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;
};

```

(continues on next page)

(continued from previous page)

```
class ChildStruct : public ParentStruct
{
    long child_member;
};
```

Optional members

A member of a structure can be optional. This is achieved by writing the `@optional` annotation before the member.

```
struct StructWithOptionalMember
{
    @optional octet octet_opt;
};
```

An optional member is converted into a template class `eprosima::fastcdr::optional<T>`, where `T` is the member's type.

```
class StructWithOptionalMember
{
    eprosima::fastcdr::optional<octet> octet_opt;
};
```

Before reading the value of the optional member, it should be checked the optional contains a value using `has_value()` function. Accessing a *null* optional throws a `eprosima::fastcdr::exception::BadOptionalAccessException` exception.

```
if (octet_opt.has_value())
{
    octet oc = octet_opt.value();
}
```

Extensibility

In order to support evolving types without breaking interoperability, the concept of type extensibility is supported by *Fast DDS-Gen*. There are three extensibility kinds: *final*, *appendable* and *mutable*.

- *FINAL* extensibility indicates that the type is strictly defined. It is not possible to add members while maintaining type assignability.
- *APPENDABLE* extensibility indicates that two types, where one contains all of the members of the other plus additional members appended to the end, may remain assignable.
- *MUTABLE* extensibility indicates that two types may differ from one another in the additional, removal, and/or transposition of members while remaining assignable.

```
@extensibility(FINAL)
struct FinalStruct
{
    octet octet_opt;
};
```

(continues on next page)

(continued from previous page)

```

@extensibility(APPENDABLE)
struct AppendableStruct
{
    octet octet_opt;
};

@extensibility(MUTABLE)
struct MutableStruct
{
    octet octet_opt;
};

```

Note: XCDRV1 encoding algorithm is not able to manage correctly the deserialization of an appendable structure when it is used as a member of another one.

Unions

In IDL, a union is defined as a sequence of members with their own types and a discriminant that specifies which member is in use. An IDL union type is mapped as a C++ class with member functions to access the union members and the discriminant.

The following IDL union:

```

union Union switch(long)
{
    case 1:
        octet octet_value;
    case 2:
        long long_value;
    case 3:
        string string_value;
};

```

Would be converted to:

```

class Union
{
public:

    Union();
    ~Union();
    Union(
        const Union& x);
    Union(
        Union&& x);
    Union& operator =(
        const Union& x);
    Union& operator =(
        Union&& x);

```

(continues on next page)

(continued from previous page)

```

void d(
    int32_t __d);
int32_t _d() const;
int32_t& _d();

void octet_value(
    uint8_t _octet_value);
uint8_t octet_value() const;
uint8_t& octet_value();
void long_value(
    int64_t _long_value);
int64_t long_value() const;
int64_t& long_value();
void string_value(
    const std::string
    & _string_value);
void string_value(
    std::string&& _string_value);
const std::string& string_value() const;
std::string& string_value();

private:

    int32_t m__d;
    uint8_t m_octet_value;
    int64_t m_long_value;
    std::string m_string_value;
};

```

Bitsets

Bitsets are a special kind of structure, which encloses a set of bits. A bitset can represent up to 64 bits. Each member is defined as *bitfield* and eases the access to a part of the bitset.

For example:

```

bitset MyBitset
{
    bitfield<3> a;
    bitfield<10> b;
    bitfield<12, long> c;
};

```

The type `MyBitset` will store a total of 25 bits (3 + 10 + 12) and will require 32 bits in memory (lowest primitive type to store the bitset's size).

- The bitfield 'a' allows us to access to the first 3 bits (0..2).
- The bitfield 'b' allows us to access to the next 10 bits (3..12).
- The bitfield 'c' allows us to access to the next 12 bits (13..24).

The resulting C++ code will be similar to:

```
class MyBitset
{
public:
    void a(
        char _a);
    char a() const;

    void b(
        uint16_t _b);
    uint16_t b() const;

    void c(
        int32_t _c);
    int32_t c() const;

private:
    std::bitset<25> m_bitset;
};
```

Internally, it is stored as a `std::bitset`. For each bitfield, `get()` and `set()` member functions are generated with the smaller possible primitive unsigned type to access it. In the case of bitfield 'c', the user has established that this accessing type will be long, so the generated code uses `int32_t` instead of automatically use `uint16_t`.

Bitsets can inherit from other bitsets, extending their member set.

```
bitset ParentBitset
{
    bitfield<3> parent_member;
};

bitset ChildBitset : ParentBitset
{
    bitfield<10> child_member;
};
```

In this case, the resulting C++ code will be:

```
class ParentBitset
{
    std::bitset<3> parent_member;
};

class ChildBitset : public ParentBitset
{
    std::bitset<10> child_member;
};
```

Note that in this case, `ChildBitset` will have two `std::bitset` data members, one belonging to `ParentBitset` and the other belonging to `ChildBitset`.

Enumerations

An enumeration in IDL format is a collection of identifiers that have an associated numeric value. An IDL enumeration type is mapped directly to the corresponding C++11 enumeration definition.

The following IDL enumeration:

```
enum Enumeration
{
    RED,
    GREEN,
    BLUE
};
```

Would be converted to:

```
enum Enumeration : uint32_t
{
    RED,
    GREEN,
    BLUE
};
```

Bitmasks

Bitmasks are a special kind of Enumeration to manage masks of bits. It allows defining bit masks based on their position.

The following IDL bitmask:

```
@bit_bound(8)
bitmask MyBitMask
{
    @position(0) flag0,
    @position(1) flag1,
    @position(4) flag4,
    @position(6) flag6,
    flag7
};
```

Would be converted to:

```
enum MyBitMask : uint8_t
{
    flag0 = 0x01 << 0,
    flag1 = 0x01 << 1,
    flag4 = 0x01 << 4,
    flag6 = 0x01 << 6,
    flag7 = 0x01 << 7
};
```

The annotation `bit_bound` defines the width of the associated enumeration. It must be a positive number between 1 and 64. If omitted, it will be 32 bits. For each `flag`, the user can use the annotation `position` to define the position of the flag. If omitted, it will be auto incremented from the last defined flag, starting at 0.

Modules

In order to avoid collision between variable names, modules can be defined within the IDL file. A module would be converted into a namespace in C++.

Data types with a key

In order to use keyed topics, the user should define some key members inside the structure. This is achieved by writing the `@key` annotation before the members of the structure that are used as keys. For example in the following IDL file the `id` and `type` field would be the keys:

```
struct MyType
{
    @key long id;
    @key string type;
    long positionX;
    long positionY;
};
```

Fast DDS-Gen automatically detects these tags and correctly generates the serialization methods for the key generation function in `TopicDataType (getKey())`. This function will obtain the 128-bit MD5 digest of the big-endian serialization of the Key Members.

7.39.2 Including other IDL files

Other IDL files can be included in addition to the current IDL file. *Fast DDS-Gen* uses a C/C++ preprocessor for this purpose, and `#include` directive can be used to include an IDL file. Preprocessor directives guarding against multiple inclusion of the same IDL file are also advisable.

```
#include "OtherFile.idl"
#include <AnotherFile.idl>
```

If *Fast DDS-Gen* does not find a C/C++ preprocessor in default system paths, the preprocessor path can be specified using parameter `-ppPath`. The parameter `-ppDisable` can be used to disable the usage of the C/C++ preprocessor.

7.39.3 Annotations

The application allows the user to define and use their own annotations as defined in the [OMG IDL specification](#).

```
@annotation MyAnnotation
{
    long value;
    string name;
};
```

Additionally, the following standard annotations are defined by the specification and considered builtin (these annotations might be applied without the need of defining them).

Builtin Annotation	Behavior
@ami	Asynchronous interface or operation.

Builtin Annotation	Behavior
@appendable	Shortcut for @extensibility(APPENDABLE).
@autoid	Member ID algorithm configuration if there is no member ID explicitly set using @id annotation. Possi
@bit_bound	Set number of bits on <i>bitmasks</i> and <i>enumerations</i> underlying primitive type. Currently, @bit_bound ca
@data_representation	Specify the DataRepresentationId required for a specific type.
@default	Set constant default value to a member.
@default_literal	Mark an <i>enumerations</i> literal as default.
@default_nested	Use in module declaration to mark as @nested every element defined within the module.
@extensibility	Applicable to any constructed element. Specify how the element is allowed to evolve. Please, refer to <i>E</i>
@external	Member is stored in external storage.
@final	Shortcut for @extensibility(FINAL)
@hashid	Calculate the member ID with the string provided or, if empty, with the member name.
@id	Assign member ID to a structure or union member.
@ignore_literal_names	When checking evolved type compatibility, take or not into account member names.
@key	Mark a structure member as part of the key. @Key is also supported. Please, refer to <i>Topics, keys and in</i>
@max	Set a maximum constant value to the member.
@min	Set a minimum constant value to the member.
@must_understand	Mark a structure member as essential for the structure cohesion.
@mutable	Shortcut for @extensibility(MUTABLE)
@nested	Type is always used within another one.
@non_serialized	Omit member during serialization.
@oneway	One-way operation, flowing the information only on one direction.
@optional	Configure a structure member as optional. Please refer to <i>Optional Members</i> for more information.
@position	Set a bitflag position in <i>bitmasks</i> .
@range	Set a range of allowed values for the member.
@service	Interface is to be treated as a service.
@topic	Structure or union is meant to be used as Topic Data Type.
@try_construct	When checking evolved type compatibility, configure the behavior for collection/aggregated types const
@unit	Specify a unit of measurement for the member.
@value	Set constant value to <i>enumerations</i> literal.
@verbatim	Add comment or text to the element.

7.39.4 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:

```

(continues on next page)

(continued from previous page)

```
};
    string empty;
```

7.39.5 IDL 4.2 aliases

IDL 4.2 allows using the following names for primitive types:

int8
uint8
int16
uint16
int32
uint32
int64
uint64

7.39.6 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 MyStruct
{
    string mymessage;    // mymessage data member.
};
```

7.40 CLI

The *Fast DDS* command line interface provides a set commands and sub-commands to perform, *Fast DDS* related, maintenance and configuration tasks.

- *discovery*
 - *How to run*
 - *Examples*
- *shm*
- *xml*

An executable file for Linux and Windows that runs the *Fast DDS CLI* application is available in the *tools* folder. If the *tools/fastdds* folder path is added to the *PATH*, or by sourcing the `<path/to/fastdds>/install/setup.bash` configuration file, *Fast DDS CLI* can be executed running the following commands:

- Linux:

```
$ fastdds <command> [<command-args>]
```

- Windows:

```
> fastdds.bat <command> [<command-args>]
```

There are three verbs whose functionality is described in the following table:

Verbs	Description
discovery	Launches a server for <i>Discovery Server</i> .
shm	Allows manual cleaning of garbage files that may be generated by <i>Shared Memory Transport</i>
xml	Checks if a xml profile is well formed.

7.40.1 discovery

This command launches a SERVER (or BACKUP) for *Discovery Server*. This *server* will manage the discovery phases of the CLIENTS which are connected to it. *Clients* must know how to reach the *server*, which is accomplished by specifying an IP address, the *servers* GUID prefix, and a transport protocol like UDP or TCP. *Servers* do not need any prior knowledge of their *clients*, but require a GUID prefix, and the listening IP address where they may be reached. For more information on the different *Fast DDS* discovery mechanisms and how to configure them, please refer to *Discovery*.

Important: It is possible to interconnect *servers* (or *backup servers*) instantiated with `fastdds discovery` using environment variable `ROS_DISCOVERY_SERVER` (see *ROS_DISCOVERY_SERVER*) or a XML configuration file.

How to run

On a shell, execute:

```
fastdds discovery -i {0-255} [optional parameters]
```

Where the parameters are:

Option	Description
-h	Produce help message.
-help	
-i	Unique server identifier. Specifies zero based server position in ROS_DISCOVERY_SERVER environment variable. Must be an integer in range [0, 255] If not specified, it must be defined using a XML configuration file.
--serve	
-l	IPv4/IPv6 address chosen to listen the clients using UDP transport. Defaults to any (0.0.0.0/::0). Instead of an address, a DNS domain name can be specified.
--udp-a	
-p	UDP port chosen to listen the clients. Defaults to '11811'.
--udp-p	
-t	IPv4/IPv6 address chosen to listen the clients using TCP transport. Instead of an address, a DNS domain name can be specified. Defaults to localhost (127.0.0.1).
--tcp-a	
-q	TCP port chosen to listen the clients. Defaults to '42100'. Only one server can be configured using the default port.
--tcp-p	
-b	Creates a <i>BACKUP server</i> (see Discovery Protocol)
--backu	
-x	XML configuration file (see XML profiles). In this case, the default configuration file is not loaded. The CLI options override XML configuration for that specific parameter. The default profile in the XML file is loaded except if a specific profile name is specified: <code>profile_name@xml_file</code>
--xml-f	

The output is:

```
### Server is running ###
Participant Type: <SERVER|BACKUP>
Security: <YES|NO>
Server ID: <server-id>
Server GUID prefix: 44.53.<server-id-in-hex>.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  UDPv4: [<ip-address>]:<port>
                   UDPv6: [<ip-address>]:<port>
                   TCPv4: [<ip-address>]:<physical-port>-<logical-port>
                   TCPv6: [<ip-address>]:<physical-port>-<logical-port>
```

Once the *server* is instantiated, the *clients* can be configured either programmatically or by XML (see [Discovery Server Settings](#)), or using environment variable ROS_DISCOVERY_SERVER (see [ROS_DISCOVERY_SERVER](#))

Note: The [Security](#) configuration of the discovery server should be done through XML. See example below.

Examples

1. Launch a **default server** with id 0 (first on ROS_DISCOVERY_SERVER) listening on all available interfaces on UDP port '11811'. Only one server can use default values per machine.

```
fastdds discovery -i 0
```

Output:

```
### Server is running ###
Participant Type:  SERVER
Security:         NO
Server ID:        0
```

(continues on next page)

(continued from previous page)

```
Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
Server Addresses:   UDPv4:[0.0.0.0]:11811
```

2. Launch a default server with id 1 (second on ROS_DISCOVERY_SERVER) listening on localhost with UDP port 14520. Only localhost clients can reach the server defining as `ROS_DISCOVERY_SERVER=;127.0.0.1:14520`.

```
fastdds discovery -i 1 -l 127.0.0.1 -p 14520
```

Output:

```
### Server is running ###
Participant Type:  SERVER
Security:         NO
Server ID:        1
Server GUID prefix: 44.53.01.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  UDPv4:[127.0.0.1]:14520
```

This same output can be obtained loading the following XML configuration file `DiscoveryServerCLI.xml`:

```
<participant profile_name="participant_profile_discovery_server_cli" is_default_
->profile="true">
  <rtps>
    <prefix>44.53.01.5f.45.50.52.4f.53.49.4d.41</prefix>
    <builtin>
      <discovery_config>
        <discoveryProtocol>SERVER</discoveryProtocol>
      </discovery_config>
      <metatrafficUnicastLocatorList>
        <locator>
          <udp4>
            <address>localhost</address>
            <port>14520</port>
          </udp4>
        </locator>
      </metatrafficUnicastLocatorList>
    </builtin>
  </rtps>
</participant>

<participant profile_name="second_participant_profile_discovery_server_cli">
  <rtps>
    <prefix>44.53.02.5f.45.50.52.4f.53.49.4d.41</prefix>
    <builtin>
      <discovery_config>
        <discoveryProtocol>SERVER</discoveryProtocol>
      </discovery_config>
      <metatrafficUnicastLocatorList>
        <locator>
          <udp4>
            <address>192.168.36.34</address>
            <port>8783</port>
          </udp4>
        </locator>
```

(continues on next page)

(continued from previous page)

```

        <locator>
          <udp4>
            <address>172.20.96.1</address>
            <port>51083</port>
          </udp4>
        </locator>
      </metatrafficUnicastLocatorList>
    </builtin>
  </rtsp>
</participant>

<participant profile_name="secure_discovery_server_cli">
  <rtsp>
    <prefix>44.53.00.5f.45.50.52.4f.53.49.4d.41</prefix>
    <builtin>
      <discovery_config>
        <discoveryProtocol>SERVER</discoveryProtocol>
      </discovery_config>
      <metatrafficUnicastLocatorList>
        <locator>
          <udp4>
            <address>0.0.0.0</address>
            <port>11811</port>
          </udp4>
        </locator>
      </metatrafficUnicastLocatorList>
    </builtin>
    <propertiesPolicy>
      <properties>
        <!-- Activate Auth:PKI-DH plugin -->
        <property>
          <name>dds.sec.auth.plugin</name>
          <value>builtin.PKI-DH</value>
        </property>

        <!-- Configure Auth:PKI-DH plugin -->
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.identity_ca</name>
          <value>file://maincacert.pem</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.identity_certificate</name>
          <value>file://appcert.pem</value>
        </property>
        <property>
          <name>dds.sec.auth.builtin.PKI-DH.private_key</name>
          <value>file://appkey.pem</value>
        </property>

        <!-- Activate Access:Permissions plugin -->
        <property>
          <name>dds.sec.access.plugin</name>

```

(continues on next page)

(continued from previous page)

```

        <value>builtin.Access-Permissions</value>
      </property>

      <!-- Configure Access:Permissions plugin -->
      <property>
        <name>dds.sec.access.builtin.Access-Permissions.permissions_ca</
↪name>
        <value>file://maincacet.pem</value>
      </property>
      <property>
        <name>dds.sec.access.builtin.Access-Permissions.governance</
↪name>
        <value>file://governance.smime</value>
      </property>
      <property>
        <name>dds.sec.access.builtin.Access-Permissions.permissions</
↪name>
        <value>file://permissions.smime</value>
      </property>

      <!-- Activate Crypto:AES-GCM-GMAC plugin -->
      <property>
        <name>dds.sec.crypto.plugin</name>
        <value>builtin.AES-GCM-GMAC</value>
      </property>
    </properties>
  </propertiesPolicy>
</rtps>
</participant>

```

```
fastdds discovery -x [PATH_TO_FILE]/DiscoveryServerCLI.xml
```

3. Launch a default server with id 1 (second on ROS_DISCOVERY_SERVER) listening on IPv6 address 2a02:ec80:600:ed1a::3 with UDP port 14520.

```
fast-discovery-serverd-1.0.1.exe -i 1 -l 2a02:ec80:600:ed1a::3 -p 14520
```

Output:

```

### Server is running ###
Participant Type:  SERVER
Security:         NO
Server ID:        1
Server GUID prefix: 44.53.01.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  UDPv6:[2a02:ec80:600:ed1a::3]:14520

```

4. Launch a default server with id 2 (third on ROS_DISCOVERY_SERVER) listening on WiFi (192.168.36.34) and Ethernet (172.20.96.1) local interfaces with UDP ports 8783 and 51083 respectively (addresses and ports are made up for the example).

```
fastdds discovery -i 2 -l 192.168.36.34 -p 8783 -l 172.20.96.1 -p 51083
```

Output:

```
### Server is running ###
Participant Type    SERVER
Security:          NO
Server ID:         2
Server GUID prefix: 44.53.02.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  UDPv4:[192.168.36.34]:8783
                   UDPv4:[172.20.96.1]:51083
```

Using the same XML configuration file from the second example, the same output can be obtained loading a specific profile: *second_participant_profile_discovery_server_cli*.

```
fastdds discovery -x second_participant_profile_discovery_server_cli@[PATH_TO_FILE]/
↪DiscoveryServerCLI.xml
```

5. Launch a default server with id 3 (fourth on ROS_DISCOVERY_SERVER) listening on 172.30.144.1 with UDP port 12345 and provided with a backup file. If the server crashes it will automatically restore its previous state when re-enacted.

```
fastdds discovery -i 3 -l 172.30.144.1 -p 12345 -b
```

Output:

```
### Server is running ###
Participant Type    BACKUP
Security:          NO
Server ID:         3
Server GUID prefix: 44.53.03.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  UDPv4:[172.30.144.1]:12345
```

6. Launch a default server with id 0 (first on ROS_DISCOVERY_SERVER) listening on localhost with UDP port 14520. Only localhost clients can reach the server defining as *ROS_DISCOVERY_SERVER=localhost:14520*.

```
fastdds discovery -i 0 -l localhost -p 14520
```

Output:

```
### Server is running ###
Participant Type:   SERVER
Security:          NO
Server ID:         0
Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  UDPv4:[127.0.0.1]:14520
```

7. Launch a secure server with id 0 (first on ROS_DISCOVERY_SERVER) listening on all available interfaces on UDP port '11811'.

```
fastdds discovery -x secure_discovery_server_cli@[PATH_TO_FILE]/DiscoveryServerCLI.
↪xml
```

Output:

```
### Server is running ###
Participant Type:   SERVER
Security:          YES
```

(continues on next page)

(continued from previous page)

```

Server ID:          0
Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
Server Addresses:   UDPv4:[0.0.0.0]:11811

```

8. Launch a server with id 0 (first on ROS_DISCOVERY_SERVER) reading specific *profile_name* configuration from XML file.

```
fastdds discovery -i 0 -x profile_name@[PATH_TO_FILE]/config.xml
```

Output:

```

### Server is running ###
Participant Type:  SERVER
Security:          NO
Server ID:         0
Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  UDPv4:[127.0.0.1]:56542

```

9. Launch a server with id 0 (first on ROS_DISCOVERY_SERVER) listening on localhost on default TCP port '42100'.

```
fastdds discovery -i 0 -t 127.0.0.1
```

Output:

```

### Server is running ###
Participant Type:  SERVER
Security:          NO
Server ID:         0
Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  TCPv4:[127.0.0.1]:42100-42100

```

10. Launch a server with id 0 (first on ROS_DISCOVERY_SERVER) listening on localhost and WiFi (192.163.6.34). Two TCP ports need to be specified because TCP transports cannot share ports.

```
fastdds discovery -i 0 -t 127.0.0.1 -q 42100 -t 192.163.6.34 -q 42101
```

Output:

```

### Server is running ###
Participant Type:  SERVER
Security:          NO
Server ID:         0
Server GUID prefix: 44.53.00.5f.45.50.52.4f.53.49.4d.41
Server Addresses:  TCPv4:[127.0.0.1]:42100-42100
                  TCPv4:[192.163.6.34]:42101-42101

```

Note: When using Discovery Server over TCP, the first port shown in the output refers to the TCP Physical port and the second one to the TCP Logical port (see [TCP Transport](#)).

Note: A server can be instantiated just by passing the port arguments `-p` and `-q`. Fast DDS CLI will use the default values of the IP addresses, that is, `0.0.0.0` for UDP and `127.0.0.1` for TCP.

7.40.2 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.

7.40.3 xml

Checks if a given xml profile is well formed, by matching it against a XSD schema. If the given input to the command is a path to a folder instead of a path to a file, all xml files contained in the folder will be validated.

This validation consists in checking the lack of parameters, values bounds, expected values data types and main profile structure. For further information see *Creating an XML profiles file*.

```
fastdds xml [<xml-command>]
```

Sub-command	Description
validate	Checks a xml profile by matching it against a XSD schema.

Option	Description
-h --help	Produce help message.
-d --debug	Print debug information (disabled by default)
-x --xsd_file	XSD schema for validation (not required, Fast DDS schema is used by default)

Example

```
fastdds xml validate my_profile.xml
```

7.41 Docker Images

eProsima provides the Fast DDS and the Fast DDS Suite Docker images for those who want a quick demonstration of Fast DDS running on an Ubuntu platform. They can be downloaded from [eProsima's downloads page](#).

This Docker images were built for Ubuntu 22.04 (Jammy Jellyfish).

To run a container you need Docker installed. From a terminal, run:

```
sudo apt install docker.io
```

7.41.1 Leveraging Fast DDS SHM in Docker deployments

By default, *Fast DDS* enables both a *Shared Memory Transport* and *Data-sharing delivery* (when the configuration allows it, see *Constraints*). The way *Fast DDS* utilizes to find out whether a remote *DomainParticipant* is running on the same host as the local one is through a hashing of the network interfaces, which entails that two Docker container which are not sharing the same network stack will be detected as two different hosts, and therefore *Fast DDS* will defer to communicating through the *UDP Transport* instead.

To enable the use of both the *Shared Memory Transport* and *Data-sharing* in Docker deployments, two additional options need to be passed to the *docker run* command, those are:

- `--network=host`: This option shares the host's network stack with the containers, and therefore *Fast DDS* will be able to identify them as the same host.
- `--ipc=host`: This option shares the host's shared memory mechanism with the containers. Without it, *Fast DDS* will identify both containers as the same host, but since they will have separate shared memory spaces, they will not be able to communicate with one another. The use of this option is the Docker recommended way of sharing shared memory between containers, in opposition of for instance sharing the `/dev/shm` volume in Linux machines. A more advanced user could set the flag to *shared*, thus sharing the shared memory mechanism of one of the container with the others.

```
docker run -it --rm --network=host --ipc=host [OPTIONS] <docker-image>
```

7.41.2 Fast DDS Image

This Docker image contains the Fast DDS library and its dependencies, ready to be used in a final user application. This includes:

- *eProsima Fast DDS libraries and examples*: Fast DDS libraries bundled with several examples that showcase a variety of capabilities of eProsima's Fast DDS implementation.

To load this image into your Docker repository, from a terminal, run:

```
docker load -i "ubuntu-fastdds <FastDDS-Version>.tar"
```

You can run this Docker container as follows:

```
docker run -it ubuntu-fastdds:<FastDDS-Version>
```

From the resulting Bash Shell you can run each feature.

Fast DDS Examples

Included in this Docker container is a set of binary examples that showcase several functionalities of the Fast DDS libraries. These examples' path can be accessed from a terminal by typing:

```
goToExamples
```

From this folder, you can access all examples, both for DDS and RTPS layers.

Hello World Example

This is a minimal example that will perform a Publisher/Subscriber match and start sending samples.

```
goToExamples
cd dds/HelloWorldExample/bin
tmux new-session "./DDSHelloWorldExample publisher 0 1000" \; \
    split-window "./DDSHelloWorldExample subscriber" \; \
    select-layout even-vertical
```

This example is not constrained to the current instance. It's possible to run several instances of this container to check the communication between them by running the following from each container.

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample publisher
```

or

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample subscriber
```

Benchmark Example

This example creates either a Publisher or a Subscriber and on a successful match starts sending samples. After a few seconds the process that launched the Publisher will show a report with the number of samples transmitted.

On the subscriber side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark subscriber udp
```

On the publisher side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark publisher udp
```

7.41.3 Fast DDS Suite Image

This Docker image contains the complete Fast DDS suite. This includes:

- *eProsima Fast DDS libraries and examples*: Fast DDS libraries bundled with several examples that showcase a variety of capabilities of eProsima's Fast DDS implementation.
- *Shapes Demo*: eProsima Shapes Demo is an application in which Publishers and Subscribers are shapes of different colors and sizes moving on a board. Each shape refers to its own topic: Square, Triangle or Circle. A single instance of the eProsima Shapes Demo can publish on or subscribe to several topics at a time.

You can read more about this application on the [Shapes Demo documentation page](#).

- **Fast DDS Monitor:** eProsima Fast DDS Monitor is a graphical desktop application aimed at monitoring DDS environments deployed using the *eProsima Fast DDS* library. Thus, the user can monitor in real time the status of publication/subscription communications between DDS entities. They can also choose from a wide variety of communication parameters to be measured (latency, throughput, packet loss, etc.), as well as record and compute in real time statistical measurements on these parameters (mean, variance, standard deviation, etc.).

You can read more about this application on the [Fast DDS Monitor documentation page](#).

- **DDS Router:** eProsima DDS Router is an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of eProsima DDS Router. This is achieved by deploying a DDS Router on an edge device of each local network so that the DDS Router routes DDS traffic from one network to the other through WAN communication.

You can read more about this application on the [DDS Router documentation website](#).

- **Plotjuggler eProsima Edition:** eProsima Fast DDS Visualizer Plugin is a plugin for the *PlotJuggler* application. PlotJuggler is a graphical desktop application providing visualization features of data series, time series, X-Y plots. It also adds data management features, such as data import and export, custom and built-in data manipulation functions, data series merges, etc. Also, this software supports many different layouts, with dynamic, rich and user-friendly customization.

You can read more about this application on the [Plotjuggler eProsima Edition documentation website](#).

To load this image into your Docker repository, from a terminal run

```
docker load -i "ubuntu-fastdds-suite <FastDDS-Version>.tar"
```

You can run this Docker container as follows

```
xhost local:root
docker run -it --privileged -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix \
ubuntu-fastdds-suite:<FastDDS-Version>
```

From the resulting Bash Shell you can run each feature.

Fast DDS Examples

Included in this Docker container is a set of binary examples that showcase several functionalities of the Fast DDS libraries. These examples' path can be accessed from a terminal by typing

```
goToExamples
```

From this folder you can access all examples, both for DDS and RTPS. We detail the steps to launch two such examples below.

Hello World Example

This is a minimal example that will perform a Publisher/Subscriber match and start sending samples.

```
goToExamples
cd dds/HelloWorldExample/bin
tmux new-session "./DDSHelloWorldExample publisher 0 1000" \; \
    split-window "./DDSHelloWorldExample subscriber" \; \
    select-layout even-vertical
```

This example is not constrained to the current instance. It's possible to run several instances of this container to check the communication between them by running the following from each container.

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample publisher
```

or

```
goToExamples
cd dds/HelloWorldExample/bin
./DDSHelloWorldExample subscriber
```

Benchmark Example

This example creates either a Publisher or a Subscriber and on a successful match starts sending samples. After a few seconds the process that launched the Publisher will show a report with the number of samples transmitted.

On the subscriber side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark subscriber udp
```

On the publisher side, run:

```
goToExamples
cd dds/Benchmark/bin
./DDSBenchmark publisher udp
```

Shapes Demo

To launch the Shapes Demo, from a terminal run

```
ShapesDemo
```

eProsima Shapes Demo usage information can be found on the [Shapes Demo documentation](#).

Fast DDS Monitor

To launch the Fast DDS Monitor, from a terminal run

```
fastdds_monitor
```

eProsima Fast DDS Monitor user manual can be found on the [Fast DDS Monitor documentation](#).

DDS Router

This example configures a DDS Router to communicate a publisher and subscriber running in different DDS Domains.

Run the following command to create the DDS Router *yml* configuration file (*/config.yml*).

```
echo "version: v2.0
participants:
- name: simple_dds_participant_0
  kind: local
  domain: 0
- name: simple_dds_participant_1
  kind: local
  domain: 1" > /config.yml
```

Then execute the following command to run the Publisher in Domain 0, the Subscriber in Domain 1, and the DDS Router communicating both Domains.

```
goToExamples
cd dds/BasicConfigurationExample/bin
tmux new-session \
    "ddsrouter --config-path /config.yml" \; \
    split-window -h "./BasicConfigurationExample publisher --domain 0 --interval 1000 --
↪transport udp" \; \
    split-window -v "./BasicConfigurationExample subscriber --domain 1 --transport udp"
```

eProsima DDS Router usage information can be found on the [DDS Router documentation](#).

PlotJuggler eProsima Edition

To launch the PlotJuggler eProsima Edition, from a terminal run

```
plotjuggler
```

eProsima PlotJuggler eProsima Edition usage information can be located on the [PlotJuggler eProsima Edition User Manual](#).

7.42 Dependencies and compatibilities

Fast DDS is continuously evolving and improving. This means that the different software products that are part of the Fast DDS ecosystem are evolving and improving together with Fast DDS. This section provides information about the required dependencies for building Fast DDS, as well as about the versions of the eProsima software products related to Fast DDS.

7.42.1 Platform support

This following definitions reflects the level of support offered by **eprosima Fast DDS** on different platforms:

- **Tier 1:** these platforms are subjected to our unit test suite and other testing tools on a frequent basis including continuous integration jobs, nightly jobs, packaging jobs, and performance testing. Errors or bugs discovered in these platforms are prioritized for correction by the development team. Significant errors discovered in Tier 1 platforms can impact release dates and we strive to resolve all known high priority errors in Tier 1 platforms prior to new version releases.
- **Tier 2:** these platforms are subject to periodic CI testing which runs both builds and tests with publicly accessible results. The CI is expected to be run at least within a week of relevant changes for the current release of *Fast DDS*. Installation instructions should be available and up-to-date in order for a platform to be listed in this category. Package-level binary packages may not be provided but providing a downloadable archive of the built workspace is encouraged. Errors may be present in released product versions for Tier 2 platforms. Known errors in Tier 2 platforms will be addressed subject to resource availability on a best effort basis and may or may not be corrected prior to new version releases. One or more entities should be committed to continuing support of the platform.
- **Tier 3:** these platforms are those for which community reports indicate that the release is functional. The development team does not run the unit test suite or perform any other tests on platforms in Tier 3. Community members may provide assistance with these platforms.

7.42.2 Build system dependencies

The following table shows the minimum version required of the Fast DDS build system dependencies.

2.14.x

CMake	3.20
--------------	------

OS Architecture	amd64	amd32	arm64
Ubuntu Jammy (22.04)	Tier 1: GCC 11.4 Tier 3: Clang 15	—	Tier 1: GCC 11.4 Tier 3: Clang 15
Ubuntu Focal (20.04)	Tier 3: GCC 9	—	Tier 3: GCC 9
MacOS Mojave (10.14)	Tier 1: Clang 15	—	—
Windows 10	Tier 1: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Studio 2017)	Tier 2: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Studio 2017)	—
Windows 11	Tier 3: MSVC v143 (Visual Studio 2022)	Tier 3: MSVC v143 (Visual Studio 2022)	—
Debian Buster (10)	Tier 3: GCC 8	—	Tier 3: GCC 8
Android 12	Tier 3: SDK 31	—	Tier 3: SDK 31
Android 13	Tier 3: SDK 33	—	Tier 3: SDK 33
QNX 7.1	Tier 3: QCC (over GCC 8.3)	—	Tier 3: QCC (over GCC 8.3)

2.13.x

CMake 3.20

OS Archi- tecture	amd64	amd32	arm64
Ubuntu Jammy (22.04)	Tier 1: GCC 11.4 Tier 3: Clang 15	—	Tier 1: GCC 11.4 Tier 3: Clang 15
Ubuntu Focal (20.04)	Tier 3: GCC 9	—	Tier 3: GCC 9
MacOS Mojave (10.14)	Tier 1: Clang 15	—	—
Windows 10	Tier 1: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Studio 2017)	Tier 2: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Studio 2017)	—
Windows 11	Tier 3: MSVC v143 (Visual Studio 2022)	Tier 3: MSVC v143 (Visual Studio 2022)	—
Debian Buster (10)	Tier 3: GCC 8	—	Tier 3: GCC 8
Android 12	Tier 3: SDK 31	—	Tier 3: SDK 31
Android 13	Tier 3: SDK 33	—	Tier 3: SDK 33
QNX 7.1	Tier 3: QCC (over GCC 8.3)	—	Tier 3: QCC (over GCC 8.3)

2.10.x

CMake 3.16

OS Archi- tecture	amd64	amd32	arm64
Ubuntu Jammy (22.04)	Tier 1: GCC 9 Tier 3: Clang 12	—	Tier 1: GCC 9 Tier 3: Clang 12
Ubuntu Focal (20.04)	Tier 1: GCC 9 Tier 3: Clang 12	—	Tier 1: GCC 9 Tier 3: Clang 12
MacOS Mojave (10.14)	Tier 1: Clang 12	—	—
Windows 10	Tier 1: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Studio 2017)	Tier 2: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Studio 2017)	—
Debian Buster (10)	Tier 3: GCC 8	—	Tier 3: GCC 8
Android 11	Tier 3: SDK 30	—	Tier 3: SDK 30
QNX 7.1	Tier 3: QCC (over GCC 8.3)	—	Tier 3: QCC (over GCC 8.3)

2.6.x

CMake	3.16
--------------	------

OS Archi- tecture	amd64	amd32	arm64
Ubuntu Focal (20.04)	Tier 1: GCC 9 Tier 3: Clang 12	—	Tier 1: GCC 9 Tier 3: Clang 12
MacOS Mojave (10.14)	Tier 1: Clang 12	—	—
Windows 10	Tier 1: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Stu- dio 2017)	Tier 2: MSVC v142 (Visual Studio 2019) Tier 2: MSVC v141 (Visual Stu- dio 2017)	—
Debian Buster (10)	Tier 3: GCC 8	—	Tier 3: GCC 8

7.42.3 Library dependencies

The following table shows the corresponding versions of the Fast DDS library dependencies.

2.14.x

Product	Related version
Fast CDR	v2.2.1
Foonathan Memory Vendor	v1.3.1
Asio	v1.18.1
TinyXML2	v6.0.0
OpenSSL	v3.1.1

2.13.x

Product	Related version
Fast CDR	v2.1.3
Foonathan Memory Vendor	v1.3.1
Asio	v1.18.1
TinyXML2	v6.0.0
OpenSSL	v3.1.1

2.10.x

Product	Related version
Fast CDR	v1.0.27
Foonathan Memory Vendor	v1.3.1
Asio	v1.18.1
TinyXML2	v6.0.0
OpenSSL	v3.1.1

2.6.x

Product	Related version
Fast CDR	v1.0.28
Foonathan Memory Vendor	v1.2.2
Asio	v1.18.1
TinyXML2	v6.0.0
OpenSSL	v1.1.1

7.42.4 eProsima products compatibility

The following table shows the compatibility between the different versions of the eProsima software products that use Fast DDS as the core middleware.

2.14.x

Product	Related version
Fast DDS Gen	v3.3.0
Fast DDS Gen - IDL parser	v3.0.0
Fast DDS python	v1.4.1
Shapes Demo	v2.14.1

2.13.x

Product	Related version
Fast DDS Gen	v3.2.1
Fast DDS Gen - IDL parser	v3.0.0
Fast DDS python	v1.4.0
Shapes Demo	v2.13.4

2.10.x

Product	Related version
Fast DDS Gen	v2.4.0
Fast DDS Gen - IDL parser	v1.5.0
Fast DDS python	v1.2.1
Shapes Demo	v2.10.3

2.6.x

Product	Related version
Fast DDS Gen	v2.1.3
Fast DDS Gen - IDL parser	v1.2.0
Fast DDS python	v1.0.2
Shapes Demo	v2.6.8

Information about the release lifecycle can be found [here](#).

7.43 Version 2.14.1 (latest)

Important: Fast DDS v2.14 will be the last minor version of Fast DDS v2, the next major release will be Fast DDS v3.0.0, stay tuned!

This release includes the following **features** in an ABI compatible manner:

1. Implement *copy_from_topic_qos* method (see [Creating a DataWriter](#) and see [Creating a DataReader](#)).

This release includes the following **improvements**:

1. Set 2.12.x as EOL
2. Use a plain switch for *NetmaskFilterKind operator<<*
3. Set real TCP *non_blocking_send* limitation
4. Increase ack waiting time in *reliable_on_unack_sample_removed*
5. Enforce SHM ports open mode exclusions
6. Force unlimited ResourceLimits if lower or equal to zero (see [ResourceLimitsQoSPolicy](#))
7. Allow processing of AckNack submessages with `count == 0`
8. Refactor *IStatusQueryable* and make monitor service interfaces private
9. Internal refactor on port handling
10. Improve *ThreadSettingsQoS* logging
11. Run *is_plain* method with the corresponding data representation
12. Address compilation issue for GLIBC version
13. Repository and test improvements:
 1. Run Github Ubuntu CI on PRs
 2. Only run PRs CI when a review is requested
 3. Improve filtering of DNS tests
 4. Make *sample_lost_be_dw_be_dr_fragments* test less flaky
 5. Build Fast DDS Python bindings in Fast DDS Docs Github CI job
 6. Add check for XML API to PR template
 7. Refactor Github CI sanitizer related jobs
 8. Increase sleep to miss the deadline in macOS flaky tests

9. Fix nightly sanitizer CI workflow name
10. Improve xml loading method in fuzz_XMLProfiles
11. Build ShapesDemo on Ubuntu Github CI
12. Correctly set branches for reusable-sanitizers-ci

This release includes the following **fixes**:

1. Address XMLProfiles fuzzer regressions
2. Fix flaky Log tests
3. Fix hidden overloaded virtual methods
4. Effectively assert `automatic/manual_by_participant` liveliness
5. Fix `on_sample_lost` notification on best-effort readers for fragmented samples
6. Monitor service properly managing instances
7. Fix CVE-2024-30258
8. Fix Discovery Server over TCP using `LocatorSelectorEntry`
9. Removed warning
10. Make `get_first_untaken_info()` coherent with `read()/take()`
11. Fix serialization of `Data(w)/Data(r)` in `MonitorService`
12. Bugfix in `DDSBlackboxTestsDataReader` test
13. Fix leak in `SecurityManager::participant_volatile_message_secure_writer_`
14. Fix some leaks in XML DynamicTypes Parser
15. Fix support for `@key` annotation in Dynamic types
16. Fix `SecurityTest` unit tests memory error
17. Fix Python Installation version in Github CI. Address failing system tests environment issues

Note: When upgrading to version 2.14.0 it is **advisable** to regenerate generated source from IDL files using [Fast DDS-Gen v3.3.0](#).

See also:

For further information about the corresponding versions of other products related to this Fast DDS version, please refer to the [eProsima products compatibility](#) section.

7.44 Supported versions

7.44.1 Version 2.14

Version 2.14.1 (latest)

Important: Fast DDS v2.14 will be the last minor version of Fast DDS v2, the next major release will be Fast DDS v3.0.0, stay tuned!

This release includes the following **features** in an ABI compatible manner:

1. Implement *copy_from_topic_qos* method (see [Creating a DataWriter](#) and see [Creating a DataReader](#)).

This release includes the following **improvements**:

1. Set 2.12.x as EOL
2. Use a plain switch for *NetmaskFilterKind operator<<*
3. Set real TCP *non_blocking_send* limitation
4. Increase ack waiting time in *reliable_on_unack_sample_removed*
5. Enforce SHM ports open mode exclusions
6. Force unlimited ResourceLimits if lower or equal to zero (see [ResourceLimitsQosPolicy](#))
7. Allow processing of AckNack submessages with `count == 0`
8. Refactor *IStatusQueryable* and make monitor service interfaces private
9. Internal refactor on port handling
10. Improve *ThreadSettingsQoS* logging
11. Run *is_plain* method with the corresponding data representation
12. Address compilation issue for GLIBC version
13. Repository and test improvements:
 1. Run Github Ubuntu CI on PRs
 2. Only run PRs CI when a review is requested
 3. Improve filtering of DNS tests
 4. Make *sample_lost_be_dw_be_dr_fragments* test less flaky
 5. Build Fast DDS Python bindings in Fast DDS Docs Github CI job
 6. Add check for XML API to PR template
 7. Refactor Github CI sanitizer related jobs
 8. Increase sleep to miss the deadline in macOS flaky tests
 9. Fix nightly sanitizer CI workflow name
 10. Improve xml loading method in *fuzz_XMLProfiles*
 11. Build ShapesDemo on Ubuntu Github CI
 12. Correctly set branches for *reusable-sanitizers-ci*

This release includes the following **fixes**:

1. Address XMLProfiles fuzzer regressions
2. Fix flaky Log tests
3. Fix hidden overloaded virtual methods
4. Effectively assert *automatic/manual_by_participant* liveliness
5. Fix *on_sample_lost* notification on best-effort readers for fragmented samples
6. Monitor service properly managing instances
7. Fix CVE-2024-30258

8. Fix Discovery Server over TCP using LocatorSelectorEntry
9. Removed warning
10. Make `get_first_untaken_info()` coherent with `read()/take()`
11. Fix serialization of Data(w)/Data(r) in MonitorService
12. Bugfix in DDSBlackboxTestsDataReader test
13. Fix leak in `SecurityManager::participant_volatile_message_secure_writer_`
14. Fix some leaks in XML DynamicTypes Parser
15. Fix support for `@key` annotation in Dynamic types
16. Fix SecurityTest unit tests memory error
17. Fix Python Installation version in Github CI. Address failing system tests environment issues

Note: When upgrading to version 2.14.0 it is **advisable** to regenerate generated source from IDL files using [Fast DDS-Gen v3.3.0](#).

Version 2.14.0 (latest)

Important: Fast DDS v2.14 will be the last minor version of Fast DDS v2, the next major release will be Fast DDS v3.0.0, stay tuned!

This release includes the following **new features**:

1. Add *netmask filter* transport configuration
2. Add *interface allowlist and blocklist*
3. Expose *Security Authentication plugin* Handshake Properties
4. Extend *LARGE_DATA configuration options*
5. Allow *single listening port on TCP*
6. OpenOutputChannels / CloseOutputChannels that receive a LocatorSelectorEntry
7. Support GTest v1.14.0
8. Update Fast CDR submodule to v2.2.0
9. Update roadmap and release support

This release includes the following **improvements**:

1. Ignore custom PIDs defined in Fast DDS when they are received from other vendors
2. Add catch of out-of-range exception for thread settings port
3. Explicitly pass vendor ID to `readFromCdrMessage`
4. TCP transport improvements:
 1. TCPSendResources cleanup
 2. TCP `non_blocking_send` moved to *TCPTransportDescriptor*
5. Various repository improvements and housekeeping:

1. Migrate communication tests to the DDS API
 2. Migrate TCPReqRepHelloWorldReplier/Requester to the DDS API
 3. Update APIs in video performance tests to the DDS API
 4. Remove FASTDDS_TODO_BEFORE 2.14
 5. Remove use of deprecated FindPythonInterp
 6. Remove idl parser from .repos file
 7. Force usage of semicolon in FASTDDS_TODO_BEFORE macro
 8. Ubuntu example testing automation infrastructure
6. Multiple Github CI improvements:
1. Add nightly Ubuntu Github CI
 2. Improve CI version management
 3. Avoid running GitHub CI if PR has conflicts
 4. Migrate apt package installation to eProsima-CI action
 5. Include missing nightly for 3.0.x branch
 6. Adapt nightly jobs for all supported versions
 7. Pin CMake version and vm.mmap_rnd_bits in sanitizer workflows
 8. Select reusable workflow version depending on target branch in nightly jobs

This release includes the following **fixes**:

1. Fix warnings when compiling for Windows x86
2. TCP first message loss (see *TCPTransportDescriptor*)
3. Fix warnings on tests on Windows 32bits
4. Protect asio exception hotfix
5. Fix CVE-2024-28231
6. Github CI fixes:
 1. Fix CI documentation workflow label triggering
 2. Fix nightly jobs
 3. Fix input branch on reusable windows CI
 4. Use correct version of GTest on DS ASan tests

Note: When upgrading to version 2.14.0 it is **advisable** to regenerate generated source from IDL files using *Fast DDS-Gen v3.3.0*.

7.44.2 Version 2.13

Version 2.13.4

This release includes the following **features** in an ABI compatible manner:

1. Expose *Authentication Handshake Properties*

This release includes the following **improvements**:

1. Monitor service properly managing instances
2. Effectively assert AUTOMATIC / MANUAL_BY_PARTICIPANT liveliness
3. Add catch of out-of-range exception for thread settings port
4. TCP transport improvements:
 1. TCPSendResources cleanup
 2. TCP first message loss
 3. Set real TCP non_blocking_send limitation

This release includes the following **fixes**:

1. Fix hidden overloaded virtual methods
2. Fix Discovery Server over TCP using logical port
3. Protect asio exception fix
4. Fix flaky Log tests
5. Fix CVE-2024-28231
6. Add missing virtual destructor for StatisticsAncillary
7. Increase ack waiting time in reliable_on_unack_sample_removed
8. Fix versions in fastrtps.repos
9. GitHub CI fixes:
 1. Fix CI version management
 2. Add manual Ubuntu Github CI
 3. Avoid running GitHub CI if PR has conflicts
 4. Migrate apt package installation action to eProsima-CI
 5. Only run PRs CI when review requested
 6. Pin CMake version and vm.mmap_rnd_bits in sanitizer workflows
 7. Improve filtering of DNS tests

Note: When upgrading to version 2.13.4 it is **advisable** to regenerate generated source from IDL files using *Fast DDS-Gen v3.2.1*.

Version 2.13.3

This patch release includes the following **improvements**:

1. Enabling multiple interfaces through whitelist in TCP servers
2. Set LARGE_DATA Participants logic with the same listening ports
3. Check History QoS inconsistencies

This patch release includes the following **fixes**:

1. Prevent index overflow and correctly assert the end iterator in DataSharing
2. Fix the shared memory cleaning script
3. Fix TCP reconnection after open logical port failure
4. Fix data race on PDP
5. Fix doxygen docs warnings. Prepare for compiling with Doxygen 1.10.0
6. Add missing TypeLookup listeners
7. Restore Blackbox tests names
8. Add macOS Github CI
9. Set 2.11.x as EOL

Note: When upgrading to version 2.13.3 it is **advisable** to regenerate generated source from IDL files using [Fast DDS-Gen v3.2.1](#).

Version 2.13.2

This patch release includes the following **fixes** and **improvements**:

1. Improve environment variable substitution algorithm
2. Add large data to the Advanced Configuration example
3. Discard already processed samples on PDPListener
4. Ignore 0x8007 if coming from other vendor
5. TCP unique client announced local port
6. TCP non-blocking send
7. TCP Client&Server Participant Decision-Making
8. Add non-throwing getters for socket info
9. Add a keyed fragmented change to the reader data instance only when it is completed
10. Include a variety of terminate process signals handlers in the discovery server
11. Make DataWriters always send the key hash on keyed topics
12. Update Fast DDS types with Fast DDS Gen to include <cstdint> in v1 types
13. Add serialization for Log::Kind to ostream
14. Fix wrong log info messages on TCP
15. Return const reference in get_log_resources

16. Remove unnecessary warning
17. Avoid a maybe-uninitialized warning
18. Add static cast to an unused variable
19. Prepare for v3.0.0 branch out
20. Windows CI fixes

Note: When upgrading to version 2.13.2 it is **advisable** to regenerate generated source from IDL files using [Fast DDS-Gen v3.2.1](#).

Version 2.13.1

Note: This release upgrades the following Fast DDS dependencies:

- [Fast CDR v2.1.3](#)
 - [Fast DDS-Gen v3.2.1](#)
-

This release includes the following **improvements**:

1. Downgrade CMake minimum required version to 3.20.
2. Update PR template to include check for PR description, title and backports.
3. Update Fast CDR submodule to v2.1.3.

This patch release includes the following **fixes**:

1. Revert wrong fix for TCP deadlock on channel reuse.
2. Fix wrong history selection when removing PDP samples.
3. Fix data race when processing incoming PDP samples on different threads.
4. Fix DNS test filter in CMakeLists.
5. Fix deserialization of unions in generated code.

Note: When upgrading to version 2.13.1 it is **advisable** to regenerate generated source from IDL files using [Fast DDS-Gen v3.2.1](#).

Version 2.13.0

Note: This release upgrades the following Fast DDS dependencies:

- [Fast CDR v2.1.2](#)
 - [Fast DDS-Gen v3.2.0](#)
-

This release includes the following **features**:

1. Support *Monitor Service*.

2. Enable configuration of *thread settings* for all threads (both through the C++ API and XML configuration files).
3. Support *Autofill port* (automatic assignment of a port) for *TCP Transport* listening port.
4. Support *TCP for Discovery Server* CLI and environment variable.
5. Usage of `gtest_discover_tests`.
6. Define a *super client* by environment variable.
7. Support adding interfaces to the *interface whitelist* by the name.
8. Add new methods to configure *Builtin Transport*.
9. Support *DataRepresentationQos*.
10. Change serialize function default behavior to omit the data representation.
11. Upgrade Fast CDR submodule to v2.1.2.
12. Update roadmap & platforms support.

This release includes the following **improvements**:

1. Rerun failed tests with `ctest` option instead of `colcon`'s.
2. Add CCache to all CI jobs.

This release includes the following **fixes**:

1. **Fast DDS bugfixes**

1. Fix compilation of *XMLProfileParserTests* when building without security.
2. Improve *IgnoreNonExistentSegment* test for Windows.
3. Add missing thread includes.
4. Fix warning in Mac rewarding unnecessary lambda capture.
5. Use *SO_EXCLUSIVEADDRUSE* for Win32 unicast listening sockets.
6. Fix `gtest` discovery timeout.
7. Mark *on_participant_discovery* overload removal.
8. Fix uninitialized member in *BuiltinAttributes* class.
9. Fix set affinity directive for Android.
10. Fix Monitor Service types & test without security.
11. Fix TCP deadlock on channel reuse.
12. Fix DNS filter in *CMakeLists* file for tests.
13. Fix memory issues related to ciphering payload.
14. Fix a bad-free when receiving a malformed *DATA_FRAG* submessage.
15. Fix CVE-2023-50257.
16. Fix compilation of Fast DDS Python tests.
17. Fix data race on writer destruction while sending heartbeat.
18. Fix build with TLS, when *SECURITY=OFF* and *NO_TLS=OFF*.

2. CI fixes:

1. Fix `colcon` on github CI.

2. Better handling of trigger events in docs CI.

Note: When upgrading to version 2.13.0 it is **advisable** to regenerate generated source from IDL files using [Fast DDS-Gen v3.2.0](#).

7.44.3 Version 2.10

Version 2.10.3

This release includes the following **features** in an ABI compatible manner:

1. Support Autofill `port` (*automatically set a port*) for TCP Transport.
2. Define a *super client* by environment variable
3. Support *TCP Discovery server* CLI and environment variable
4. Define methods (*environment variable*, *rtps layer*, *xml*) to *configure transport scenarios*
5. Custom pools on DDS layer (*DataWriter* and *DataReader*)

This release includes the following **improvements**:

1. Log warning upon receiver resource creation failure
2. Simplify code in `CDRMessage`
3. Backport workflows from master
4. Rerun failed tests with `ctest` option instead of `colcon`'s
5. Use foonathan memory manager for reducing allocations in `SharedMemManager.hpp`
6. Add `CCache` to all CI jobs

This release includes the following **bugfixes**:

1. Fix `RemoteBuiltinEndpointHonoring` blackbox test
2. Fix bad-free when receiving malformed DATA submessage
3. Fix clang warnings
4. Use STL implementation of `Timed/RecursiveTimedMutex` when `MSVC >= 19.36`
5. Notify data-sharing listener at the end of a successful matching in intraprocess
6. Fix the clang build for clang 14
7. Fix `HelloWorld` Data-Sharing example idl
8. Fix the behaviour of `disable_positive_acks` period
9. Fix `DomainParticipant::register_remote_type` return when negotiating type
10. Fix Data Race when updating liveliness changed in WLP
11. Fix TCP sender resources creation
12. Fix flow controllers unit tests compilation when using Fast CDR from thirdparty
13. Add XML parser `bit_bound` bounds check
14. Add tests for reconnection with same GUID

15. Fix Github Windows CI
16. Fix PubSubAsReliable test
17. Use FASTRTPS_NO_LIB on unittest root folder
18. Fix missing mandatory attribute check in XML parser struct type
19. Fix mac address overflow on windows
20. Use SO_EXCLUSIVEADDRUSE for Win32 unicast listening sockets
21. Fix FileWatchTest for Github windows CI
22. Add missing thread include
23. Update TLS unit test certificates
24. Select correct .repos file on push events
25. Fix documentation CI branch
26. Fix TCP deadlock on channel reuse
27. Fix DNS filter in CMakeLists file for tests
28. Fix bad-free when receiving malformed DATA_FRAG submessage
29. Fix memory problem when ciphering payload
30. Fix build with TLS, but not security
31. Fix CVE-2023-50257
32. Fix data race on writer destruction while sending heartbeat
33. Fix comparison in `remove_from_pdp_reader_history`
34. Fix data race in PDPListener and SecurityManager
35. Update PR template to include check for PR description, title and backports
36. Fix `std::move` warning
37. Revert “TCP deadlock on channel reuse”
38. Fix max clash with Windows CI
39. Remove unnecessary TCP warning
40. Discard already processed samples on PDPListener
41. TCP unique client announced local port

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*.

Version 2.10.2

This release includes the following **improvements**:

1. Fix Data-Sharing delivery when data_count is zero
2. Improve performance of intraprocess plus data-sharing
3. Improve content filter expression parameters checks and verbosity
4. Improve validation on PID_PROPERTY_LIST deserialization
5. Participant ignore local endpoints
6. Pick smallest available participant ID for new participants
7. Improve endpoint QoS XML tags
8. Forward compatibility with boost interprocess 1.74+
9. Cap Thread Sanitizer memory usage to prevent runner shutdown
10. Allow participant XML profile with no <rtps> tag
11. Add unsupported note in API documentation to new ignore DomainParticipantListener callbacks
12. Add documentation version fallback

This release includes the following **bugfixes**:

1. Fixed long-standing reconnection issue on SHM transport
2. Fix null dereference when fuzzing
3. Fix segfault when creating two participant with same fixed id
4. Fix UBSan (Undefined Behavior Sanitizer) issues
5. Fix listener selection for on_requested_deadline_missed
6. Fix build on msvc 19.36.32528
7. Fix XML schema to set Transport descriptor kind as NOT mandatory
8. Fix missing includes
9. Fix overhead time unit
10. Fix request reply example spelling typo
11. Fix topic deletion after endpoint in examples
12. Fix Data-Sharing delivery when data_count is zero
13. Wait for log background thread initialization on the first queued entry
14. Fix alias resolve in DDSSQLFilter
15. Fix partition copy in QoS
16. Fix StatelessWriter locators filtering
17. Fix XMLParser null-dereference in parseLogConfig
18. Fix encapsulation format in WLP
19. Replace uint64_t by 8 in alignas specifier
20. Capture all Fast CDR exceptions
21. Security module: Honor allow_unauthenticated_participants flag

22. Explicitly register type object in ContentFilteredTopicExample
23. Avoid double definition of FASTDDS_ENFORCE_LOG_INFO
24. Fix API Fast DDS v2.10.0 API break calling correctly on_participant_discovery callbacks
25. Remove mutex from TimedEventImpl

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.10.1

This release includes the following **feature** in an ABI compatible way:

1. `ignore_participant()` implementation

This release includes the following **bugfixes**:

1. Fix repeated matched event notification.
2. Fix regression introduced by #3396.
3. Initial acknack backoff.
4. Fix regular expression in XML schema.
5. Correctly assign multicast port to multicast initial peers.

This release includes the following **CI improvements**:

1. Update runner and GCC version for Thread Sanitizer job.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.10.0

This minor release includes several new features, improvements and bugfixes.

Note: Mind that, even though this release is API compatible with previous v2.x versions, it is *NOT* ABI compatible with previous versions. This means that applications upgrading Fast DDS to v2.10.0 will require recompilation, though not source code modification.

Note: It is also advisable to regenerate the type support from the IDL files using [Fast DDS-Gen v2.4.0](#). Furthermore, if upgrading to v2.10.0, it is also recommended to upgrade Fast CDR to [v1.0.27](#).

This release includes the following **features**:

1. New *on_unacknowledged_sample_removed callback* in `DataWriterListener`.
2. *Secure Discovery Server*.
3. DomainParticipant ignore empty API.

4. RTPS `ReaderListener::on_incompatible_type` and `WriterListener::on_incompatible_type` empty API.

This release includes the following **improvements**:

1. **Fast DDS improvements**

1. Improve behavior when `STRICT_REALTIME` *CMake option* is not enabled.
2. Using functors for `for_matched_readers` parameter.
3. Improve auto GAPS in Data Sharing.
4. Use standard value for `PID_RELATED_SAMPLE_IDENTITY`.

2. **Contributions and repository quality**

1. Update Pull Request template.
2. Update foonathan_memory quality declaration.
3. Update XSD schema.
4. Make network headers private avoiding exposing non-public API.
5. Improve Doxygen documentation for `ResourceLimitsQosPolicy`.

3. **Examples**

1. New *Request-Reply example*.

4. **CI improvements**

1. New workflow to check documentation build.
2. ASAN workflow updated to use Ubuntu 22.04.

5. **Dependencies**

1. Upgrade internal type supports using latest Fast DDS-Gen release v2.4.0.
2. Upgrade Fast CDR submodule to v1.0.27.

6. **Fast DDS CLI**

1. Handle `SIGTERM` signal.

7. **Community supported platforms**

1. *QNX 7.1 build infrastructure*.

This release includes the following **fixes**:

1. **Security vulnerability**

1. Fix chain of trust issues with a single CA certificate.

2. **Bugfixes**

1. Fix RTPS StatelessWriter ACK check.
2. ASAN (Address Sanitizer) fixes.
3. UBSan (Undefined Behavior Sanitizer) fixes.
4. Export public API correctly in Windows.
5. Correctly handle builtin endpoints mask.
6. Fix backwards compatibility using SHM communication.

7. Protect against uncaught exception in SHM segment creation.
8. Fix build for GCC 5.
9. Validity check for first sequence number.
10. Fix crash when enabling `DisablePositiveACKsQoS`Policy with remote best-effort readers.

3. Synchronization fixes

1. Take mutex when removing local reader in WLP.
2. Fix data races in SecurityManager authentication process.

4. CI fixes

1. Fix test building when using `GTEST_INDIVIDUAL` *CMake option*.
2. Fix overflow in received samples in performance tests.

5. Example fixes

1. Avoid creating entities within callbacks in `DynamicHelloWorldExample`.
2. Remove Asio dependency from `DeadlineQoSExample`.

6. Repository fixes

1. Remove 2.7.x as active branch.

7. Community supported platforms

1. Include right header when building for iOS.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.44.4 Version 2.6

Version 2.6.8

This release includes the following **features**:

1. *Authentication Handshake Properties* documentation.
2. TCP Client and Server Participant Decision Making.

This release includes the following **improvements**:

1. Make DataWriters always send the key hash on keyed topics.
2. Include variety of terminate process signals handler in discovery server.
3. Effectively assert `AUTOMATIC/MANUAL_BY_PARTICIPANT` liveliness.
4. Pick smallest available participant ID for new participants.
5. Check History QoS inconsistencies.
6. Add check for XML API to PR template.
7. `LARGE_DATA` Participants logic with same listening ports.

TCP transport improvements:

1. TCP unique client announced local port.
2. Remove unnecessary TCP warning and Fix some tests.
3. TCP non-blocking send.
4. Enabling multiple interfaces through whitelist in TCP servers.
5. Set real TCP non-blocking-send limitation.

Github CI management:

1. Refactor Github CI sanitizer related jobs.
2. Avoid running GitHub CI if PR has conflicts.
3. Add manual Ubuntu Github CI.
4. Improve CI version management.
5. Build ShapesDemo on Ubuntu Github CI.
6. Only run PRs CI when review is requested.
7. Add macOS and Ubuntu Github CI.
8. Build Fast DDS Python bindings in Fast DDS Docs Github CI job.
9. Pin CMake version and `vm.mmap_rnd_bits` in sanitizer workflows.

This release includes the following **fixes**:

1. Fix and refactor Windows Github CI.
2. Fix max clash with Windows CI.
3. Fix the shared memory cleaning script.
4. Fix doxygen docs warnings. Prepare for compiling with Doxygen 1.10.0.
5. Prevent index overflow and correctly assert the end iterator in DataSharing.
6. Add a keyed fragmented change to the reader data instance only when its completed.
7. Add missing virtual destructor for `StatisticsAncillary`.
8. Fix wrong log info messages on TCP.
9. Migrate apt package installation action to eProsima-CI.
10. Fix CI documentation workflow label triggering.
11. Upgrade dependency version to last patch version in `.repos` file.
12. Fix CVE-2024-28231
13. Fix data race on PDP.
14. Discard already processed samples on PDPListener.
15. Fix flaky Log tests.
16. Add missing `TypeLookup` listeners.
17. Fix hidden overloaded virtual methods.
18. Fix TCP reconnection after open logical port failure.
19. Fix CVE-2024-30258 / CVE-2024-30259
20. Make `DataReader::get_first_untaken_info()` coherent with `read()/take()`.

21. Removed warning in `ParameterList`.
22. TCP avoid first message loss.

Version 2.6.7

This release includes the following **features**:

1. Support Autofill port (*automatically set a port*) for TCP Transport.
2. Define a *super client* by environment variable
3. Support *TCP Discovery server* CLI and environment variable
4. Define methods (*environment variable, rtps layer, xml*) to *configure transport scenarios*
5. *Secure discovery server*

This release includes the following **improvements**:

1. Log warning message upon receiver resource creation failure.
2. Add tests for reconnection with same GUID
3. Usefoonathan memory manager for reducing allocations in `SharedMemManager.hpp`
4. Simplify code in `CDRMessage`.
5. Rerun failed tests with `ctest` option instead of `colcon's`.
6. Several improvements on CI jobs.
7. Upgrade CMake minimum requirement to 3.16.3
8. Update PR checklist template. Backports and Description

This release includes the following **bugfixes**:

1. Fix `DomainParticipant::register_remote_type` return when negotiating type.
2. Fix `RemoteBuiltinEndpointHonoring` blackbox test.
3. Allow participant profiles with no `rtps` tag.
4. Fix bad-free when receiving malformed DATA submessage.
5. Fix clang warnings
6. Use STL implementation of `Timed/RecursiveTimedMutex` when `MSVC >= 19.36`.
7. Fix encapsulation format in WLP.
8. Fix the clang build for clang 14.
9. Notify data-sharing listener at the end of a successful matching in intraprocess.
10. Updatable `disable_positive_acks` period.
11. Fix Data Race when updating liveliness changed in WLP.
12. Fix TCP sender resources creation.
13. Fix flow controllers unit tests compilation when using Fast CDR from thirdparty.
14. Add XML parser `bit_bound` bounds check.
15. Use `FASTRTPS_NO_LIB` on unittest root folder.
16. Use `SO_EXCLUSIVEADDRUSE` for Win32 unicast listening sockets.

17. Fix mac address overflow on windows.
18. Fix PubSubAsReliable test.
19. Fix FileWatchTest.
20. Add missing thread include.
21. Fix missing mandatory attribute check in XML parser struct type.
22. Better handling of trigger events in docs CI.
23. Fix memory problem when ciphering payload
24. Select correct .repos file on push events
25. Update TLS unit test certificates
26. Fix bad-free when receiving malformed DATA_FRAG submessage
27. Fix data race on writer destruction while sending heartbeat
28. Fix DiscoveryServer list access deadlock
29. Fix c++11 support for fast discovery server tool
30. Fix CVE-2023-50257
31. Fix std::move warning
32. Fix Github Windows CI

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*.

Version 2.6.6

This release includes the following **improvements**:

1. Improve validation on PID_PROPERTY_LIST deserialization.
2. Improved CPU usage of timed events thread.
3. Improved performance on intraprocess + data-sharing.
4. Explicitly register type object in ContentFilteredTopicExample.
5. Improve installer generation with documentation version fallback.
6. Improve content filter expression parameters checks and verbosity.

This release includes the following **bugfixes**:

1. Fixed long-standing reconnection issues on SHM transport.
2. Correctly resolve alias in DDSQLFilter.
3. Fixed partition copy in QoS.
4. Added length checks to prevent nullptr memory copy calls.
5. Fixed XMLParser null-dereference when parsing log configuration.
6. Fixed SHM in 32-bit architectures.
7. Added missing include.

8. Avoid double definition of FASTDDS_ENFORCE_LOG_INFO.
9. Fixed statistics data_count with data-sharing.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.6.5

This release includes the following **improvements**:

1. Improve behavior when STRICT_REALTIME *CMake option* is not enabled.
2. Using functors for `for_matched_readers` parameter.
3. Improve auto GAPs in Data Sharing.

This release includes the following **bugfixes**:

1. Fix RTPS StatelessWriter ACK check.
2. Fix `total_read_` to be consistent with Reader's History after `DataReader::get_first_untaken_info()`.
3. Add deprecation notice to `ThroughputControllerDescriptor`.
4. UBSan (Undefined Behavior Sanitizer) fixes.
5. Several dependencies fixes upgrading to Ubuntu 22.04.
6. Fix chain of trust issues with a single CA certificate.
7. Correctly handle builtin endpoints mask.
8. Take mutex when removing local reader in WLP.
9. Handle SIGTERM signal in Fast DDS CLI.
10. Fix data races in SecurityManager authentication process.
11. Avoid creating entities within callbacks in `DynamicHelloWorldExample`.
12. Remove Asio dependency from `DeadlineQoSExample`.
13. Validity check for first sequence number.
14. Include right header when building for iOS.
15. Fix build on MSVC 19.
16. Correctly assign multicast port to multicast initial peers.
17. Select correct listener for `on_requested_deadline_missed()`.
18. Forward compatibility with boost inter-process 1.74+.
19. Fix missing includes when building with GCC 13.
20. Honor `allow_unauthenticated_participants` flag.
21. Capture all Fast CDR exceptions.
22. Fix example to delete Topic after deleting the corresponding Endpoint.
23. Protect against uncaught exception in SHM segment creation.
24. Initial acknack backoff.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.6.4

This release includes the following **bugfixes**:

1. Fix communication with asymmetric *Ignore Participant flags*.
2. Fix deadlock in Writer Liveliness Protocol when using intraprocess.
3. Fix notification lost.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.6.3

This release includes the following **improvements**:

1. Upgrade CMake minimum requirement to 3.13.
2. Improve Guid_t operator < performance.

This release includes the following **bugfixes**:

1. Add python3 dependency to package.xml.
2. Fix complex member printing for DynamicDataHelper.
3. Fix selection of output locators.
4. Fix null references on XML parser.
5. Fix data races when creating DataWriters.
6. Send GAPs correctly when using separate sending.
7. Install Statistics IDL file.
8. Fixes for building in older compilers.
9. Fix deadlock when removing DomainParticipant when using SECURITY.
10. Ensure `shared_mutex` implementation is consistent throughout supported platforms.
11. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.6.2

This release includes the following **improvements**:

1. Support for GCC 12.
2. Overload `DataReader::get_unread_count()`.
3. Improve read/take performance when using topic with a great number of keys.
4. Improve rediscovery on lossy environments.

This release includes the following **bugfixes**:

1. Fixed several deadlocks and data races.
2. Fixed validation on `ParameterPropertyList_t`.
3. Fixed wrong usage of `std::remove_if`.
4. Fixed acknowledgement in `DataSharing`.
5. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.6.1

This release includes the following **improvements**:

1. Support for *writer side content filtering*
2. Support hexadecimal values on SQL filter
3. Support for `DataWriter::get_key_value()`
4. Support for `DataReader::lookup_instance()`
5. Support for `SampleLostStatus` on `DataReader`
6. Improved doxygen documentation

Some **bugfixes** are also included:

1. Fixed several lock order inversion issues
2. Fixed data race when closing UDP channels
3. Fixed empty partition validation checks
4. Fixed corner case with reliable writers and samples with a huge number of fragments
5. Other minor fixes and improvements

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.6.0

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed. Some API is also being deprecated.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This minor release includes the following **features**:

1. *Allow modifying remote server locators at runtime*
2. *Add statistics physical information to DATA[p] using properties*
3. *Content filter discovery information RTPS API*
4. Endpoint discovery RTPS API
5. `on_sample_lost` RTPS API
6. Transport layer API extension
7. *XML support for Fast DDS CLI*
8. *New exchange format to reduce bandwidth in Static Discovery*

It also includes the following **improvements**:

1. Support lowercase keywords on SQL filter
2. Separate initialization and enabling of BuiltinProtocols
3. Add `disable_positive_acks` to Static Discovery XML
4. Several updates in the DDS-PIM API
5. Support for octet vectors on XML parser
6. Update README and roadmap
7. Update Fast-CDR submodule to v1.0.24
8. Add new CMake option `APPEND_PROJECT_NAME_TO_INCLUDEDIR`

Some **bugfixes** are also included:

1. Fix `MatchedStatus last_*_handle`
2. Fix recommended statistics `DataReaderQos` to enable backwards compatibility
3. Fixes for supporting Python bindings in Windows platforms
4. Fix publishing physical data on statistics topic
5. Other minor fixes and improvements

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using `fastdds-gen`. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45 Previous end-of-life versions

7.45.1 Version 2.12 (EOL)

Version 2.12.2 (EOL)

This release includes the following **features**:

1. Methods to configure *transport scenarios*
2. Support `Autofill port` (*automatically set the port*) for TCP Transport
3. Support *TCP for Discovery Server* CLI and environment variable
4. Define a *super client by environment variable*
5. Change `serialize` function default behaviour to omit the data representation
6. `LARGE_DATA` Participants logic with same listening ports
7. TCP Client&Server Participant Decision-Making logic
8. Expose *Authentication Handshake Properties*
9. Enabling multiple interfaces through whitelist in TCP servers
10. Add macOS and Ubuntu Github CI

This release includes the following **improvements**:

1. Improve environment variable substitution algorithm
2. Upgrade dependency version to last patch version in `.repos` file
3. Rerun failed tests with `ctest` option instead of `colcon's`
4. Remove unnecessary TCP warning
5. Update PR template to include check for PR description, title and backports
6. Improvements in GitHub CI

This release includes the following **fixes**:

1. Fix TCP reconnection after open logical port failure
2. TCP unique client announced local port
3. TCP non-blocking send
4. Fix wrong log info messages on TCP
5. Improve `IgnoreNonExistentSegment` test
6. Use `SO_EXCLUSIVEADDRUSE` for Win32 unicast listening sockets
7. Fix DNS filter in `CMakeLists` file for tests
8. Fix bad-free when receiving malformed `DATA_FRAG` submessage
9. Fix memory problem related to ciphering payload
10. Fix CVE-2023-50257
11. Fix build with TLS, but not security
12. Fix comparison in `remove_from_pdp_reader_history`

13. Fix data race in PDPListener and SecurityManager
14. Discard already processed samples on PDPListener
15. Fix .repos versions
16. Fix the shared memory cleaning script
17. Fix data race on writer destruction while sending heartbeat
18. Return const reference to the shared pointer instead of a copy in `get_log_resources`
19. Ignore `0x8007` if coming from other vendor
20. Fix Doxygen docs warnings and prepare for compiling with Doxygen 1.10.0
21. Include variety of terminate process signals handler in discovery server
22. Add missing TypeLookup listeners
23. Add a keyed fragmented change to the reader data instance only when its completed
24. Fix data race on PDP
25. Check History QoS inconsistencies
26. Make DataWriters always send the key hash on keyed topics
27. Prevent index overflow and correctly assert the end iterator in DataSharing
28. Fix uninitialized member in RTPSParticipantAttributes
29. Remove unnecessary `std::move` in `FileWatch.hpp` causing warning
30. Add missing thread include
31. Add missing virtual destructor for `StatisticsAncillary`
32. Protect asio exception
33. TCPSendResources cleanup
34. Downgrade CMake version to 3.20

Version 2.12.1 (EOL)

This release includes the following **improvements**:

1. Support for linking with Fast CDR v1.
2. The period for the timer within the *DisablePositiveACKsQosPolicy* is now updatable.
3. Log error message upon receiver resource creation failure.
4. CI and repository improvements.
5. Simplify code in `CDRMessage`.

This release includes the following **fixes**:

1. **Fast DDS bugfixes**
 1. Fix transient local durability for reliable readers using intra-process and data-sharing.
 2. Use STL implementation of Timed/RecursiveTimedMutex when *MSVC* ≥ 19.36 .
 3. Fix updatability of immutable *DataWriterQos*.
 4. Fix the clang build for clang 14.

5. Fix remote locators filtering when whitelist provided.
 6. Fix Data Race when updating liveliness changed in WLP.
 7. Add XML parser bit_bound bounds check.
 8. Fix missing mandatory attribute check in XML parser struct type.
 9. SHM transport: ignore non-existing segment on pop.
 10. Fix: mac address overflow on Windows.
2. CI fixes:
1. Fix flow controllers unit tests compilation when using Fast CDR from thirdparty.
 2. PubSubAsReliable test fix.
 3. FileWatchTest fix for github windows CI.

Note: When upgrading to version 2.12.1 it is **advisable** to regenerate generated source from IDL files using [Fast DDS-Gen v3.1.0](#).

Version 2.12.0 (EOL)

Note: This release upgrades the following Fast DDS dependencies:

- [Fast CDR v2.0.0](#)
- [Fast DDS-Gen v3.0.1](#)

Please, read also the release notes of [Fast DDS-Gen v3.0.0](#) to be aware of every possible break in the application code.

As Fast DDS dependencies have been upgraded to new major releases, depending on the types defined in the IDL files, it might be required to modify the user application source code besides recompiling it (more information can be found in the corresponding release notes).

Note: There is a minor API break with previous v2.x versions: `MEMBER_INVALID` identifier was declared using `#define`. In order to prevent polluting the user workspace, it has been transformed into a `constexpr` within `eprosima::fastrtps::types` namespace.

This release includes the following **features**:

1. *New participant property* to configure SHM transport metatraffic behavior.
2. Exposed custom payload pool on DDS *DataWriter* and *DataReader* declaration.
 1. Feature example.
3. *Processing environment variables in XML text*.
4. Dependencies
 1. Upgrade internal type supports using latest Fast DDS-Gen release v3.0.0. This release introduces the following features:
 1. *Support for @optional builtin annotation <optional_members>*.
 2. *Support for @extensibility builtin annotation <extensibility>*.

2. Upgrade Fast CDR submodule to v2.0.0 introducing XCDR encoding version 2.

This release includes the following **improvements**:

1. `fixed_string` comparison operators.
2. Remove mutex from *TimedEventImpl* (#3745, #3760)
3. Performance improvements on intraprocess and datasharing.
4. Improve Shared Memory resilience to crashing participants.
5. Improve scripts shebang portability.
6. Use `foonathan_memory` to reduce allocations in `SharedMemManager`.

This release includes the following **fixes**:

1. **Fast DDS bugfixes**

1. Fixed XMLParser null-dereference when parsing log configuration.
2. Allow participant XML profiles with no `<rtps>` tag.
3. Fix encapsulation format in Writer Liveliness Protocol.
4. Fix `DomainParticipant::register_remote_type` return when negotiating type.
5. Fix strict real-time feature when using Flow Controller feature.
6. Fix `ParameterPropertyList` increment operators.
7. Fix bad-free when receiving malformed DATA submessage.
8. Fix asymmetric whitelist matching.
9. Fix heap-use-after-free on `XMLElementParser`.
10. Fix History remove change return statement.

2. **CI fixes**

1. Fix `RemoteBuiltinEndpointHonoring` blackbox test.
2. Improve repository workflows.
3. Use `FASTRTPS_NO_LIB` on unittest root folder.
4. Fix Windows workflow.

3. **Tools**

1. Remove C++11 check in `fastdds-discovery-server` CLI tool.

4. **Examples**

1. Fix `HelloWorldDataSharing` data type.

5. **Documentation**

1. Doxygen typos.

6. **Repository**

1. Remove 2.9.x as active branch.

7. **Non Tier 1 support**

1. Fixed SHM in 32-bit architectures.
2. Fix warning on Win32 architecture.

Note: Upgrading to version 2.12.0 **requires** to regenerate generated source from IDL files using [Fast DDS-Gen v3.0.1](#).

7.45.2 Version 2.11 (EOL)

Version 2.11.3 (EOL)

This release includes the following **features** in an ABI compatible manner:

1. Support Autofill port (*automatically set a port*) for TCP Transport.
2. Define a *super client* by environment variable
3. Support *TCP Discovery server* CLI and environment variable
4. Define methods (*environment variable, rtps layer, xml*) to *configure transport scenarios*
5. Custom pools on DDS layer (*DataWriter* and *DataReader*)

This release includes the following **improvements**:

1. Allow participant profiles with no rtps tag
2. Add Log warning message upon receiver resource creation failure, instead of an error
3. Updatable disable_positive_acks period
4. Backport workflows from master
5. Update GitUtils.cmake
6. Use foonathan memory manager for reducing allocations in SharedMemManager.hpp
7. Rerun failed tests with ctest option instead of colcon's
8. Add CCache to all CI jobs
9. Simplify code in CDRMessage
10. TCP unique client announced local port
11. Make DataWriters always send the key hash on keyed topics
12. Include terminate process signals handler in discovery server

This release includes the following **fixes**:

1. Fix encapsulation format in WLP used for the ParticipantMessageData
2. Fix DomainParticipant::register_remote_type return when negotiating type
3. Fix RemoteBuiltinEndpointHonoring blackbox test
4. Fix .repos branches
5. Fix bad-free when receiving malformed DATA submessages
6. Fix clang warnings
7. Use STL implementation of Timed/RecursiveTimedMutex when MSVC >= 19.36
8. Fix the clang build for clang 14
9. Fix HelloWorld DataSharing example idl
10. Use FASTRTPS_NO_LIB on unittest root folder

11. Fix Data Race when updating liveliness changed in WLP
12. Fix TCP sender resources creation
13. Fix flow controllers unit tests compilation when using Fast CDR from thirdparty
14. Add XML parser `bit_bound` bounds check
15. Fix branch selection on Github CI
16. Better handling of trigger events in docs CI
17. Use `SO_EXCLUSIVEADDRUSE` for Win32 unicast listening sockets
18. Fix `PubSubAsReliable` test
19. Fix `FileWatchTest` for Github windows CI
20. Fix mac address overflow on windows
21. Fix missing mandatory attribute check in XML parser struct type
22. Update TLS unit test certificates
23. Add missing thread include
24. Add tests for reconnection with same GUID
25. Notify data-sharing listener at the end of a successful matching in intraprocess
26. Fix TCP deadlock on channel reuse
27. TCP non-blocking send
28. Fix DNS filter in `CMakeLists` file for tests
29. Fix bad-free when receiving malformed `DATA_FRAG` submessage
30. Fix memory problem when ciphering payload
31. Fix CVE-2023-50257
32. Fix build with TLS, but not security
33. Fix `std::move` warning
34. Update PR template to include check for PR description, title and backports
35. Fix data race on writer destruction while sending heartbeat
36. Fix comparison in `remove_from_pdp_reader_history`
37. Fix data race in `PDPListener` and `SecurityManager`
38. Fix an uninitialized value when building with GCC 13.2.0
39. Fix max clash with Windows CI
40. Discard already processed samples on `PDPListener`
41. Remove unnecessary TCP warning
42. Fix wrong log info messages on TCP
43. Revert “TCP deadlock on channel reuse”
44. Return const reference in `get_log_resources`
45. Add a keyed fragmented change to the reader data instance only when it is completed
46. Fix and refactor Windows Github CI

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.11.2 (EOL)

This release includes the following **improvements**:

1. Improve Shared Memory resilience to crashing participants
2. User configuration for *Shared Memory metatraffic*
3. Performance improvements on intraprocess and data-sharing

This release includes the following **fixes**:

1. Remove Mutex from TimedEventImpl
2. Replace uint64_t by 8 in alignas specifier
3. Fix XMLParser null-dereference in parseLogConfig

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.11.1 (EOL)

This release includes the following **improvements**:

1. Correct CONTRIBUTING.md typo
2. Improve validation on PID_PROPERTY_LIST deserialization
3. Apply eProsima brand style to Fast DDS repository
4. Fix spelling mistake: SUBTRACTION to SUBTRATION

This release includes the following **fixes**:

1. Fixed long-standing reconnection issue on SHM transport
2. Added missing include
3. Fixed Boost handle usage regression
4. Fix StatelessWriter locators filtering
5. Avoid double definition of FASTDDS_ENFORCE_LOG_INFO
6. Explicitly register type object in ContentFilteredTopicExample
7. Properly handle zero-sized payloads on dynamic memory payload pools

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.11.0 (EOL)

This release includes the following **features**:

1. *Ignore every local endpoint within the DomainParticipant preventing local matching.*
2. Extend DynamicDataHelper API providing a `print` overload with `std::ostream` as parameter.
3. TypeLookup Service settings.
4. Static Discovery XSD Schema.

This release includes the following **improvements**:

1. Fast DDS improvements

1. Assign minimum available participant ID to new participants.
2. Export symbols correctly on ContentFilteredTopic.
3. Improve content filter expression parameters check and verbosity.
4. Check TCP headers endianness.
5. Security module: distinguished names (DN) comparison.

2. Fast DDS deprecation

1. DDS:Crypto:AES-GCM-GMAC configuration using Property Policy QoS (security vulnerability).

3. CI improvements

1. Include BitmapRange unit tests.
2. Support for running some tests in parallel.
3. Windows workflow.

4. Build system

1. Improve CMake target loading. Removal of FASTDDS_STATIC CMake option.
2. Avoid auto-linkage using CMake.

5. Dependencies

1. Upgrade internal type supports using latest Fast DDS-Gen release v2.5.1.
2. Upgrade Fast CDR submodule to v1.1.0.

6. Examples

1. Admit XML configuration files in AdvanceConfigurationExample.
2. New Discovery Server example.

This release includes the following **fixes**:

1. Fast DDS bugfixes

1. Fix crash when creating two participants with the same fixed participant ID.
2. Fix crash when calling `on_requested_deadline_missed()` callback.
3. Fix crashes caused by not capturing every Fast CDR exception.
4. Correctly resolve aliases in DDSSQLFilter.
5. Wait for log background thread initialization on the first queued entry.
6. Fix data race when accessing `WRITE_PARAM_DEFAULT` static variable.

7. Fix partition copy in QoS.
8. Fix Data-Sharing delivery when data_count is zero.
9. Fix API Fast DDS v2.10.0 API break calling correctly. `on_participant_discovery()` callbacks.
10. Security module: Honor *Allow Unauthenticated Participants* flag.
11. Fix concurrent access to `load_profiles()`.
12. Fix UBSan (Undefined Behavior Sanitizer) issues.
13. Improve Doxygen documentation about DomainParticipantListener discovery callbacks.

2. XSD fixes

1. Set TransportDescriptor kind parameter as optional.
2. Correctly assign QoS to the proper endpoint.
3. Add missing tags.

3. CI fixes

1. Fix null dereference in fuzzer code.
2. Limit Thread Sanitizer memory usage to prevent runner shutdown.
3. Use correct time unit in latency tests.
4. Run communication tests.

4. Examples

1. Correct DDS entity deletion order.

5. Installer generation

1. Add documentation fallback when the documentation tag is not found.

6. Repository

1. Remove 2.1.x as active branch.
2. Remove 2.8.x as active branch.

7. Non Tier 1 support

1. Fix build on MSVC 19.36.
2. Forward compatibility with Boost inter-process 1.74+.
3. Include missing header files required for compiling with GCC 13.
4. QNX build fixes.
5. Fix build issues in RPM systems.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.3 Version 2.9 (EOL)

Version 2.9.2 (EOL)

This release includes the following ABI compatible **improvements**:

1. Library improvements

1. Improve behavior when `STRICT_REALTIME` *CMake option* is not enabled.
2. Using functors for `for_matched_readers` parameter.
3. Improve auto GAPS in Data Sharing.
4. Improve content filter expression parameters check and verbosity.
5. Improve validation on `PID_PROPERTY_LIST` deserialization.

2. Fast DDS CLI

1. Handle `SIGTERM` signal.

This release includes the following **bugfixes**:

1. Security vulnerability

1. Fix chain of trust issues with a single CA certificate.

2. Library bugfixes

1. Fix RTPS StatelessWriter ACK check.
2. UBSan (Undefined Behavior Sanitizer) fixes.
3. Fix backwards compatibility using SHM communication.
4. Correctly handle builtin endpoint mask.
5. Fix crash when enabling `DisablePositiveACKsQosPolicy` with remote best-effort readers.
6. Validity check for first sequence number.
7. ASAN (Address Sanitizer) fixes.
8. Correctly assign multicast port to multicast initial peers.
9. Protect against uncaught exception in SHM segment creation.
10. Initial acknack backoff.
11. Fix crash when calling `on_requested_deadline_missed()` callback.
12. Security module: Honor `allow_unauthenticated_participants` flag.
13. Fix crashes caused by not capturing every Fast CDR exception.
14. Correctly resolve aliases in `DDSSQLFilter`.
15. Wait for log background thread initialization on the first queued entry.
16. Fix partition copy in QoS.
17. Fix Data-Sharing delivery when `data_count` is zero.
18. Fix StatelessWriter locators filtering.
19. Avoid double definition of `FASTDDS_ENFORCE_LOG_INFO`.
20. Fixed long-standing reconnection issue on SHM transport.

3. CI fixes

1. Fix test building when using GTEST_INDIVIDUAL *CMake* option.
2. Use correct time unit in latency tests.

4. Synchronization fixes

1. Take mutex when removing local reader in WLP.
2. Fix data races in SecurityManager authentication process.

5. Example fixes

1. Avoid creating entities within callbacks in DynamicHelloWorldExample.
2. Remove Asio dependency from DeadlineQosExample.
3. Correct DDS entity deletion order.
4. Explicitly register TypeObject in ContentFilteredTopicExample.

6. Installer generation

1. Add documentation fallback when the documentation tag is not found.

7. Non Tier 1 Support

1. Fix build for GCC5.
2. Fix build on MSVC 19.36.
3. Include right header when building for iOS.
4. Forward compatibility with Boost inter-process 1.74+.
5. Include missing header files required for compiling with GCC13.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.9.1 (EOL)

This release includes the following **bugfixes**:

1. Synchronization fixes:

1. Fix deadlock in Writer Liveliness Protocol (WLP) using intraprocess.
2. Fix data race in DomainParticipant::set_listener().
3. Fix deadlock on TLS closure.

2. Other fixes:

1. Fix notification lost.
2. Fix total_read_ to be consistent with Reader's History after DataReader::get_first_untaken_info().
3. Use shared pointers for internal singletons.
4. Support CCache on Windows.
5. Avoid null dereference on fuzzer.

6. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.9.0 (EOL)

This minor release includes several new features (new log macros to avoid conflicts with external libraries), behavior change (default memory management policy is no longer `PREALLOCATED_MEMORY_MODE`, enable by default Fast DDS Statistics module but only taking statistics metrics if the corresponding Statistics DataWriter is enabled), performance improvements, CI improvements (including address sanitizer jobs), and several bug fixes.

Note: Mind that, even though this release is API compatible with previous v2.x versions, it is *NOT* ABI compatible with previous versions. This means that applications upgrading Fast DDS to v2.9.0 will require recompilation, though not source code modification.

Note: It is also advisable to regenerate the type support from the IDL files using [Fast DDS-Gen v2.3.0](#). Furthermore, if upgrading to v2.9.0, it is also recommended to upgrade Fast CDR to [v1.0.26](#).

This release includes the following **features**:

1. New log macros `EPROSIMA_LOG_INFO`, `EPROSIMA_LOG_WARNING` and `EPROSIMA_LOG_ERROR`.
2. Add `ENABLE_OLD_LOG_MACROS` CMake option to support *disabling the compilation of previous log macros*.

This release includes the following **behavior changes**:

1. Default *memory management policy* set to `PREALLOCATED_WITH_REALLOC_MEMORY_MODE`.
2. Statistics metrics are only calculated/accumulated when their corresponding DataWriter is enabled in *Fast DDS Statistics Module*.
3. Enable `FASTDDS_STATISTICS` CMake option by default.

This release includes the following **improvements**:

1. **CI improvements:**
 1. Add address sanitizer job for Fast DDS library.
 2. Add address sanitizer job for Discovery Server test suite.
2. Upgrade Fast CDR submodule.

This release includes the following **bugfixes**:

1. **Synchronization fixes:**
 1. Fix deadlock when removing remote DomainParticipants by expired liveliness when using Security.
 2. Protect `DomainParticipant::set_listener()` avoiding null reference.
 3. Fix data race on `WriterProxy::stop` while `TimedEvent` is being triggered.
 4. Protect creation/destruction of Boost's `named_mutex`.
2. **CI fixes:**
 1. Fix Statistics Module test suite.

2. Fix recurring data races in test suite.
3. Fix thread sanitizer job keeping Ubuntu 20.04.
3. **Other:**
 1. Fix Topic creation when registering a type name different from the internal TypeSupport name.
 2. Fix communication with asymmetric *Ignore Participant flags*.
 3. Several dependencies fixes upgrading to Ubuntu 22.04.
 4. Disable error logged when DomainParticipant profile is not found.
 5. Fix CMake for Fast DDS use as submodule.
 6. Upgrade internal type supports generated with Fast DDS-Gen v2.3.0.
 7. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.4 Version 2.8 (EOL)

Version 2.8.2 (EOL)

This release includes the following **improvements**:

1. Add ASAN CI tests for Fast DDS and Discovery Server
2. Mirror master onto both 2.8.x & 2.9.x
3. Doxygen documentation: add deprecation notice to *ThroughputControllerDescriptor*
4. Several fixes to remove warnings in Ubuntu Jammy (22.04)
5. Improve behavior when *HAVE_STRICT_REALTIME* is not set
6. Using functors in *StatefulWriter.cpp* for *_matched_readers*
7. Fix build on old compilers
8. Avoid creation of DynamicTypes on example
9. Implement a validity check for *firstSN*

This release includes the following **bugfixes**:

1. Fix bug in Topic creation with different Type Name
2. Fix tests failing with subprocess aborted error
3. Fix communication with asymmetric *ignoreParticipantFlags*
4. Added *ignore_participant_flags()* to *Blackbox_FastRTPS PubSubReader*.
5. Fix Deadlock in *remove_participant* (ResourceEvent thread) when compiled WITH_SECURITY
6. Fix failed tests when compiling with statistics enabled
7. Fix Windows *StatistisQosTests.cpp* linkage and Failed test
8. Fixing deadlock in WLP

9. Fix notification lost
10. Fix *StatelessWriter* ACK check
11. Fix *total_unread_* consistent with reader's history upon *get_first_untaken_info()*
12. Fix chain of trust issues
13. Fixed *StatisticsSubmessageData* unaligned access
14. Fix build error when *GTEST_INDIVIDUAL* is OFF
15. Correctly handle builtin endpoints mask
16. Added missing mutex to *WLP::remove_local_reader*
17. Handle SIGTERM in fast discovery server
18. Improve auto gaps in data sharing
19. Replaced *SecurityManager* temporary *ProxyDatas* with *ProxyPools*
20. Fix crash when *disable_positive_acks* is enable and the remote reader is best-effort
21. Protect from uncaught exception during SHM Segment creation
22. Fix asio dependency
23. Include the right header when building for iOS

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.8.1 (EOL)

This release includes the following **bugfixes**:

1. Statistics module fixes:

1. Increase statistics *DataWriter* history.
2. Fix Statistics module CI.
3. Install Statistics IDL file.
4. Fix for building in old compilers.
5. Fix core dumped in *delete_contained_entities()*.

1. Address sanitizer fixes:

1. Add ASAN CI job and **SANITIZE** *CMake option*.
2. Fixes reported by address sanitizer.

1. Synchronization fixes:

1. Fix data races when creating *DataWriters*.
2. Ensure *shared_mutex* implementation is consistent throughout supported platforms.

1. Other fixes:

1. Include missing *ReadCondition* header.
2. Fix selection of output locators.

3. Fix null-dereference on `parseXMLEnumDynamicType`.
4. Include 2.8.x branch release support.
5. Send GAPs correctly when using separate sending.
6. Fixes for building in old compilers.
7. Fix `DataReader::read_next_instance()` and `DataReader::take_next_instance()` implementation.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.8.0 (EOL)

This minor release includes several new features, performance improvements (especially in the case of topics with many instances), CI improvements (including the ability to run the CI in Android emulators or devices), and several bug fixes.

Note: Mind that, even though this release is API compatible with previous v2.x versions, it is *NOT* ABI compatible with previous versions. This means that applications upgrading Fast DDS to v2.8.0 will require recompilation, though not source code modification.

Note: It is also advisable to regenerate the type support from the IDL files using [Fast DDS-Gen v2.2.0](#). Furthermore, if upgrading to v2.8.0, it is also recommended to upgrade Fast CDR to [v1.0.25](#).

This release includes the following **features**:

1. Full *Ownership* and *OwnershipStrength* QoS support
2. *External locators*
3. UDPv6 support for *fast-discovery-server tool* and *ROS_DISCOVERY_SERVER*
4. *XML configuration support for statistics DataWriters QoS*
5. *SNI support*
6. *Propagate PropertyQoS properties when explicitly set*
7. Add API to create RTPSWriter with a custom pool
8. Add `std::string::compare` API to `fixed_string`
9. Get WAN address API in TCPv4 transport descriptors
10. Adding `DomainParticipantFactory::get_shared_instance()` API

This release includes the following **improvements**:

1. **Performance improvements:**
 1. Skip `writer_removed` processing for unaccounted instances
 2. Improve `GUID_t` operator< performance
2. **CI improvements:**

1. Add optional parameters to thread-sanitizer job
2. Enable Android testing on device
3. **Examples:**
 1. Update BasicConfigurationExample to allow set up TTL
 2. Add Guid info to BasicConfiguration Example cout
4. **Internal implementation improvements:**
 1. Add script to generate idl files
 2. Group set_qos_from_attributes free functions into a separate file
 3. Update script for generating idl files
 4. Set last_heartbeat_count_ private member of WriterProxy as atomic
5. Android Improvements
6. Upgrade Fast CDR submodule

This release includes the following **bugfixes**:

1. **Synchronization fixes:**
 1. Fix datarace using writer's locator selectors
 2. Add lock guard at changing SHM port listener status members
 3. Add atomic variable to prevent datarace in FlowController
 4. Disable RTPSParticipantImpl after removing it from RTPSDomain participants list
 5. Fixing datarace on listener callbacks
 6. Protect access to reader listeners
 7. Use thread-safe localtime function in unix distributions
 8. Fixed usage of uninitialised ifreq
 9. Adding protection to id_counter access
2. **Repository fixes:**
 1. Fix spelling mistake
 2. Add python3 dependency to package.xml
3. **Other:**
 1. Fix null dereference on parseXMLBitsetDynamicType
 2. Change internal include path of nlohmann/json header file
 3. Instance allocation consistency
 4. Fix complex member printing for DynamicDataHelper
 5. Fix initialization order in mock
 6. Upgraded internal type supports

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.5 Version 2.7 (EOL)

Version 2.7.2 (EOL)

This release includes the following **improvements**:

1. Skip `writer_removed` processing for unaccounted instances.
2. Improve `GUID_t` operator `<` performance.

This release also includes the following **bugfixes**:

1. Fix complex member printing for `DynamicDataHelper`.
2. Add `python3` dependency.
3. Fix selection of output locators.
4. Fix data races when creating `DataWriters`.
5. Fix null dereferences on XML parser.
6. Send GAPS correctly when using separate sending.
7. Install Statistics IDL file.
8. Fixes for building in old compilers.
9. Fix several deadlocks.
10. Fix communication with asymmetric *Ignore Participant flags*.
11. Fix notification lost.
12. Fix `StatelessWriter` ACK check.
13. Fix `total_read_` to be consistent with Reader's History after `DataReader::get_first_untaken_info()`.
14. Fix doxygen documentation adding deprecated to `ThroughputControllerDescriptor`.
15. Several dependencies fixes upgrading to Ubuntu 22.04.
16. Ensure `shared_mutex` implementation is consistent throughout supported platforms.
17. Fix `StatisticsSubmessageData` unaligned access.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.7.1 (EOL)

This release includes the following **features** in an ABI compatible way:

1. *Checking STATIC EDP XML Files* by means of `DomainParticipantFactory::check_xml_static_discovery()`.
2. *ReadCondition* implementation.

This release includes the following **improvements**:

1. Thread sanitizer CI.
2. Overload `get_unread_count()`.
3. Improve read/take performance when using topic with a great number of keys.

4. Improve rediscovery on lossy environments.
5. New *CMake option* `USE_THIRDPARTY_SHARED_MUTEX`.
6. Notify changes in bulk in RTPS readers.

This release includes the following **bugfixes**:

1. Fix Fast CDR submodule update to v1.0.24.
2. Fix access to some pointers.
3. Fixed validation on `ParameterPropertyList_t`.
4. Fixed acknowledgement in `DataSharing`.
5. Fixed wrong usage of `std::remove_if`.
6. Suppress OpenSSL 3.0 warnings.
7. Fixed race condition in Logging module.
8. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.7.0 (EOL)

This release includes the following **improvements**:

1. Support for *DDS SampleRejectedStatus API*
2. Support for DDS DataWriter methods:
 1. `DataWriter::write_w_timestamp()`
 2. `DataWriter::register_instance_w_timestamp()`
 3. `DataWriter::unregister_instance_w_timestamp()`
 4. `DataWriter::dispose_w_timestamp()`
3. Support for `DDS find_topic()`
4. Support for GCC 12
5. Upgrade CMake minimum requirement to 3.16.3
6. Add Windows DLL support to Dynamic Types API

Some **bugfixes** are also included:

1. Deadlocks and data races
2. Move deprecated OpenSSL cleanup function to match the right version

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.6 Version 2.5 (EOL)

Version 2.5.2 (EOL)

This release includes the following **improvements**:

1. Support lowercase keywords and hexadecimal values on SQL filter.
2. Support for GCC 12.

This release includes the following **bugfixes**:

1. Fix MatchedStatus *last_*_handle*.
2. Fix recommended statistics DataReaderQos to enable backwards compatibility.
3. Fix deadlocks and data races.
4. Fix empty partition validation checks.
5. Fix corner case with reliable writers and samples with a huge number of fragments.
6. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.5.1 (EOL)

This release includes the following **improvements**:

1. *ContentFilterTopic filtering at the DataReader side*.
2. Release lifecycle.

This release includes the following **bugfixes**:

1. XML parser fixes.
2. Discovery Server fixes.
3. Fix DataSharing sample validation.
4. PKCS#11 support fixes.
5. Test fixes.
6. Doxygen documentation fixes.
7. GAP message fix.
8. Enable memory protection on DataSharing readers.
9. TCP reconnection issues.
10. Fix dynamic network interfaces feature.
11. Several Security module fixes.
12. STRICT_REALTIME fix.
13. Suppress OpenSSL 3.0 warnings.
14. Move optionparser to thirdparty.

15. Thread-safe access to endpoints collections.
16. MemberDescriptor fully qualified name.
17. Setting QoS fix.
18. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.5.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This minor release includes the following **features**:

1. *Support for PKCS#11* format URIs for private keys
2. Added interfaces for content filter APIs
3. Allow new *network interfaces to be detected at runtime*
4. New API on *DataWriter* to wait for a specific instance to be acknowledged
5. Added interfaces for *concatenation of transports*
6. Allow *XML profiles to be loaded from a string*
7. Allow *disabling piggyback heartbeat* on XML and DataWriter QoS
8. New basic configuration example

It also includes the following **improvements**:

1. Working implementation of instance_state and view_state
2. Allow zero-valued keys
3. Made some type aliases public to ease python bindings integration
4. Improved performance by avoiding unnecessary payload copies for samples that are going to be rejected
5. Removed unnecessary headers from Log module public headers
6. Add support for Key annotation in TypeObjectFactory
7. Only export public symbols on non-windows platforms
8. Some documentation improvements

Some important **bugfixes** are also included:

1. Fixed payload pool handling on EDPSimple destructor
2. Fixed null dereference on XML parser

3. Correctly export XTypes related methods on Windows
4. Ensure correct boost singleton destruction order
5. Avoid warning when environment file filename is empty
6. Correctly set GUID of DataWriter and DataReader upon creation

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.7 Version 2.4 (EOL)

Version 2.4.2 (EOL)

This release includes the following **improvements**:

1. Enable memory protection on *DataSharing* readers
2. Add const overload of `DataReader::guid()`
3. Set recommended statistics *DataReaderQos* to `PREALLOCATED_WITH_REALLOC`
4. Allow fully qualified name on MemberDescriptor

This release includes the following bugfixes:

1. Fix and refactor `EDPSimple` destructor
2. Fix several build warnings on certain platforms
3. Fix OSS fuzz issues
4. Fix *TCP* synchronization issues
5. Correct reporting of *MatchedStatus* `last_*_handle`
6. Ensure correct boost singleton destruction order
7. Fix inserting minimum `CacheChange_t` in GAP message

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 2.4.1 (EOL)

This release includes the following **improvements**:

1. Fixed several flaky tests
2. Improved bandwidth usage of GAPs and HEARTBEATs
3. Correctly implement `delete_contained_entities`
4. Use native inter-process on Windows
5. Improved performance of `unregister_instance`
6. Improved OSS-fuzz integration
7. Support for partitions on *DataWriterQoS* and *DataReaderQoS*
8. Some documentation improvements

9. Removed unused macro to avoid naming clashes

This release includes the following **bugfixes**:

1. Avoid bad_node_size exception when cross building
2. Fixed build on old compilers
3. Fixed buffers exhaustion when compiled with statistics
4. Fixed runtime addition of Discovery Servers
5. Fixed dangling sample references with big data
6. Fixed history record issues with persistence
7. Correctly disable DataReader on destruction
8. Fixed alignment issues on XTypes QoS policies serialization
9. Fixed reconnection to Discovery Server
10. Correctly use builtin publisher for statistics DataWriters
11. Fixed various GCC-11 warnings
12. Use only public APIs from foonathan::memory
13. Fixed installation directories for DDS examples
14. Fixed read after free on security code

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.4.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This minor release includes the following **features**:

- *Conditions and Wait-sets* implementation.
- *Flow controllers*.
- *Configure Discovery Server locators using names*.
- *Modifying remote servers list at run time*.
- *Environment file override*.

It also includes the following **improvements**:

- Allow setting custom folder for *data-sharing* files.
- Allow setting persistence guid with static discovery.

- Check for NDEBUG in logInfo.
- Removed old unused CMake code.
- Fixed TLS behavior on TCP example.
- Prepare API for easy integration of python bindings.
- Improved statistics performance.

Some important **bugfixes** are also included:

- Fixed order of returned samples on topics with keys.
- Allow updating partitions to an empty set.
- Correctly propagate DomainParticipantQos updates.
- Avoid a volatile data-sharing reader to block a writer.
- Correctly give priority to intra-process over data-sharing.
- Fixed reallocation issue on LivelinessManager.
- Fixed deadline issue on volatile DataWriter
- Fixed STRICT_REALTIME silently not active with Unix POSIX systems.
- Fixed build errors with OpenSSL 3.0

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.8 Version 2.3 (EOL)

Version 2.3.6 (EOL)

This release includes the following **improvements**:

1. Improve rediscovery on lossy environments.
2. Upgrade CMake minimum requirement to 3.13.
3. Improve Guid_t operator < performance.

This release includes the following **bugfixes**:

1. Fixed validation on ParameterPropertyList_t.
2. Add python3 dependency to package.xml.
3. Fix null references and uncaught exceptions on XML parser.
4. Install Statistics IDL file.
5. Fix data races when creating DataWriters.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.3.5 (EOL)

This release includes the following **improvements**:

1. Fixed several flaky tests.
2. Use native inter-process on Windows.
3. Support for partitions on DataWriterQoS and DataReaderQoS.
4. Support for GCC 12.
5. Correctly implement delete_contained_entities.

This release also includes the following **bugfixes**:

1. Fixed deadline issue on volatile DataWriter.
2. Allow updating partitions to an empty set.
3. Fixed order of returned samples on topics with keys.
4. Fixed issues in LivelinessManager.
5. Correctly give priority to intra-process over data-sharing.
6. Avoid bad_node_size exception when cross-building.
7. Fixed build errors with OpenSSL 3.0.
8. Avoid a volatile data-sharing reader to block a writer.
9. Fixed history record issues with persistence.
10. Correctly disable DataReader on destruction.
11. Fixed various GCC 11 warnings.
12. Fixed payload pool handling on EDPSimple destructor.
13. Fixed read after free on security code.
14. Fixed null dereference on XML parser.
15. Ensure correct boost singleton destruction order.
16. Enable memory protection on DataSharing readers.
17. TCP reconnection issues.
18. MemberDescriptor fully qualified name.
19. Fix recommended statistics DataReaderQos to enable backwards compatibility.
20. Fixed dangling sample references with big data.
21. Fixed deadlocks and data races.
22. Fixed reconnection to Discovery Server.
23. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.3.4 (EOL)

This release includes the following **improvements**:

1. Support of googletest using colcon
2. Network latency reports source participant
3. Update Fast DDS Gen to v2.0.2

This release includes the following **bugfixes**:

1. Fix mutex lock count on PDPListener
2. Limit SequenceNumberSet number of bits on deserialization
3. Fix segmentation fault on discovery server
4. Fix deadlock with security and timers
5. Fix bug using not protected code in a test
6. Fix deadlock with LivelinessManager
7. Fix interval loop on events
8. Fix run event when was cancelled
9. Validate sequence range on CDRMessage::readSequenceNumberSet
10. Fix subscription throughput data generation
11. Allow examples to build on QNX
12. Fix code on SHM clean
13. Accept Statistics DataWriters in Discovery Server
14. Fix read/take behavior when a future change is found
15. Correctly handle deserialization errors on read_next_sample() / take_next_sample()
16. Fixing SequenceNumberSet_t deserialization
17. Proper history clean up when a reader unmatched a writer
18. Unprotected code loaning samples
19. Fix publication throughput statistic on volatile writers
20. Fix Fast DDS CLI server name
21. Several fixes in examples and tests

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.3.3 (EOL)

This release includes the following **improvements**:

- Added more durability kinds in Static Discovery xml parser
- Explicitly enable/disable data-sharing on performance tests
- Allow fully qualified name in TypeDescriptor
- Added missing DynamicData::get_union_id() method
- Change log severity in DiscoveryServer first announcement
- Several corrections to README

This release includes the following **bugfixes**:

- Fixed warnings and segfaults on 32-bit platforms
- Fixed UDPv6 behavior
- Fixed persistence guid issue on statistics writers
- Fixed static linking with open SSL
- Fixed statistics header file inclusion
- Fixed build on RedHat systems

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.3.2 (EOL)

This release includes the following **feature**:

- Statistics Module

It also includes the following **improvements**:

- Update Asio submodule and avoid exporting Asio API
- Improve Windows installers
- Ease Google Fuzz integration
- Improve Doxygen documentation on lifetime of pointers created with RTPSDomain
- Update Fast CDR to v1.0.21

This release includes the following **bugfixes**:

- Add a correct multicast address for UDPv6
- Recover from out-of-sync TCP datagrams

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.3.1 (EOL)

This release includes several **bugfixes** and **improvements**:

- Added *Fast DDS Statistics Module* implementation
- Fixed alignment issues on generated code calculation of maximum serialized size
- Fixed calculation of data-sharing domain id
- Fixed issues on data-sharing with volatile writers
- Fixed build issues on old compilers
- Fixed some tests when the library is built without security
- Fixed and exposed pull mode on writers
- Fixed handling of `-data_sharing` on latency test
- Fixed calculation of memory pools sizes on debug builds
- Correctly update memory policy on writers and readers

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.3.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This release adds the following **features**:

- *Unique network flows*
- *Discovery super-client*
- *Statistics module API*
- *New flow controller API*
- *Static discovery configuration from raw string*
- *Added reception timestamp to SampleInfo*
- *Exposing get_unread_count on DataReader*

It also includes the following **improvements**:

- Data-sharing delivery internal refactor
- Additional metadata on persistence databases
- Refactor on ReturnCode_t to make it switch friendly
- Performance tests refactored to use DDS-PIM high-level API

- Receive const pointers on delete_xxx methods
- Discovery server improvements
- Made SOVERSION follow major.minor

Some important **bugfixes** are also included:

- Fixed shared memory usage on QNX
- Fixed reference counting on internal pools
- Fixed singleton destruction order
- Fixed interoperability issues with x-types information
- Fixed recovery of shared memory buffers
- Lifespan support in persistent writers

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.9 Version 2.2 (EOL)

Version 2.2.1 (EOL)

This release includes the following **improvements**:

1. Data-sharing delivery internal refactor.
2. Performance tests refactored to use DDS-PIM high-level API.
3. Discovery server improvements.

This release includes the following **bugfixes**:

1. Fixed reference counting on internal pools.
2. Fixed singleton destruction order.
3. Fixed default multicast locators.
4. Fixed interoperability issues with x-types information.
5. Fixed Reader history issues.
6. Fixed data races issues.
7. Fixed shared memory issues.
8. Fixed heartbeat and ACK issues.
9. Fixed LivelinessManager issues.
10. Fixed TCP reception synchronization.
11. Fixed build issues on old compilers.
12. Allow modifying Partition QoS in enabled entities.
13. Other minor fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

Version 2.2.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

This release adds the following **features**:

- Data Sharing delivery (avoids transport encapsulation for localhost communications)
- Complete DDS-PIM high-level API declarations
- Extension APIs allowing zero-copy delivery (both intra-process and inter-process)
- Upgrade to Quality Level 1

It also includes the following **improvements**:

- Code coverage policy
- Added several tests to increase coverage
- Increased GUID uniqueness
- Allow logInfo messages to be compiled on build types other than debug

Some important **bugfixes** are also included:

- Fixed timed events manager race condition
- Fixed payload protection issues with SHM transport
- Writers correctly handle infinite resource limits on keyed topics
- Fixed unsafe code on AESGCMGMAC plugin
- Several fixes for IPv6 (whitelists, address parser)
- Fixes on liveliness timing handling
- Fixed warnings building on C++20

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from any older version, regenerating the code is *highly recommended*.

7.45.10 Version 2.1 (EOL)

Version 2.1.4 (EOL)

This release includes the following **improvements**:

1. Improve auto GAPs in Data Sharing
2. Improve behavior when `STRICT_REALTIME` *CMake option* is not enabled
3. Handle SIGTERM in fast discovery server CLI

This release includes the following **bugfixes**:

1. Select correct listener for `on_requested_deadline_missed`
2. Correctly assign multicast port to multicast initial peers
3. Fix chain of trust issues with a single CA certificate
4. Correctly handle builtin endpoints mask
5. Fix build on MSVC 19

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 2.1.3 (EOL)

This release includes the following **improvements**:

1. Improve rediscovery on lossy environments.

This release includes the following **bugfixes**:

1. Fixed validation on `ParameterPropertyList_t`.
2. Fixed `SequenceNumberSet_t` deserialization.
3. Add python3 dependency to package.xml.
4. Fix data races when creating `DataWriters`.
5. Fix deadlock when removing remote `DomainParticipants` by expired liveliness when using Security.
6. Fix communication with asymmetric *Ignore Participant flags*.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 2.1.2 (EOL)

This release includes the following **improvements**:

1. Allow fully qualified name in TypeDescriptor.
2. Use native inter-process on Windows.
3. Support for GCC 12.
4. Support of googletest using colcon.

This release also includes the following **bugfixes**:

1. Fixed recovery of shared memory buffers.
2. Fixed issues in LivelinessManager.
3. Fixed default multicast locators.
4. Fixed TCP issues.
5. Fixed deadlocks and data races.
6. Fixed deadline issue on volatile DataWriter.
7. Avoid bad_node_size exception when cross-building.
8. Fixed order of returned samples on topics with keys.
9. Allow updating partitions to an empty set.
10. Suppress OpenSSL 3.0 warnings.
11. MemberDescriptor fully qualified name.
12. Fixed history record issues with persistence.
13. Fixed reconnection to Discovery Server.
14. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 2.1.1 (EOL)

This release includes the following **bugfixes**:

- Fixed race condition on security handshake
- Fixed SHM data corruption when using both payload and sub-message protection
- Fixed some interoperability issues
- Fixed race condition on timed-events thread
- Fixed usage of SHM on QNX systems

It also includes the following **improvements**:

- Increased uniqueness of GUID prefix
- Discovery server improvements

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 2.1.0 (EOL)

This minor release is API compatible with the previous minor release, but introduces **ABI breaks** on two of the three public APIs:

- Methods and attributes have been added on several classes of the DDS-PIM high-level API, so indexes of symbols on dynamic libraries may have changed.
- Methods and attributes have been added on several classes of the RTPS low-level API, so indexes of symbols on dynamic libraries may have changed.
- Old Fast-RTPS high-level API remains ABI compatible.

Users of the RTPS low-level API should also be aware of the following **API deprecations**:

- History::reserve_Cache has been deprecated
 - Methods RTPSWriter::new_change or RTPSReader::reserveCache should be used instead
- History::release_Cache has been deprecated
 - Methods RTPSWriter::release_change or RTPSReader::releaseCache should be used instead

This release adds the following **features**:

- Support persistence for large data
- Added support for *on_requested_incompatible_qos* and *on_offered_incompatible_qos*
- SKIP_DEFAULT_XML environment variable
- Added FORCE value to THIRDPARTY cmake options
- New log consumer (StdOutErrConsumer)
- Added methods to get qos defined in XML Profile
- Support for persistence on TRANSIENT_LOCAL

It also includes the following **improvements**:

- Internal refactor for intra-process performance boost
- Allow usage of foonathan/memory library built without debug tool
- Large data support on performance tests
- Reduced flakiness of several tests

Some important **bugfixes** are also included:

- Fixed behavior of several DDS API methods
- Fixed interoperability issues with RTI connect
- 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 *fastdds-gen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

7.45.11 Version 2.0 (EOL)

Version 2.0.3 (EOL)

It also includes the following **improvements**:

1. Increased uniqueness of GUID prefix (#1648)
2. Upgrade Fast CDR to v1.0.20 (#1793)

This release includes the following **bugfixes**:

1. Fixed some race conditions (#1540, #2023)
2. Fixed SHM issues (#1644, #1895, #2266)
3. Fixed some interoperability issues (#1624, #1752, #1849)
4. Fixed Discovery Server 2.0 issues (#1639, #1651, #1761, #1796)
5. Fixed several issues on QNX systems (#1744, #1773, #1776)
6. Fix singleton destruction order (#1758)
7. Fix heartbeat and ACK issues (#1865)
8. Fix issues in LivelinessManager (#1872, #2147)
9. Fix multicast issues (#1966, #1905)
10. Fix TCP reception synchronization (#1981)
11. XTypes standard compliance and fixes (#2006, #2278)
12. Other minor fixes (#1558, #1734, #1814, #1935, #1978, #2121)

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 2.0.2 (EOL)

This release includes the following improvements:

- Improve QNX support
- Security improvements
- Fast DDS Quality Declaration (QL 2)
- Large traffic reduction when using Discovery Server (up to 85-90% for large deployments)
- Configuration of Clients of Discovery Server using an environment variable
- A CLI for Fast DDS:
 - This can be used to launch a discovery server
 - Clean SHM directories with one command
- Shared memory transport enabled by default
- Solved edge-case interoperability issue with CycloneDDS
- Add package.xml

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 2.0.1 (EOL)

This release includes the following bug fixes:

- Fixed sending GAPs to late joiners
- Fixed asserting liveness on data reception
- Avoid calling `OpenSSL_add_all_algorithms()` when not required

Other improvements:

- Fixing warnings

PRs in merge order: #1295, #1300, #1304, #1290, #1307.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 2.0.0 (EOL)

This release has the following **API breaks**:

- eClock API, which was deprecated on v1.9.1, has been removed
- *eprosima::fastrtps::rtps::RTPSDomain::createParticipant* methods now have an additional first argument *domain_id*
- Data member *domainId* has been removed from *eprosima::fastrtps::rtps::RTPSParticipantAttributes* and added to *eprosima::fastrtps::ParticipantAttributes*

Users should also be aware of the following **deprecation announcement**:

- All classes inside the namespace *eprosima::fastrtps* should be considered deprecated. Equivalent functionality is offered through namespace *eprosima::fastdds*.
- Namespaces beneath *eprosima::fastrtps* are not included in this deprecation, i.e. *eprosima::fastrtps::rtps* can still be used)

This release adds the following **features**:

- Added support for register/unregister/dispose instance
- Added DDS compliant API. This new API exposes all the functionality of the Publisher-Subscriber Fast RTPS API adhering to the [Data Distribution Service \(DDS\) version 1.4 specification](#)
- Added Security Logging Plugin (contributed by Canonical 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

7.45.12 Version 1.10 (EOL)

Version 1.10.1 (EOL)

This release includes the following **improvements**:

1. Add new CMake option: *SHM_TRANSPORT_DEFAULT*. Shared Memory (SHM) Transport disabled by default.
2. Comply with the RTPS standard concerning *PID_SENTINEL*.
3. Support for OpenSSL 1.1.1d.

This release includes the following bugfixes:

1. Fix crash when there are no network interfaces.
2. Several Shared Memory Transport fixes.
3. Other minor fixes.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

Version 1.10.0 (EOL)

This release adds the following features:

- New built-in *Shared Memory Transport*
- Transport API refactored to support locator iterators
- Added subscriber API to retrieve info of first non-taken sample
- Added parameters to fully avoid dynamic allocations
- History of built-in endpoints can be configured
- Bump to FastCDR v1.0.13.
- Bump to Fast-RTPS-Gen v1.0.4.
- Require CMake 3.5 but use policies from 3.13

It also includes the following bug fixes and improvements:

- Fixed alignment on parameter lists
- Fixed error sending more than 256 fragments.
- Fix handling of STRICT_REALTIME.
- Fixed submessage_size calculation on last data_frag.
- Solved an issue when recreating a publishing participant with the same GUID.
- Solved an issue where a publisher could block on write for a long time when a new subscriber (late joiner) is matched, if the publisher had already sent a large number of messages.
- Correctly handling the case where lifespan expires at the same time on several samples.
- Solved some issues regarding liveliness on writers with no readers.
- Correctly removing changes from histories on keyed topics.
- Not reusing cache change when sample does not fit.
- Fixed custom wait_until methods when time is in the past.
- Several data races and ABBA locks fixed.
- Reduced CPU and memory usage.
- Reduced flakiness of liveliness tests.
- Allow for more use cases on performance tests.

Several bug fixes on discovery server:

- Fixed local host communications.
- Correctly trimming server history.
- Fixed backup server operation.
- Fixed timing issues.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

7.45.13 Version 1.9 (EOL)

Version 1.9.5 (EOL)

This release includes the following **improvements**:

1. Propagate serialization error when reading samples from Subscriber History.
2. Improvements in test suite.
3. Improvements to reduce memory consumption.
4. Update CMake (3.5) using newer policies.
5. Improve PDP StatefulWriter announcement.
6. Performance improvements.
7. Message receiver improvements.
8. QoS Policies improvements.

This release includes the following bugfixes:

1. Fix compiler warnings in Windows when building the test suite.
2. Fix several data races.
3. Fix History issues.
4. Fix errors when sending data fragments.
5. Fix strict real-time behavior.
6. Fix in Discovery Server.
7. Fix CMake option.
8. Fix interoperability issues.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.9.4 (EOL)

This release adds the following features:

- Intra-process delivery mechanism is now active by default.
- Synchronous writers are now allowed to send fragments.
- New memory mode DYNAMIC_RESERVE on history pool.
- Performance tests can now be run on Windows and Mac.
- XML profiles for requester and replier.

It also includes the following bug fixes and improvements:

- Bump to FastCDR v1.0.12.
- Bump to Fast-RTPS-Gen v1.0.3.
- Fixed deadlock between PDP and StatefulReader.
- Improved CPU usage and allocations on timed events management.

- Performance improvements on reliable writers.
- Fixing bugs when Intra-process delivery is activated.
- Reducing dynamic allocations and memory footprint.
- Improvements and fixes on performance tests.
- Other minor bug fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.9.3 (EOL)

This release adds the following features:

- Participant discovery filtering flags.
- Intra-process delivery mechanism opt-in.

It also includes the following bug fixes and improvements:

- Bump to Fast-RTPS-Gen v1.0.2.
- Bring back compatibility with XTypes 1.1 on PID_TYPE_CONSISTENCY.
- Ensure correct alignment when reading a parameter list.
- Add CHECK_DOCUMENTATION *cmake* option.
- EntityId_t and GuidPrefix_t have now their own header files.
- Fix potential race conditions and deadlocks.
- Improve the case where *check_acked_status* is called between reader matching process and its acknack reception.
- RTPSMessageGroup_t instances now use the thread-local storage.
- FragmentedChangePitStop manager removed.
- Remove the data fragments vector on CacheChange_t.
- Only call find_package for TinyXML2 if third-party options are off
- Allow XMLProfileManager methods to not show error log messages if a profile is not found.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.9.2 (EOL)

This release includes the following feature:

- Multiple initial PDP announcements.
- Flag to avoid builtin multicast.

It also adds the following bug fixes and improvements:

- Bump to Fast-RTPS-Gen v1.0.1.
- Bump to IDL-Parser v1.0.1.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.9.1 (EOL)

This release includes the following features:

- Fast-RTPS-Gen is now an independent project.
- Header **eClock.h** is now marked as deprecated.

It also adds the following bug fixes and improvements:

- Bump to FastCDR v1.0.11.
- Installation from sources documentation fixed.
- Fixed assertion on WriterProxy.
- Fixed potential fall through while parsing Parameters.
- Removed deprecated guards causing compilation errors in some 32 bits platforms.
- *addTOCDRMessage* method is now exported in the DLL, fixing issues related with Parameters' constructors.
- Improve windows performance by avoiding usage of *_Cnd_timedwait* method.
- Fixed reported communication issues by sending multicast through *localhost* too.
- Fixed potential race conditions and deadlocks.
- Eliminating use of *acceptMsgDirectTo*.
- Discovery Server framework reconnect/recreate strategy.
- Removed unused folders.
- Restored subscriber API.
- SequenceNumber_t improvements.
- Added STRICT_REALTIME *cmake* option.
- SubscriberHistory improvements.
- Assertion of participant liveliness by receiving RTPS messages from the remote participant.
- Fixed error while setting next deadline event in *create_new_change_with_params*.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.9.0 (EOL)

This release includes the following features:

- Partial implementation of allocation QoS.
- Implementation of Discovery Server.
- Implementation of non-blocking calls.

It also adds the following bug fixes and improvements:

- Added sliding window to BitmapRange.
- Modified default behavior for unknown writers.
- A *Flush()* method was added to the logger to ensure all info is logged.
- A test for loading *Duration_t* from XML was added.

- Optimized WLP when removing local writers.
- Some liveliness tests were updated so that they are more stable on Windows.
- A fix was added to *CMakeLists.txt* for installing static libraries.
- A fix was added to performance tests so that they can run on the RT kernel.
- Fix for race condition on built-in protocols creation.
- Fix for setting *nullptr* in a *fixed_string*.
- Fix for v1.8.1 not building with `-DBUILD_JAVA=ON`.
- Fix for GAP messages not being sent in some cases.
- Fix for coverity report.
- Several memory issues fixes.
- *fastrtps.repos* file was updated.
- Documentation for building with Colcon was added.
- Change CMake configuration directory if `INSTALLER_PLATFORM` is set.
- IDL sub-module updated to current version.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

7.45.14 Version 1.8 (EOL)

Version 1.8.5 (EOL)

This release includes the following **bugfixes**:

1. Fix Subscriber History to correctly notify late-joiners in case of `KEEP_LAST`, `RELIABLE`, and `TRANSIENT_LOCAL`.
2. Fix Writer History behavior when there are no matched readers.
3. Fix heartbeat and ACK issues.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.8.4 (EOL)

This release adds the following **feature**:

- XML profiles for *requester* and *replier*

It also has the following **important bug fixes**:

- Solved an issue when recreating a publishing participant with the same GUID (either on purpose or by chance)
- Solved an issue where a publisher could block on *write* for a long time when, after a large number of samples have been sent, a new subscriber is matched.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*

Version 1.8.3 (EOL)

This release adds the following bug fixes and improvements:

- Fix serialization of `TypeConsistencyEnforcementQosPolicy`.
- Bump to Fast-RTPS-Gen v1.0.2.
- Bump to IDL-Parser v1.0.1.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*

Version 1.8.2 (EOL)

This release includes the following features:

- Modified unknown writers default behavior.
- Multiple initial PDP announcements.
- Flag to avoid builtin multicast.
- *STRICT_REALTIME* compilation flag.

It also adds the following bug fixes and improvements:

- Fix for setting *nullptr* in a fixed string.
- Fix for not sending GAP in several cases.
- Solve *Coverity* report issues.
- Fix issue of *fastddsgen* failing to open *IDL.g4* file.
- Fix unnamed lock in *AESGCMGMAC_KeyFactory.cpp*.
- Improve *XMLProfiles* example.
- Multicast is now sent through *localhost* too.
- *BitmapRange* now implements sliding window.
- Improve *SequenceNumber_t* struct.
- Participant's liveliness is now asserted when receiving any RTPS message.
- Fix leak on *RemoteParticipantLeaseDuration*.
- Modified default values to improve behavior in *Wi-Fi* scenarios.
- *SubscriberHistory* improvements.
- Removed use of *acceptMsgDirectTo*.
- *WLP* improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*

Version 1.8.1 (EOL)

This release includes the following features:

- Implementation of *LivelinessQosPolicy* QoS.

It also adds the following bug fixes and improvements:

- Fix for get_change on history, which was causing issues during discovery.
- Fix for announcement of participant state, which was sending ParticipantBuiltinData twice.
- Fix for closing multicast UDP channel.
- Fix for race conditions in SubscriberHistory, UDPTransportInterface and StatefulReader.
- Fix for lroundl error on Windows in Time_t.
- CDR & IDL submodules update.
- Use of java 1.8 or greater for fastddsgen.jar generation.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.8.0 (EOL)

This release included the following features:

- Implementation of IDL 4.2.
- Implementation of *DeadlineQosPolicy* QoS.
- Implementation of *LifespanQosPolicy* QoS.
- Implementation of *DisablePositiveACKsQosPolicy* QoS.
- Secure sockets on TCP transport (*TLS over TCP*).

It also adds the following improvements and bug fixes:

- Real-time improvements: non-blocking write calls for best-effort writers, addition of fixed size strings, fixed size bitmaps, resource limited vectors, etc.
- Duration parameters now use nanoseconds.
- Configuration of participant mutation tries.
- Automatic calculation of the port when a value of 0 is received on the endpoint custom locators.
- Non-local addresses are now filtered from whitelists.
- Optimization of check for acked status for stateful writers.
- Linked libs are now not exposed when the target is a shared lib.
- Limitation on the domain ID has been added.
- UDP non-blocking send is now optional and configurable via XML.
- Fix for non-deterministic tests.
- Fix for ReaderProxy history being reloaded incorrectly in some cases.
- Fix for RTPS domain hostid being potentially not unique.
- Fix for participants with different lease expiration times failing to reconnect.

Known issues

- When using TPC transport, sometimes callbacks are not invoked when removing a participant due to a bug in ASIO.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

7.45.15 Version 1.7 (EOL)

Version 1.7.3 (EOL)

This release includes the following **bugfixes**:

1. Remove inline specifier from public method not defined in header file.
2. Fix FastRTPS-Gen version generation

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.7.2 (EOL)

This release fixes an important bug:

- Allocation limits on subscribers with a KEEP_LAST QoS was taken from resource limits configuration and didn't take history depth into account.

It also has the following improvements:

- Vendor FindThreads.cmake from CMake 3.14 release candidate to help with sanitizers.
- Fixed format of gradle file.

Some other minor bugs and performance improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastddsgen*.

Version 1.7.1 (EOL)

This release included the following features:

- LogFileConsumer added to the logging system.
- Handle FASTRTPS_DEFAULT_PROFILES_FILE environment variable indicating the default profiles XML file.
- XML parser made more restrictive and with better error messages.

It also fixes some important bugs: * Fixed discovery issues related to the selected network interfaces on Windows. * Improved discovery times. * Workaround ASIO issue with multicast on QNX systems. * Improved TCP transport performance. * Improved handling of key-only data submessages.

Some other minor bugs and performance improvements.

KNOWN ISSUES

- Allocation limits on subscribers with a KEEP_LAST QoS is taken from resource limits configuration and doesn't take history depth into account.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastdds-gen*.

Version 1.7.0 (EOL)

This release included the following features:

- *TCP Transport*.
- *XTypes*.
- 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 *fastdds-gen*.

7.45.16 Version 1.6 (EOL)

Version 1.6.0 (EOL)

This release included the following features:

- Persistence.
- Security access control plugin API and builtin *Access control plugin: DDS:Access:Permissions* plugin.

Also bug fixing.

Note: If you are upgrading from an older version than 1.4.0, it is advisable to regenerate generated source from IDL files using *fastdds-gen*.

7.45.17 Version 1.5 (EOL)

Version 1.5.0 (EOL)

This release included the following features:

- Configuration of Fast RTPS entities through XML profiles.
- Added heartbeat piggyback support.

Also bug fixing.

Note: If you are upgrading from an older version than 1.4.0, it is advisable to regenerate generated source from IDL files using *fastdds-gen*.

7.45.18 Version 1.4 (EOL)

Version 1.4.0 (EOL)

This release included the following:

- Added secure communications.
- Removed all Boost dependencies. Fast RTPS is not using Boost libraries anymore.
- Added compatibility with Android.
- Bug fixing.

Note: After upgrading to this release, it is advisable to regenerate generated source from IDL files using *fastdds-gen*.

7.45.19 Version 1.3 (EOL)

Version 1.3.1 (EOL)

This release included the following:

- New examples that illustrate how to tweak Fast RTPS towards different applications.
- Improved support for embedded Linux.
- Bug fixing.

Version 1.3.0 (EOL)

This release introduced several new features:

- Unbound Arrays support: Now you can send variable size data arrays.
- Extended Fragmentation Configuration: It allows you to setup a Message/Fragment max size different to the standard 64Kb limit.
- Improved logging system: Get even more introspection about the status of your communications system.
- Static Discovery: Use XML to map your network and keep discovery traffic to a minimum.
- Stability and performance improvements: A new iteration of our built-in performance tests will make benchmarking easier for you.
- ReadTheDocs Support: We improved our documentation format and now our installation and user manuals are available online on ReadTheDocs.

7.45.20 Version 1.2 (EOL)

Version 1.2.0 (EOL)

This release introduced two important new features:

- Flow Controllers: A mechanism to control how you use the available bandwidth avoiding data bursts. The controllers allow you to specify the maximum amount of data to be sent in a specific period of time. This is very useful when you are sending large messages requiring fragmentation.

- Discovery Listeners: Now the user can subscribe to the discovery information to know the entities present in the network (Topics, Publishers & Subscribers) dynamically without prior knowledge of the system. This enables the creation of generic tools to inspect your system.

But there is more:

- Full ROS 2 Support: Fast RTPS is used by ROS 2, the upcoming release of the Robot Operating System (ROS).
- Better documentation: More content and examples.
- Improved performance.
- Bug fixing.