FastRTPS Documentation

Release 2.0.0

eProsima

Nov 11, 2020

INSTALLATION MANUAL

1 DDS API **3**

2 Fast DDS-Gen **5**

3 RTPS Wire Protocol **7**

4 Main Features **9**

5 Contacts and Commercial support **11**

6 Structure of the documentation **13**

6.1 Requirements . . . 13
6.2 Installation from Binaries . . . 14
6.3 Installation from Sources . . . 14
6.4 Getting Started . . . 16
6.5 Library Overview . . . 38
6.6 DDS Layer . . . 42
6.7 RTPS Layer . . . 155
6.8 Discovery . . . 159
6.9 Transports . . . 186
6.10 Security . . . 200
6.11 Logging . . . 226
6.12 XML profiles . . . 229
6.13 Environment variables . . . 275
6.14 Dynamic Topic Types . . . 275
6.15 Typical Use-Cases . . . 294
6.16 Advanced Functionalities . . . 312
6.17 Real-time behavior . . . 322
6.18 Persistence . . . 326
6.19 Code generation using fastrtpsgen . . . 328
6.20 API Reference . . . 329
6.21 Introduction . . . 428
6.22 Building a publish/subscribe application . . . 429
6.23 Execution and IDL Definition . . . 431
6.24 Version 2.0.0 . . . 439
6.25 Previous versions . . . 440

Index **451**

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:

- The *DDS API* implementation.
- *Fast DDS-Gen*, a generation tool for bridging typed interfaces with the middleware implementation.
- 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).

CHAPTER

ONE

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.

CHAPTER

TWO

FAST DDS-GEN

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

CHAPTER

THREE

RTPS WIRE PROTOCOL

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

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

CHAPTER

FOUR

MAIN FEATURES

- **Two API Layers.** *eProsima Fast DDS* comprises a high-level DDS compliant layer focused on usability and a lower-level RTPS compliant layer that provides finer access to the RTPS protocol.
- **Real-Time behaviour.** *eProsima Fast DDS* can be configured to offer real-time features, guaranteeing responses within specified time constrains.
- **Built-in Discovery Service.** *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.
- **Throughput controllers.** We support user-configurable throughput controllers, that can be used to limit the amount of data to be sent under certain conditions.
- **Plug-and-play Connectivity.** New applications and services are automatically discovered, and can join and leave the network at any time without the need for reconfiguration.
- **Scalability and Flexibility.** DDS builds on the concept of a global data space. The middleware is in charge of propagating the information between publishers and subscribers. This guarantees that the distributed network is adaptable to reconfigurations and scalable to a large number of entities.
- **Application Portability.** The DDS specification includes a platform specific mapping to IDL, allowing an application using DDS to switch among DDS implementations with only a re-compile.
- **Extensibility.** *eProsima Fast DDS* allows the protocol to be extended and enhanced with new services without breaking backwards compatibility and interoperability.
- **Configurability and Modularity.** *eProsima Fast DDS* provides an intuitive way to be configured, either through code or XML profiles. Modularity allows simple devices to implement a subset of the protocol and still participate in the network.
- **High performance.** *eProsima Fast DDS* uses a static low-level serialization library, Fast CDR, a C++ library that serializes according to the standard CDR serialization mechanism defined in the RTPS Specification (see the Data Encapsulation chapter as a reference).

- **Easy to use.** The project comes with an out-of-the-box example, the *DDSHelloWorld* (see *Getting Started*) that puts into communication a publisher and a subscriber, showcasing how *eProsima Fast DDS* is deployed. Additionally, the interactive demo *ShapesDemo* is available for the user to dive into the DDS world. The DDS and the RTPS layers are thoroughly explained in the *DDS Layer* and *RTPS Layer* sections.
- **Low resources consumption.** *eProsima Fast DDS*:
  - Allows to preallocate resources, to minimize dynamic resource allocation.
  - Avoids the use of unbounded resources.
  - Minimizes the need to copy data.
- **Multi-platform.** The OS dependencies are treated as pluggable modules. The user can easily implement his platform modules to *eProsima Fast DDS* library in his specific platform. By default, the project can run over Linux, Windows and MacOS.
- **Free and Open Source.** The Fast DDS library, the underneath RTPS library, the generator tool, the internal dependencies (such as *eProsima Fast CDR*) and the external ones (such as the *foonathan* library) are free and open source.

CHAPTER

FIVE

CONTACTS AND COMMERCIAL SUPPORT

Find more about us at eProsima's webpage.

Support available at:

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

CHAPTER

SIX

STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the sections below.

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

6.1 Requirements

eProsima Fast RTPS requires the following packages to work.

6.1.1 Common Dependencies

Gtest

Gtest is needed to compile the tests when building from sources.

Java & Gradle

Java & gradle is required to make use of our built-in code generation tool *fastrtpsgen* (see *Compile*).

6.1.2 Windows 7 32-bit and 64-bit

Visual C++ 2015 or 2017 Redistributable Package

eProsima Fast RTPS requires the Visual C++ Redistributable packages for the Visual Studio version you choose during the installation or compilation. The installer gives you the option of downloading and installing them.

6.2 Installation from Binaries

You can always download the latest binary release of *eProsima Fast RTPS* from the company website.

6.2.1 Windows 7 32-bit and 64-bit

Execute the installer and follow the instructions, choosing your preferred Visual Studio version and architecture when prompted.

Environmental Variables

eProsima Fast RTPS requires the following environmental variable setup in order to function properly

- FASTRTPSHOME: Root folder where *eProsima Fast RTPS* is installed.
- Additions to the PATH: the /bin folder and the subfolder for your Visual Studio version of choice should be appended to the PATH.

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

6.2.2 Linux

Extract the contents of the package. It will contain both *eProsima Fast RTPS* and its required package *eProsima Fast CDR*. You will have to follow the same procedure for both packages, starting with *Fast CDR*.

Configure the compilation:

```
$ ./configure --libdir=/usr/lib
```

If you want to compile with debug symbols (which also enables verbose mode):

```
$ ./configure CXXFLAGS="-g -D__DEBUG"  --libdir=/usr/lib
```

After configuring the project compile and install the library:

```
$ sudo make install
```

6.3 Installation from Sources

6.3.1 Dependencies

Asio and TinyXML2 libraries

On Linux, you can install these libraries using the package manager of your Linux distribution. For example, on Ubuntu you can install them by using its package manager with the next command.

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

On Windows, you can install these libraries using Chocolatey. First, download the following chocolatey packages from this ROS2 Github repository.

- asio.1.12.1.nupkg

- tinyxml2.6.0.0.nupkg

Once these packages are downloaded, open an administrative shell and execute the following command:

```
choco install -y -s <PATH\TO\DOWNLOADS\> asio tinyxml2
```

Please replace `<PATH\TO\DOWNLOADS>` with the folder you downloaded the packages to.

6.3.2 Colcon installation

colcon is a command line tool to build sets of software packages. This section explains to use it to compile easily Fast-RTPS and its dependencies. First install ROS2 development tools (colcon and vcstool):

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

Download the repos file that will be used to download Fast RTPS and its dependencies:

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

Finally, use colcon to compile all software:

```
$ colcon build
```

6.3.3 Manual installation

Before compiling manually Fast RTPS you need to clone the following dependencies and compile them using CMake.

- Fast CDR

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

- Foonathan memory

```
$ git clone https://github.com/eProsima/foonathan_memory_vendor.git
$ cd foonathan_memory_vendor
$ mkdir build && cd build
$ cmake ..
$ cmake --build . --target install
```

Once all dependencies are installed, you will be able to compile and install Fast RTPS.

```
$ git clone https://github.com/eProsima/Fast-RTPS.git
$ mkdir Fast-RTPS/build && cd Fast-RTPS/build
$ cmake ..
$ cmake --build . --target install
```

If you want to compile the examples, you will need to add the argument `-DCOMPILE_EXAMPLES=ON` when calling CMake.

If you want to compile the performance tests, you will need to add the argument `-DPERFORMANCE_TESTS=ON` when calling CMake.

For generate *fastrtpsgen* please see *Compile*.

6.3.4 Fast-RTPS-gen

If you want to compile *fastrtpsgen* java application, you will need to download its source code from the Fast-RPTS-Gen repository and with `--recursive` option and compile it calling `gradle assemble`. For more details see *Compile*.

6.3.5 Security

By default, Fast RTPS doesn't compile security support. You can activate it adding `-DSECURITY=ON` at CMake configuration step. More information about security on Fast RTPS, see *Security*.

When security is activated on compilation Fast RTPS builds several built-in security plug-ins. Some of them have the dependency of OpenSSL library.

OpenSSL installation on Linux

Surely you can install OpenSSL using the package manager of your Linux distribution. For example, on Ubuntu you can install OpenSSL using its package manager with next command.

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

OpenSSL installation on Windows

You can download OpenSSL 1.0.2 for Windows in this webpage. This is the OpenSSL version tested by our team. Download and use the installer that fits your requirements. After installing, add the environment variable `OPENSSL_ROOT_DIR` pointing to the installation root directory. For example:

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

6.4 Getting Started

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

6.4.1 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. 1: DCPS model entities in the DDS Domain.

6.4.2 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. 2: 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, FastDDS supports numerous configurations that allow you to change the behavior of RTPSWriters/RTPSReaders. A modification in the default configuration of the RTPS entities implies a change in the data exchange flow between RTPSWriters and RTPSReaders. Moreover, by choosing Quality of Service (QoS) policies, you can affect how these history caches are managed in several ways, but the communication loop remains the same. You can continue reading section *RTPS Layer* to learn more about the implementation of the RTPS protocol in Fast DDS.

6.4.3 Writing a simple publisher and subscriber application

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

- *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 *Requirements* for the installation of required packages, and in *Installation from Sources* for the installation of *eProsima Fast DDS* and all its dependencies. You also need to have completed the steps outlined in *Compile* for the installation of the *eProsima Fast DDS-Gen* tool. Moreover, all the commands provided in this tutorial are outlined for a Linux environment.

Create the application workspace

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

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

Let's create the directory tree first.

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

Import linked libraries and its dependencies

The DDS application requires the Fast DDS and Fast CDR libraries. The way we will make these accessible from the workspace depends on the installation procedure we have followed in *Installation from Binaries* or in *Installation from Sources*.

Installation from binaries and manual installation

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

Colcon installation

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

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

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

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

Configure the CMake project

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

```
cmake_minimum_required(VERSION 3.12.4)

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

project(DDSHelloWorld)

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

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

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

(continues on next page)

(continued from previous page)

```
        add_compile_options(-std=c++11)
    else()
        message(FATAL_ERROR "Compiler doesn't support C++11")
    endif()
endif()
```

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

Build the topic data type

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

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

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

In the workspace directory, execute the following commands:

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

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

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

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

```
<path/to/Fast-RTPS-Gen>/scripts/fastrtpsgen HelloWorld.idl
```

This must have generated the following files:

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

CMakeLists.txt

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

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

Write the Fast DDS publisher

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

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

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

```
// Copyright 2016 Proyectos y Sistemas de Mantenimiento SL (eProsima).
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

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

#include "HelloWorldPubSubTypes.h"

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

using namespace eprosima::fastdds::dds;

class HelloWorldPublisher
{
private:

    HelloWorld hello_;

    DomainParticipant* participant_;

```

(continues on next page)

(continued from previous page)

```
    Publisher* publisher_;

    Topic* topic_;

    DataWriter* writer_;

    TypeSupport type_;

    class PubListener : public DataWriterListener
    {
    public:

        PubListener()
            : matched_(0)
        {
        }

        ~PubListener() override
        {
        }

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

        std::atomic_int matched_;

    } listener_;

public:

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

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

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

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

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

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

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

    delete mypub;
    return 0;
}
```

Examining the code

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

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

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

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

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

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

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

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

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

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

```
class HelloWorldPublisher
```

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

```
private:

    HelloWorld hello_;

    DomainParticipant* participant_;

    Publisher* publisher_;
```

(continues on next page)

(continued from previous page)

```
    Topic* topic_;

    DataWriter* writer_;

    TypeSupport type_;
```

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

```
class PubListener : public DataWriterListener
{
public:

    PubListener()
        : matched_(0)
    {
    }

    ~PubListener() override
    {
    }

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

    std::atomic_int matched_;

} listener_;
```

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

and thus cleans the system memory.

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

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

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

Finally, the HelloWorldPublisher is initialized and run in main.

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

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

    delete mypub;
    return 0;
}
```

CMakeLists.txt

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

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

At this point you can build, compile and run the publisher application. From the build directory in the workspace, run the following commands.

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

Write the Fast DDS subscriber

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

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

Now you have the subscriber's source code. The application runs a subscriber until it receives 10 samples under the topic HelloWorldTopic. At this point the subscriber stops.

```
// Copyright 2016 Proyectos y Sistemas de Mantenimiento SL (eProsima).
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

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

#include "HelloWorldPubSubTypes.h"

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

using namespace eprosima::fastdds::dds;

class HelloWorldSubscriber
{
private:

    DomainParticipant* participant_;

    Subscriber* subscriber_;

    DataReader* reader_;

    Topic* topic_;

    TypeSupport type_;

```

(continues on next page)

(continued from previous page)

```
    class SubListener : public DataReaderListener
    {
    public:

        SubListener()
            : samples_(0)
        {
        }

        ~SubListener() override
        {
        }

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

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

        HelloWorld hello_;

        std::atomic_int samples_;

    } listener_;

public:

```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

        if (subscriber_ == nullptr)
        {
```

(continues on next page)

(continued from previous page)

```
            return false;
        }

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

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

        return true;
    }

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

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

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

    delete mysub;
    return 0;
}
```

Examining the code

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

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

- `Subscriber`. It is the object responsible for the creation and configuration of DataReaders.
- `DataReader`. It is the object responsible for the actual reception of the data. It registers in the application the

topic (TopicDescription) that identifies the data to be read and accesses the data received by the subscriber.

- *DataReaderListener*. This is the listener assigned to the data reader.
- DataReaderQoS. Structure that defines the QoS of the DataReader.
- SampleInfo. It is the information that accompanies each sample that is 'read' or 'taken.'

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

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

```
class HelloWorldSubscriber
```

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

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

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

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

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

    return true;
}
```

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

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

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

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

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

    delete mysub;
    return 0;
}
```

CMakeLists.txt

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

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

At this point you can build, compile and run the subscriber application. From the build directory in the workspace, run the following commands.

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

Putting all together

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

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

Summary

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

Next steps

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

6.5 Library Overview

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

6.5.1 Architecture

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

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

Fig. 3: Fast DDS layer model architecture

DDS Layer

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

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

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

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

- **DataReader**. It is the entity that subscribes to the topics for the reception of publications. The user must provide a subscription Topic when creating this entity. A DataReader receives the messages as changes in its HistoryDataReader.
- **DataReaderHistory**. It contains the *changes* in the data-objects that the DataReader receives as a result of subscribing to a certain Topic.
- **Topic**. Entity that binds Publishers' DataWriters with Subscribers' DataReaders.

RTPS layer

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

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

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

Transport layer

Fast DDS supports the implementation of applications over various transport protocols. Those are UDPv4, UDPv6, TCPv4, TCPv6 and Shared Memory Transport (SHM). By default, a DomainParticipant implements two transport protocols:

- **SHM**: for communications between DomainParticipants in the same machine.
- **UDPv4**: for inter machine communications.

The configuration of all supported transport protocols is detailed in the *Transports* section.

6.5.2 Programming and execution model

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

Concurrency and multithreading

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

- Main thread: Managed by the application.
- Event thread: Each DomainParticipant owns one of these. It processes periodic and triggered time events.

- Asynchronous writer thread: This thread manages asynchronous writes for all DomainParticipants. Even for synchronous writers, some forms of communication must be initiated in the background.
- Reception threads: DomainParticipants spawn a thread for each reception channel, where the concept of a channel depends on the transport layer (e.g. a UDP port).

Event-driven architecture

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

6.5.3 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.
- **Server-Client Discovery**. This discovery mechanism uses a centralized discovery architecture, where servers act as a hubs for discovery meta traffic.
- **Static Discovery**. This implements the discovery of DomainParticipants to each other but it is possible to skip the discovery of the entities contained in each DomainParticipant (DataReader/DataWriter) if these entities are known in advance by the remote DomainParticipants.
- **Manual Discovery**. This mechanism is only compatible with the RTPS layer. It allows the user to manually match and unmatch RTPSParticipants, RTPSWriters, and RTPSReaders using whatever external meta-information channel of its choice.

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

Security

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

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

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

Logging

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

XML profiles configuration

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

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

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

Environment variables

Environment variables are those variables that are defined outside the scope of the program, through operating system functionalities. Fast DDS relies on two environment variables so that the user can easily customize the default settings of DDS applications. These two environment variables are as follows:

- `FASTRTPS_DEFAULT_PROFILES_FILE`. Defines the location of the profile configuration XML files.
- `ROS_DISCOVERY_SERVER`. Sets as the default discovery protocol the Server-Client Discovery. Through this environment variable, the user can provide a list of locators (IP-port pairs) that specify the addresses of the servers. In this case, the server can be a user configured DomainParticipant, or one created by a binary provided by Fast DDS. In this sense, the variable configures the DomainParticipants as clients to the servers in the list.

More information about Fast DDS environment variables can be found in the *Environment variables* section.

6.6 DDS Layer

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

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

- *Subscriber*: It describes the classes used on the subscription side, including *Subscriber* and *DataReader* classes, as well as the *SubscriberListener* and *DataReaderListener* interfaces.
- *Topic*: It describes the classes used to define communication topics and data types, including *Topic* and *TopicDescription* classes, as well as *TypeSupport*, and the *TopicListener* interface.

6.6.1 Core

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

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

- **Domain Participant**: 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.

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 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.

The listener interfaces that each entity type and their callbacks are explained in the documentation for each entity type.

Status

Each entity is associated with a set of status objects whose values represent the *communication status* of that entity. The changes on these status values are the ones that trigger the invocation of the appropriate *Listener* callback to asynchronously inform the application.

See *Status* for a list of all the status objects and a description of their content. There you can also find which status applies to which entity type.

Enabling Entities

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

A disabled entity has its operations limited to the following ones:

- Set/Get the entity *Qos policy*
- Set/Get the entity *Listener*
- Create/Delete subentities
- Get the *Status* of the entity, even if they will not change
- Lookup operations

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

Policy

The Quality of Service (QoS) is used to specify the behavior of the Service, allowing the user to 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.

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.

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. Imagine for example we are publishing vehicle positions, and we want to enforce a position of each vehicle is published periodically. In that case, we can set the ID of the vehicle as the key of the data type, and set the deadline QoS to the desired publication period.

List of QoS Policy data members:

Data Member Name	Type	Default Value
period	fastrtps::Duration_t	c_TimeInfinite

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

It can be changed on enabled entities.

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

Compatibility Rule

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

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

Example

C++

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

XML

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

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

DestinationOrderQosPolicy

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

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*DestinationOrderQosPolicyKind*	BY_RECEPTION_TIMESTAMP

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:

- `BY_RECEPTION_TIMESTAMP`: 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`: 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 < BY_SOURCE_TIMESTAMP
```

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
BY_RECEPTION_TIMESTAMP	BY_RECEPTION_TIMESTAMP	Yes
BY_RECEPTION_TIMESTAMP	BY_SOURCE_TIMESTAMP	No
BY_SOURCE_TIMESTAMP	BY_RECEPTION_TIMESTAMP	Yes
BY_SOURCE_TIMESTAMP	BY_SOURCE_TIMESTAMP	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.

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*DurabilityQosPolicyKind_t*	VOLATILE for DataReaders TRANSIENT LOCAL for DataWriters

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

It cannot be changed on enabled entities.

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

DurabilityQosPolicyKind

There are four possible values:

- `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*).
- `PERSISTENT_DURABILITY_QOS` (*Not Implemented*): All the sample are stored on a permanent storage, so that they can outlive a system session.

Compatibility Rule

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

```
VOLATILE_DURABILITY_QOS < TRANSIENT_LOCAL_DURABILITY_QOS < TRANSIENT_DURABILITY_QOS <␣
↪PERSISTENT_DURABILITY_QOS
```

Table with the possible combinations:

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

Example

C++

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

XML

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

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

DurabilityServiceQosPolicy

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

This QoS Policy is used to configure the *HistoryQosPolicy* and *ResourceLimitsQosPolicy* of the fictitious DataReader and DataWriter used when the *DurabilityQosPolicy* kind is set to TRANSIENT or PERSISTENT.

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 doesn't 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	fastrtps::Duration_t	c_TimeZero
history_kind	*HistoryQosPolicyKind*	KEEP_LAST
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*.
  - There isn't 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.

List of QoS Policy data members:

Data Member Name	Type	Default Value
autoenable_created_entities	bool	true

Note: This QoS Policy concerns to DomainParticipantFactory (as factory for DomainParticipant), DomainParticipant (as factory for Publisher, Subscriber and Topic), Publisher (as factory for DataWriter) and Subscriber (as factory for DataReader).

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

Example

C++

```
EntityFactoryQosPolicy entity_factory;
//The EntityFactoryQosPolicy is default constructed with autoenable_created_entities␣
↪= true
//Change it to false
entity_factory.autoenable_created_entities = false;
```

XML

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

GroupDataQosPolicy

Allows the application to attach additional information to created Publishers or Subscribers. This data is common to all DataWriters\DataReaders belonging to the Publisher\Subscriber and it is propagated by means of the built-in topics.

This QoS Policy can be used in combination with DataWriter and DataReader listeners to implement a matching policy similar to the *PartitionQosPolicy*.

List of QoS Policy data members:

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

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

It can be changed on enabled entities.

Example

C++

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

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

XML

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

HistoryQosPolicy

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*HistoryQosPolicyKind*	KEEP_LAST
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 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:

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

Example

C++

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

XML

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

LatencyBudgetQosPolicy

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

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
duration	fastrtps::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.

By default, the *duration* is infinite, which means that there isn't 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	fastrtps::Duration_t	c_TimeInfinite

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

It can be changed on enabled entities.

Example

C++

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

XML

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

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

LivelinessQosPolicy

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*LivelinessQosPolicyKind*	AUTOMATIC
lease_duration	fastrtps::Duration_t	c_TimeInfinite
announcement_period	fastrtps::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 or MANUAL_BY_PARTICIPANT 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:

- `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 < MANUAL_BY_PARTICIPANT_LIVELINESS_QOS < MANUAL_BY_TOPIC_
↪LIVELINESS_QOS
```

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
AUTOMATIC_LIVELINESS_QOS	AUTOMATIC_LIVELINESS_QOS	Yes
AUTOMATIC_LIVELINESS_QOS	MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	No
AUTOMATIC_LIVELINESS_QOS	MANUAL_BY_TOPIC_LIVELINESS_QOS	No
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	AUTOMATIC_LIVELINESS_QOS	Yes
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	Yes
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	MANUAL_BY_TOPIC_LIVELINESS_QOS	No
MANUAL_BY_TOPIC_LIVELINESS_QOS	AUTOMATIC_LIVELINESS_QOS	Yes
MANUAL_BY_TOPIC_LIVELINESS_QOS	MANUAL_BY_PARTICIPANT_LIVELINESS_QOS	Yes
MANUAL_BY_TOPIC_LIVELINESS_QOS	MANUAL_BY_TOPIC_LIVELINESS_QOS	Yes

Additionally, the *lease_duration* of the DataWriter must also be greater than the *lease_duration* of the DataReader.

Example

C++

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

XML

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

(continues on next page)

(continued from previous page)

```
        </lease_duration>
        <kind>AUTOMATIC</kind>
      </liveliness>
    </qos>
</publisher>

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

OwnershipQosPolicy

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*OwnershipQosPolicyKind*	SHARED

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:

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

Compatibility Rule

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

Table with the possible combinations:

DataWriter kind	DataReader kind	Compatibility
SHARED_OWNERSHIP_QOS	SHARED_OWNERSHIP_QOS	Yes
SHARED_OWNERSHIP_QOS	EXCLUSIVE_OWNERSHIP_QOS	No
EXCLUSIVE_OWNERSHIP_QOS	SHARED_OWNERSHIP_QOS	No
EXCLUSIVE_OWNERSHIP_QOS	EXCLUSIVE_OWNERSHIP_QOS	Yes

Example

C++

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

XML

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

OwnershipStrengthQosPolicy

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
value	uint32_t	0

Note: This QoS Policy concerns to DataWriter entities.

It can be changed on enabled entities.

Example

C++

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

XML

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

PartitionQosPolicy

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

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)
partitions	fastrtps::rtps::Serialized_Payload_t	Empty List

- **Max Size**: Maximum size for the list of partition names.
- **Partitions**: List of partition names.

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

It can be changed on enabled entities.

Example

C++

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

(continues on next page)

(continued from previous page)

```
part.push_back("part2");
partitions.names(part); //Setter function
```

XML

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

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

PresentationQosPolicy

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

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
access_scope	*PresentationQosPolicyAccessScopeKind*	INSTANCE
coherent_access	bool	false
ordered_access	bool	false

- **Access Scope**: Determines the largest scope spanning the entities for which the order and coherency can be preserved. See *PresentationQosPolicyAccessScopeKind* for further details.
- **Coherent Access**: Controls whether the service will preserve grouping of changes made on the publishing side, such that they are received as a unit on the subscribing side.
- **Ordered Access**: Controls whether the service supports the ability of the subscriber to see changes in the same order as they occurred on the publishing side.

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

It cannot be changed on enabled entities.

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

PresentationQosPolicyAccessScopeKind

There are three possible values, which have different behaviors depending on the values of coherent_access and ordered_access variables:

- `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 < TOPIC_PRESENTATION_QOS < GROUP_PRESENTATION_QOS
```

Table with the possible combinations:

Publisher scope	Subscriber scope	Compatibility
INSTANCE_PRESENTATION_QOS	INSTANCE_PRESENTATION_QOS	Yes
INSTANCE_PRESENTATION_QOS	TOPIC_PRESENTATION_QOS	No
INSTANCE_PRESENTATION_QOS	GROUP_PRESENTATION_QOS	No
TOPIC_PRESENTATION_QOS	INSTANCE_PRESENTATION_QOS	Yes
TOPIC_PRESENTATION_QOS	TOPIC_PRESENTATION_QOS	Yes
TOPIC_PRESENTATION_QOS	GROUP_PRESENTATION_QOS	No
GROUP_PRESENTATION_QOS	INSTANCE_PRESENTATION_QOS	Yes
GROUP_PRESENTATION_QOS	TOPIC_PRESENTATION_QOS	Yes
GROUP_PRESENTATION_QOS	GROUP_PRESENTATION_QOS	Yes

Additionally, the coherent_access and ordered_access of the Subscriber can only be enabled if they are also enabled on the Publisher.

ReaderDataLifecycleQosPolicy

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

This QoS Policy specifies the behavior of the DataReader with respect to the lifecycle of the data instances it manages , that is, the instances that have been received and for which the DataReader maintains some internal resources. The DataReader maintains the samples that have not been taken by the application, subject to the constraints imposed by *HistoryQosPolicy* and *ResourceLimitsQosPolicy*.

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 doesn't take those samples as the service will prevent the DataReader from reclaiming the resources and they will remain in the DataReader indefinitely. This QoS exist to avoid that situation.

List of QoS Policy data members:

Data Member Name	Type	Default Value
autopurge_no_writer_samples_delay	fastrtps::Duration_t	c_TimeInfinite
autopurge_disposed_samples_delay	fastrtps::Duration_t	c_TimeInfinite

- **Autopurge no writers samples delay**: Defines the maximum duration the DataReader must retain the information regarding an instance once its *instance_state* becomes *NOT_ALIVE_NO_WRITERS*. 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*. 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.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*ReliabilityQosPolicyKind*	BEST_EFFORT for DataReaders RELIABLE for DataWriters
max_blocking_time	fastrtps::Duration_t	100 ms

- **Kind**: Specifies the behavior of the service regarding delivery of the samples. See *ReliabilityQosPolicyKind* for further details.
- **Max blocking time**: Configures the maximum duration that the write operation can be blocked.

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

It cannot be changed on enabled entities.

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

ReliabilityQosPolicyKind

There are two possible values:

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

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

Compatibility Rule

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

```
BEST_EFFORT_RELIABILITY_QOS < RELIABLE_RELIABILITY_QOS
```

Table with the possible combinations:

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

Example

C++

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

XML

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

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

ResourceLimitsQosPolicy

This QoS Policy controls the resources that the service can use in order to meet the requirements imposed by the application and other QoS Policies.

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

- **Max samples**: Controls the maximum number of samples that the DataWriter or DataReader can manage across all the instances associated with it. In other words, it represents the maximum samples that the middleware can store for a DataReader or DataWriter.
- **Max instances**: Controls the maximum number of instances that a DataWriter or DataReader can manage.
- **Max samples per instance**: Controls the maximum number of samples within an instance that the DataWriter or DataReader can manage.
- **Allocated samples**: States the number of samples that will be allocated on initialization.

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;
```

(continues on next page)

(continued from previous page)

```
//The ResourceLimitsQosPolicy is default constructed with allocated_samples = 100
//Change allocated_samples to 50
resource_limits.allocated_samples = 50;
```

XML

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

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

TimeBasedFilterQosPolicy

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

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

The *minimum_separation* must be lower than the *DeadlineQosPolicy period*. By default, the *minimum_separation* is zero, which means that the DataReader is potentially interested in all the values.

List of QoS Policy data members:

Data Member Name	Type	Default Value
minimum_separation	fastrtps::Duration_t	c_TimeZero

Note: This QoS Policy concerns to DataReader entities.

It can be changed on enabled entities.

TopicDataQosPolicy

Allows the application to attach additional information to a created Topic so that when it is discovered by a remote application, it can access the data and use it.

List of QoS Policy data members:

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

Note: This QoS Policy concerns to Topic entities.

It can be changed even if it is already created.

Example

C++

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

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

XML

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

TransportPriorityQosPolicy

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

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

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.

List of QoS Policy data members:

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

Note: This QoS Policy concerns to all DDS entities.

It can be changed on enabled entities.

Example

C++

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

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

XML

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

WriterDataLifecycleQosPolicy

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

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

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

List of QoS Policy data members:

Data Member Name	Type	Default Value
autodispose_unregistered_instances	bool	true

Note: This QoS Policy concerns to DataWriter entities.

It can be changed on enabled entities.

eProsima Extensions

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

- *DisablePositiveACKsQosPolicy*
- *ParticipantResourceLimitsQos*
- *PropertyPolicyQos*
- *PublishModeQosPolicy*
- *ReaderResourceLimitsQos*
- *RTPSEndpointQos*
- *RTPSReliableReaderQos*
- *RTPSReliableWriterQos*
- *TransportConfigQos*
- *TypeConsistencyQos*
- *WireProtocolConfigQos*
- *WriterResourceLimitsQos*

DisablePositiveACKsQosPolicy

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

List of QoS Policy data members:

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

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

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

It cannot be changed on enabled entities.

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

Compatibility Rule

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

Table with the possible combinations:

DataWriter *enabled* value	DataReader *enabled* value	Compatibility
true	true	Yes
true	false	Yes
false	true	No
false	false	Yes

Example

C++

```
DisablePositiveACKsQosPolicy disable_acks;
//The DisablePositiveACKsQosPolicy is default constructed with enabled = false
//Change enabled to true
disable_acks.enabled = true;
//The DisablePositiveACKsQosPolicy is default constructed with infinite duration
//Change the duration to 1 second
disable_acks.duration = {1, 0};
```

XML

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

<subscriber profile_name="subscriber_xml_conf_disable_positive_acks_profile">
    <qos>
```

(continues on next page)

(continued from previous page)

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

ParticipantResourceLimitsQos

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

List of QoS Policy data members:

Data Member Name	Type
locators	*RemoteLocatorsAllocationAttributes*
participants	*ResourceLimitedContainerConfig*
readers	*ResourceLimitedContainerConfig*
writers	*ResourceLimitedContainerConfig*
send_buffers	*SendBuffersAllocationAttributes*
data_limits	*VariableLengthDataLimits*

- **Locators**: Defines the limits for collections of remote locators.
- **Participants**: Specifies the allocation behavior and limits for collections dependent on the total number of participants.
- **Readers**: Specifies the allocation behavior and limits for collections dependent on the total number of readers per participant.
- **Writers**: Specifies the allocation behavior and limits for collections dependent on the total number of writers per participant.
- **Send buffers**: Defines the allocation behavior and limits for the send buffer manager.
- **Data Limits**: States the limits for variable-length data.

Note: This QoS Policy concerns to DomainParticipant entities.

It cannot be changed on enabled entities.

RemoteLocatorsAllocationAttributes

This structure holds the limits for the remote locators' collections.

List of structure members:

Member Name	Type	Default Value
max_unicast_locators	size_t	4
max_multicast_locators	size_t	1

- **Max unicast locators**: This member controls the maximum number of unicast locators to keep for each discovered remote entity. It is recommended to use the highest number of local addresses found on all the systems belonging to the same domain.

- **Max multicast locators**: This member controls the maximum number of multicast locators to keep for each discovered remote entity. The default value is usually enough, as it doesn't make sense to add more than one multicast locator per entity.

ResourceLimitedContainerConfig

This structure holds the limits of a resource limited collection, as well as the allocation configuration, which can be fixed size or dynamic size.

List of structure members:

Member Name	Type	Default Value
initial	size_t	0
maximum	size_t	std::numeric_limits<size_t>::max()
increment	size_t	1 (dynamic size), 0 (fixed size)

- **Initial**: Indicates the number of elements to preallocate in the collection.
- **Maximum**: Specifies the maximum number of elements allowed in the collection.
- **Increment**: States the number of items to add when the reserved capacity limit is reached. This member has a different default value depending on the allocation configuration chosen.

SendBuffersAllocationAttributes

This structure holds the limits for the allocations of the send buffers.

List of structure members:

Member Name	Type	Default Value
preallocated_number	size_t	0
dynamic	bool	false

- **Preallocated number**: This member controls the initial number of send buffers to be allocated. The default value will perform an initial guess of the number of buffers required, based on the number of threads from which a send operation could be started.
- **Dynamic**: This member controls how the buffer manager behaves when a send buffer is not available. When true, a new buffer will be created. Otherwise, it will wait for a buffer to be returned.

VariableLengthDataLimits

This structure holds the limits for variable-length data.

List of structure members:

Member Name	Type	Default Value
max_properties	size_t	0
max_user_data	size_t	0
max_partitions	size_t	0

- **Max properties**: Defines the maximum size, in octets, of the properties data in the local or remote participant.
- **Max user data**: Establishes the maximum size, in octets, of the user data in the local or remote participant.

- **Max partitions**: States the maximum size, in octets, of the partitions data in the local or remote participant.

Example

C++

```
ParticipantResourceLimitsQos participant_limits;
//Set the maximum size of participant resource limits collection to 3 and it␣
↪allocation configuration to fixed size
participant_limits.participants =␣
↪eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(3u);
//Set the maximum size of reader's resource limits collection to 2 and its allocation␣
↪configuration to fixed size
participant_limits.readers =␣
↪eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(2u);
//Set the maximum size of writer's resource limits collection to 1 and its allocation␣
↪configuration to fixed size
participant_limits.writers =␣
↪eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(1u);
//Set the maximum size of the partition data to 256
participant_limits.data_limits.max_partitions = 256u;
//Set the maximum size of the user data to 256
participant_limits.data_limits.max_user_data = 256u;
//Set the maximum size of the properties data to 512
participant_limits.data_limits.max_properties = 512u;
```

XML

```
<participant profile_name="participant_alloc_qos_example">
    <rtps>
        <allocation>
            <!-- We know we have 3 participants on the domain -->
            <total_participants>
                <initial>3</initial>
                <maximum>3</maximum>
                <increment>0</increment>
            </total_participants>
            <!-- We know we have at most 2 readers on each participant -->
            <total_readers>
                <initial>2</initial>
                <maximum>2</maximum>
                <increment>0</increment>
            </total_readers>
            <!-- We know we have at most 1 writer on each participant -->
            <total_writers>
                <initial>1</initial>
                <maximum>1</maximum>
                <increment>0</increment>
            </total_writers>
            <max_partitions>256</max_partitions>
            <max_user_data>256</max_user_data>
            <max_properties>512</max_properties>
        </allocation>
```

(continues on next page)

(continued from previous page)

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

PropertyPolicyQos

This additional QoS Policy stores name/value pairs that can be used to configure certain DDS settings that cannot be configured directly using an standard QoS Policy. In Fast DDS, it can be used to configure the security settings (See *Security* for further details of the security functionality).

Example

C++

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

XML

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

                <!-- Activate Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.plugin</name>
                    <value>builtin.Access-Permissions</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

PublishModeQosPolicy

This QoS Policy configures how the DataWriter sends the data.

List of QoS Policy data members:

Data Member Name	Type	Default Value
kind	*PublishModeQosPolicyKind*	SYNCHRONOUS

Note: This QoS Policy concerns to DataWriter entities.

It cannot be changed on enabled entities.

PublishModeQosPolicyKind

There are two possible values:

- `SYNCHRONOUS_PUBLISH_MODE`: The data is sent in the context of the user thread that calls the write operation.
- `ASYNCHRONOUS_PUBLISH_MODE`: An internal thread takes the responsibility of sending the data asynchronously. The write operation returns before the data is actually sent.

Example

C++

```
PublishModeQosPolicy publish_mode;
//The PublishModeQosPolicy is default constructed with kind = SYNCHRONOUS
//Change the kind to ASYNCHRONOUS
publish_mode.kind = ASYNCHRONOUS_PUBLISH_MODE;
```

XML

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

ReaderResourceLimitsQos

This QoS Policy states the limits for the matched DataWriters' resource limited collections based on the maximum number of DataWriters that are going to match with the DataReader.

List of QoS Policy data members:

Data Member Name	Type
matched_publisher_allocation	*ResourceLimitedContainerConfig*

Note: This QoS Policy concerns to DataReader entities.

It cannot be changed on enabled entities.

Example

C++

```
ReaderResourceLimitsQos reader_limits;
//Set the maximum size for writer matched resource limits collection to 1 and its␣
↪allocation configuration to fixed size
reader_limits.matched_publisher_allocation =␣
↪eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(1u);
```

XML

```
<subscriber profile_name="alloc_qos_example_sub">
    <!-- we know we will only have one matching publisher -->
    <matchedPublishersAllocation>
        <initial>1</initial>
        <maximum>1</maximum>
        <increment>0</increment>
    </matchedPublishersAllocation>
</subscriber>
```

RTPSEndpointQos

This QoS Policy configures the aspects of an RTPS endpoint, such as the list of locators, the identifiers, and the history memory policy.

List of QoS Policy data members:

Data Member Name	Type	Default Value
unicast_locator_list	fastrtps::rtps::LocatorList_t	Empty List
multicast_locator_list	fastrtps::rtps::LocatorList_t	Empty List
remote_locator_list	fastrtps::rtps::LocatorList_t	Empty List
user_defined_id	int16_t	-1
entity_id	int16_t	-1
history_memory_policy	*MemoryManagementPolicy*	PREALLOCATED

- **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 don't use any multicast locator, but it can be changed by means of this QoS.
- **Remote locator list**: States the list of remote locators associated to the DDS Entity.
- **User defined ID**: Establishes the unique identifier used for StaticEndpointDiscovery.
- **Entity ID**: The user can specify the identifier for the endpoint.
- **History memory policy**: Indicates the way the memory is managed in terms of dealing with the CacheChanges.

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

It cannot be changed on enabled entities.

MemoryManagementPolicy

There are four possible values:

- `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);
//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
```

(continues on next page)

(continued from previous page)

```
//Change the history_memory_policy to DYNAMIC_RESERVE
endpoint.history_memory_policy = eprosima::fastrtps::rtps::DYNAMIC_RESERVE_MEMORY_
↪MODE;
```

XML

```
<publisher profile_name="publisher_xml_conf_unicast_locators_profile">
    <userDefinedID>3</userDefinedID>
    <entityID>2</entityID> <!-- Int16 -->
    <unicastLocatorList>
        <locator>
            <udpv4>
                <port>7800</port>
            </udpv4>
        </locator>
    </unicastLocatorList>
    <multicastLocatorList>
        <locator>
            <udpv4>
                <address>239.255.0.4</address>
                <port>7900</port>
            </udpv4>
        </locator>
    </multicastLocatorList>
    <!-- The history memory policy is changed to DYNAMIC_RESERVE -->
    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
</publisher>

<subscriber profile_name="subscriber_xml_conf_unicast_locators_profile">
    <userDefinedID>5</userDefinedID>
    <entityID>4</entityID> <!-- Int16 -->
    <unicastLocatorList>
        <locator>
            <udpv4>
                <port>7800</port>
            </udpv4>
        </locator>
    </unicastLocatorList>
    <multicastLocatorList>
        <locator>
            <udpv4>
                <address>239.255.0.4</address>
                <port>7900</port>
            </udpv4>
        </locator>
    </multicastLocatorList>
    <historyMemoryPolicy>PREALLOCATED_WITH_REALLOC</historyMemoryPolicy>
</subscriber>
```

RTPSReliableReaderQos

This RTPS QoS Policy allows the configuration of several RTPS reliable reader's aspects.

List of QoS Policy data members:

Data Member Name	Type
times	*ReaderTimes*
disable_positive_ACKs	*DisablePositiveACKsQosPolicy*

- **Times**: Defines the duration of the RTPSReader events. See *ReaderTimes* for further details.
- **Disable positive ACKs**: Configures the settings to disable the positive acks. See *DisablePositiveACKsQosPolicy* for further details.

Note: This QoS Policy concerns to DataReader entities.

It cannot be changed on enabled entities.

ReaderTimes

This structure defines the times associated with the Reliable Readers' events.

List of structure members:

Member Name	Type	Default Value
initialAcknackDelay	fastrtps::Duration_t	70 milliseconds
heartbeatResponseDelay	fastrtps::Duration_t	5 milliseconds

- **Initial acknack delay**: Defines the duration of the initial acknack delay.
- **Heartbeat response delay**: 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

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

RTPSReliableWriterQos

This RTPS QoS Policy allows the configuration of several RTPS reliable writer's aspects.

List of QoS Policy data members:

Data Member Name	Type
times	*WriterTimes*
disable_positive_acks	*DisablePositiveACKsQosPolicy*

- **Times**: Defines the duration of the RTPSWriter events. See *WriterTimes* for further details.
- **Disable positive ACKs**: Configures the settings to disable the positive acks. See *DisablePositiveACKsQosPolicy* for further details.

Note: This QoS Policy concerns to DataWriter entities.

It cannot be changed on enabled entities.

WriterTimes

This structure defines the times associated with the Reliable Writers' events.

List of structure members:

Member Name	Type	Default Value
initialHeartbeatDelay	fastrtps::Duration_t	12 milliseconds
heartbeatPeriod	fastrtps::Duration_t	3 seconds
nackResponseDelay	fastrtps::Duration_t	5 milliseconds
nackSupressionDuration	fastrtps::Duration_t	0 seconds

- **Initial heartbeat delay**: Defines duration of the initial heartbeat delay.
- **Heartbeat period**: Specifies the interval between periodic heartbeats.
- **Nack response delay**: Establishes the duration of the delay applied to the response of an ACKNACK message.

- **Nack supression duration**: The RTPSWriter ignores the nack messages received after sending the data until the duration time elapses.

Example

C++

```
RTPSReliableWriterQos reliable_writer_qos;
//The RTPSReliableWriterQos is default constructed with initialHeartbeatDelay = 12 ms
//Change the initialHeartbeatDelay to 20 nanoseconds
reliable_writer_qos.times.initialHeartbeatDelay = {0, 20};
//The RTPSReliableWriterQos is default constructed with heartbeatPeriod = 3 s
//Change the heartbeatPeriod to 5 seconds
reliable_writer_qos.times.heartbeatPeriod = {5, 0};
//The RTPSReliableWriterQos is default constructed with nackResponseDelay = 5 ms
//Change the nackResponseDelay to 10 nanoseconds
reliable_writer_qos.times.nackResponseDelay = {0, 10};
//The RTPSReliableWriterQos is default constructed with nackSupressionDuration = 0 s
//Change the nackSupressionDuration to 20 nanoseconds
reliable_writer_qos.times.nackSupressionDuration = {0, 20};
//You can also change the DisablePositiveACKsQosPolicy. For further details see␣
↪DisablePositiveACKsQosPolicy section.
reliable_writer_qos.disable_positive_acks.enabled = true;
```

XML

```
<publisher profile_name="pub_profile_name">
    <times> <!-- writerTimesType -->
        <initialHeartbeatDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>20</nanosec>
        </initialHeartbeatDelay>
        <heartbeatPeriod> <!-- DURATION -->
            <sec>5</sec>
            <nanosec>0</nanosec>
        </heartbeatPeriod>
        <nackResponseDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>10</nanosec>
        </nackResponseDelay>
        <nackSupressionDuration> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>20</nanosec>
        </nackSupressionDuration>
    </times>
    <!--You can also change the values of DisablePositiveACKsQosPolicy.-->
    <!--See DisablePositiveACKsQosPolicy section for further details-->
</publisher>
```

TransportConfigQos

This QoS Policy allows the configuration of the transport layer settings.

List of QoS Policy data members:

Data Member Name	Type	Default Value
user_transports	std::vector<std::shared_ptr<*TransportDescriptorInterface*>>	Empty Vector
use_builtin_transports	bool	true
send_socket_buffer_size	uint32_t	0
listen_socket_buffer_size	uint32_t	0

- **User transports**: This data member defines the list of transports to use alongside or in place of builtins.
- **Use builtin transports**: It controls whether the built-in transport layer is enabled or disabled. If it is set to false, the default UDPv4 implementation is disabled.
- **Send socket buffer size**: By default, Fast DDS creates socket buffers using the system default size. This data member allows to change the send socket buffer size used to send data.
- **Listen socket buffer size**: The listen socket buffer size is also created with the system default size, but it can be changed using this data member.

Note: This QoS Policy concerns to DomainParticipant entities.

It cannot be changed on enabled entities.

TransportDescriptorInterface

This structure is the base for the data type used to define transport configuration.

List of structure members:

Member Name	Type
maxMessageSize	uint32_t
maxInitialPeersRange	uint32_t

- **Max message size**: This member sets the maximum size in bytes of the transport's message buffer.
- **Max initial peers range**: 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);
```

(continues on next page)

(continued from previous page)

```
//Set use_builtin_transports to false
transport.use_builtin_transports = false;
```

XML

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>my_transport</transport_id>
        <type>UDPv4</type>
        <sendBufferSize>9216</sendBufferSize>
        <receiveBufferSize>9216</receiveBufferSize>
    </transport_descriptor>
</transport_descriptors>

<participant profile_name="my_transport">
    <rtps>
        <userTransports>
            <transport_id>my_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
    </rtps>
</participant>
```

TypeConsistencyQos

This QoS Policy allows the configuration of the XTypes extension QoS on the DataReader.

List of QoS Policy data members:

Data Member Name	Type
type_consistency	*TypeConsistencyEnforcementQosPolicy*
representation	*DataRepresentationQosPolicy*

- **Type consistency**: It states the rules for the data types compatibility. See *TypeConsistencyEnforcementQosPolicy* for further details.
- **Representation**: It specifies the data representations valid for the entities. See *DataRepresentationQosPolicy* for further details.

Note: This QoS Policy concerns to DataReader entities.

It cannot be changed on enabled entities.

Example

C++

```
TypeConsistencyQos consistency_qos;
//You can change the DataRepresentationQosPolicy. For further details see␣
↪DataRepresentationQosPolicySection section.
consistency_qos.representation.m_value.push_back(DataRepresentationId_t::XCDR2_DATA_
↪REPRESENTATION);
//You can change the TypeConsistencyEnforcementQosPolicy. For further details see␣
↪TypeConsistencyEnforcementQosPolicy section.
consistency_qos.type_consistency.m_kind = TypeConsistencyKind::ALLOW_TYPE_COERCION;
```

XML

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

WireProtocolConfigQos

This QoS Policy allows the configuration of the wire protocol.

List of QoS Policy data members:

Data Member Name	Type	Default Value
prefix	fastrtps::rtps::GuidPrefix_t	0
participant_id	int32_t	-1
builtin	*RTPS BuiltinAttributes*	
throughput_controller	*ThroughputControllerDescription*	
default_unicast_locator_list	fastrtps::rtps::LocatorList_t	Empty List
default_multicast_locator_list	fastrtps::rtps::LocatorList_t	Empty List

- **Prefix**: This data member allows the user to set manually the GUID prefix.
- **Participant ID**: It sets the participant identifier. By default, it will be automatically generated by the Domain.
- **Builtin**: This data member allows the configuration of the built-in parameters. See *RTPS BuiltinAttributes* for further details.
- **Throughput controller**: It allows the configuration of the throughput settings.
- **Default unicast locator list**: States the default list of unicast locators to be used for any endpoint defined inside the RTPSParticipant in the case that it was defined without unicast locators. This list should include at least one locator.
- **Default multicast locator list**: Stores the default list of multicast locators to be used for any endpoint defined inside the RTPSParticipant in the case that it was defined without multicast locators. This list is usually left empty.

Note: This QoS Policy concerns to DomainParticipant entities.

It cannot be changed on enabled entities.

ThroughputControllerDescription

This structure allows to limit the output bandwidth.

List of structure members:

Member Name	Type
bytesPerPeriod	uint32_t
periodMillisecs	uint32_t

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

Example

C++

```
eprosima::fastdds::dds::WireProtocolConfigQos wire_protocol;
//Set the guid prefix
std::istringstream("72.61.73.70.66.61.72.6d.74.65.73.74") >> wire_protocol.prefix;
//Configure Builtin Attributes
wire_protocol.builtin.discovery_config.discoveryProtocol =␣
↪eprosima::fastrtps::rtps::DiscoveryProtocol_t::SERVER;
//Add locator to unicast list
eprosima::fastrtps::rtps::Locator_t server_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(server_locator, "192.168.10.57");
server_locator.port = 56542;
wire_protocol.builtin.metatrafficUnicastLocatorList.push_back(server_locator);
// Limit to 300kb per second.
eprosima::fastrtps::rtps::ThroughputControllerDescriptor␣
↪slowPublisherThroughputController{300000, 1000};
wire_protocol.throughput_controller = slowPublisherThroughputController;
//Add locator to default unicast locator list
eprosima::fastrtps::rtps::Locator_t unicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(unicast_locator, 192, 168, 1, 41);
unicast_locator.port = 7400;
wire_protocol.default_unicast_locator_list.push_back(unicast_locator);
//Add locator to default multicast locator list
eprosima::fastrtps::rtps::Locator_t multicast_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(multicast_locator, 192, 168, 1, 41);
multicast_locator.port = 7400;
wire_protocol.default_multicast_locator_list.push_back(multicast_locator);
```

XML

```
<participant profile_name="UDP SERVER" is_default_profile="true">
    <rtps>
        <prefix>72.61.73.70.66.61.72.6d.74.65.73.74</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
            </discovery_config>
```

(continues on next page)

(continued from previous page)

```
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.10.57</address>
                        <port>56542</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
        <throughputController>
            <bytesPerPeriod>300000</bytesPerPeriod>
            <periodMillisecs>1000</periodMillisecs>
        </throughputController>
        <defaultUnicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
        </defaultUnicastLocatorList>

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

WriterResourceLimitsQos

This QoS Policy states the limits for the matched DataReaders' resource limited collections based on the maximum number of DataReaders that are going to match with the DataWriter.

List of QoS Policy data members:

Data Member Name	Type
matched_subscriber_allocation	*ResourceLimitedContainerConfig*

Note: This QoS Policy concerns to DataWriter entities.

It cannot be changed on enabled entities.

Example

C++

```
WriterResourceLimitsQos writer_limits;
//Set the maximum size for reader matched resource limits collection to 3 and its␣
↪allocation configuration to fixed size
writer_limits.matched_subscriber_allocation =␣
↪eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_configuration(3u);
```

XML

```
<publisher profile_name="alloc_qos_example_pub_for_topic_1">
    <!-- we know we will have three matching subscribers -->
    <matchedSubscribersAllocation>
        <initial>3</initial>
        <maximum>3</maximum>
        <increment>0</increment>
    </matchedSubscribersAllocation>
</publisher>
```

XTypes Extensions

This section explain those QoS Policy extensions defined in the XTypes Specification:

- *DataRepresentationQosPolicy*
- *TypeConsistencyEnforcementQosPolicy*

DataRepresentationQosPolicy

This XTypes QoS Policy states which data representations will be used by the DataWriters and DataReaders.

The DataWriters offer a single data representation that will be used to communicate with the matched DataReaders. The DataReaders can request one or more data representations and in order to have communication with the DataWriter, the offered data representation needs to be contained within the DataReader request.

List of QoS Policy data members:

Data Member Name	Type	Default Value
m_value	std::vector<*DataRepresentationId*>	Empty vector

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

It cannot be changed on enabled entities.

DataRepresentationId

There are three possible values:

- `XCDR_DATA_REPRESENTATION`: This option corresponds to the first version of the *Extended CDR Representation* encoding.
- `XML_DATA_REPRESENTATION`: This option corresponds to the *XML Data Representation*.
- `XCDR2_DATA_REPRESENTATION`: This option corresponds to the second version of the *Extended CDR Representation* encoding.

Example

C++

```
DataRepresentationQosPolicy data_representation;
//Add XCDR v1 data representation to the list of valid representations
data_representation.m_value.push_back(DataRepresentationId_t::XCDR_DATA_
↪REPRESENTATION);
//Add XML data representation to the list of valid representations
data_representation.m_value.push_back(DataRepresentationId_t::XML_DATA_
↪REPRESENTATION);
```

XML

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

TypeConsistencyEnforcementQosPolicy

This XTypes QoS Policy extension defines the rules for determining whether the data type used in the DataWriter is consistent with the one used in the DataReader.

List of QoS Policy data members:

Data Member Name	Type	Default Value
m_kind	*TypeConsistencyKind*	ALLOW_TYPE_COERCION
m_ignore_sequence_bounds	bool	true
m_ignore_string_bounds	bool	true
m_ignore_member_names	bool	false
m_prevent_type_widening	bool	false
m_force_type_validation	bool	false

- **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.
- **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.

- **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.
- **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.
- **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.
- **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 don't need to support the same data type in order to communicate as long as the DataReader's type is assignable from the DataWriter's type.

Example

C++

```
TypeConsistencyEnforcementQosPolicy type_enforcement;
//The TypeConsistencyEnforcementQosPolicy is default constructed with kind = ALLOW_
↪TYPE_COERCION
//Change the kind to DISALLOW_TYPE_COERCION
type_enforcement.m_kind = TypeConsistencyKind::DISALLOW_TYPE_COERCION;
//Configures the system to ignore the sequence sizes in assignations
type_enforcement.m_ignore_sequence_bounds = true;
//Configures the system to ignore the string sizes in assignations
type_enforcement.m_ignore_string_bounds = true;
//Configures the system to ignore the member names. Members with same ID could have␣
↪different names
type_enforcement.m_ignore_member_names = true;
//Configures the system to allow type widening
type_enforcement.m_prevent_type_widening = false;
//Configures the system to not use the complete Type Information in entities match␣
↪process
type_enforcement.m_force_type_validation = false;
```

XML

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

Status

Each *Entity* is associated with a set of `Status` objects whose values represent the *communication status* of that *Entity*. Changes on the status values occur due to communication events related to each of the entities, e.g., when new data arrives, a new participant is discovered, or a remote endpoint is lost. The status is decomposed into several status objects, each concerning a different aspect of the communication, so that each of these status objects can vary independently of the others.

Changes on a status object trigger the corresponding *Listener* callbacks that allow the *Entity* to inform the application about the event. For a given status object with name `fooStatus`, the entity listener interface defines a callback function `on_foo()` that will be called when the status changes. The only exceptions are *DataOnReaders* and *DataAvailable*. Beware that some statuses have data members that are reset every time the corresponding listener is called. The only exception to this rule is when the entity has no listener attached, so the callback cannot be called. See the documentation of each status for details.

The entities expose functions to access the value of its statuses. For a given status with name `fooStatus`, the entity exposes a member function `get_foo()` to access the data in its `fooStatus`. 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 can be used as a quick reference too.

Status Name	Entity	Listener callback
InconsistentTopicStatus	*Topic*	`on_inconsistent_topic`
DataOnReaders	*Subscriber*	`on_data_on_readers`
DataAvailable	*DataReader*	`on_data_available`
LivelinessChangedStatus	*DataReader*	`on_liveliness_changed`
RequestedDeadlineMissedStatus	*DataReader*	`on_requested_deadline_missed`
RequestedIncompatibleQosStatus	*DataReader*	`on_requested_incompatible_qos`
SampleLostStatus	*DataReader*	`on_sample_lost`
SampleRejectedStatus	*DataReader*	`on_sample_rejected`
SubscriptionMatchedStatus	*DataReader*	`on_suscription_matched`
LivelinessLostStatus	*DataWriter*	`on_liveliness_lost`
OfferedDeadlineMissedStatus	*DataWriter*	`on_offered_deadline_missed`
OfferedIncompatibleQosStatus	*DataWriter*	`on_offered_incompatible_qos`
PublicationMatchedStatus	*DataWriter*	`on_publication_matched`

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*.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`

- **total_count**: Total cumulative count of inconsistent *Topics* discovered since the creation of the current *Topic*.
- **total_count_change**: The change in **total_count** since the last time `on_inconsistent_topic` was called or the status was read.

> **Warning:** Currently this status is not supported and will be implemented in future releases. As a result, trying to access this status will return `NOT_SUPPORTED` and the corresponding listener will never be called.

DataOnReaders

This status becomes active every time there is new data available for the application on any *DataReader* belonging to the current *Subscriber*. There is no getter function to access this status, as it does not keep track of any information related to the data itself. Its only purpose is to trigger the `on_data_on_readers` callback on the listener attached to the *DataReader*.

DataAvailable

This status becomes active every time there is new data available for the application on the *DataReader*. There is no getter function to access this status, as it does not keep track of any information related to the data itself. Its only purpose is to trigger the `on_data_available` callback on the listener attached to the *DataReader*.

LivelinessChangedStatus

This status changes every time the liveliness status of a matched *DataWriter* has changed. Either because a *DataWriter* that was *inactive* has become *active* or the other way around.

List of status data members:

Data Member Name	Type
alive_count	`int32_t`
not_alive_count	`int32_t`
alive_count_change	`int32_t`
not_alive_count_change	`int32_t`
last_publication_handle	`InstanceHandle_t`

- **alive_count**: Total number of currently active *DataWriters*. This count increases every time a newly matched *DataWriter* asserts its liveliness or a *DataWriter* that was considered not alive reasserts its liveliness. It decreases every time an active *DataWriter* becomes not alive, either because it failed to asserts its liveliness or because it was deleted for any reason.
- **not_alive_count**: Total number of matched *DataWriters* that are currently considered not alive. This count increases every time an active *DataWriter* becomes not alive because it fails to assert its liveliness. It decreases every time a *DataWriter* that was considered not alive reasserts its liveliness. Normal matching and unmatching of *DataWriters* does not affect this count.
- **alive_count_change**: The change in **alive_count** since the last time `on_liveliness_changed` was called or the status was read. It can have positive or negative values.
- **not_alive_count_change**: The change in **not_alive_count** since the last time `on_liveliness_changed` was called or the status was read. It can have positive or negative values.

- **last_publication_handle**: Handle to the last *DataWriter* whose liveliness status was changed. If no liveliness has ever changed, it will have value `c_InstanceHandle_Unknown`.

RequestedDeadlineMissedStatus

This status changes every time the *DataReader* does not receive data within the deadline period configured on its *DataReaderQos*.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`
last_instance_handle	`InstanceHandle_t`

- **total_count**: Total cumulative count of missed deadlines for any instance read by the current *DataReader*. As the deadline period applies to each instance of the *Topic* independently, the count will will be incremented by one for each instance for which data was not received in the deadline period.
- **total_count_change**: The change in **total_count** since the last time *`on_requested_deadline_missed`* was called or the status was read. It can only have zero or positive values.
- **last_instance_handle**: Handle to the last instance that missed the deadline. If no deadline was ever missed, it will have value `c_InstanceHandle_Unknown`.

> **Warning:** Currently this status is not supported and will be implemented in future releases. As a result, trying to access this status will return `NOT_SUPPORTED` and the corresponding listener will never be called.

RequestedIncompatibleQosStatus

This status changes every time the *DataReader* finds a *DataWriter* that matches the *Topic* and has a common partition, but with a QoS configuration incompatible with the one defined on the *DataReader*.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`
last_policy_id	`uint32_t`
policies	`std::vector<QosPolicyCount>`

- **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 list 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 *QosPolicyCount* for more information the information that is stored for each policy.

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.

QosPolicyCount

This structure holds a counter for a policy.

List of data members:

Data Member Name	Type
policy_id	`int32_t`
count	`int32_t`

- **policy_id**: The ID of the policy.
- **count**: The counter value for the policy.

SampleLostStatus

This status changes every time a new data sample is lost and will never be received.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`

- **total_count**: Total cumulative count of lost samples under the *Topic* of the current *DataReader*.
- **total_count_change**: The change in **total_count** since the last time `on_sample_lost` was called or the status was read. It can only be positive or zero.

Warning: Currently this status is not supported and will be implemented in future releases. As a result, trying to access this status will return `NOT_SUPPORTED` and the corresponding listener will never be called.

SampleRejectedStatus

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

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`
last_reason	`SampleRejectedStatusKind`
last_instance_handle	`InstanceHandle_t`

- **total_count**: Total cumulative count of rejected samples under the *Topic* of the current *DataReader*.

- **total_count_change**: The change in **total_count** since the last time `on_sample_rejected` was called or the status was read. It can only be positive or zero.
- **last_reason**: The reason for rejecting the last rejected sample. If no sample was ever rejected, it will have value `NOT_REJECTED`. See *SampleRejectedStatusKind* for further details.
- **last_instance_handle**: Handle to the last instance whose sample was rejected. If no sample was ever rejected, it will have value `c_InstanceHandle_Unknown`.

Warning: Currently this status is not supported and will be implemented in future releases. As a result, trying to access this status will return `NOT_SUPPORTED` and the corresponding listener will never be called.

SampleRejectedStatusKind

There are four possible values:

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

SubscriptionMatchedStatus

This status changes every time the *DataReader* finds a *DataWriter* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a *DataWriter* that was previously considered to be matched.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`
current_count	`int32_t`
current_count_change	`int32_t`
last_publication_handle	`InstanceHandle_t`

- **total_count**: Total cumulative count of remote *DataWriters* that have been discovered publishing on the same *Topic* and has a common partition and a compatible QoS. They may not all be matched at the moment.
- **total_count_change**: The change in **total_count** since the last time `on_suscription_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_suscription_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*.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`

- **total_count**: Total cumulative count of times that the *DataWriter* failed to assert its liveliness during the period configured on its *DataWriterQos*, becoming considered not *alive*. This count does not change when the *DataWriter* is already considered not *alive* and simply remains not *alive* for another liveliness period.
- **total_count_change**: The change in **total_count** since the last time `on_liveliness_lost` was called or the status was read. It can only have zero or positive values.

OfferedDeadlineMissedStatus

This status changes every time the *DataWriter* fails to provide data within the deadline period configured on its *DataWriterQos*.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`
last_instance_handle	`InstanceHandle_t`

- **total_count**: Total cumulative count of missed deadlines for any instance written by the current *DataWriter*. As the deadline period applies to each instance of the *Topic* independently, the count will will be incremented by one for each instance for which data was not sent in the deadline period.
- **total_count_change**: The change in **total_count** since the last time `on_offered_deadline_missed` was called or the status was read. It can only have zero or positive values.
- **last_instance_handle**: Handle to the last instance that missed the deadline. If no deadline was ever missed, it will have value `c_InstanceHandle_Unknown`.

> **Warning:** Currently this status is not supported and will be implemented in future releases. As a result, trying to access this status will return `NOT_SUPPORTED` and the corresponding listener will never be called.

OfferedIncompatibleQosStatus

This status changes every time the *DataWriter* finds a *DataReader* that matches the *Topic* and has a common partition, but with a QoS configuration that is incompatible with the one defined on the *DataWriter*.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`
last_policy_id	`uint32_t`
policies	`std::vector<QosPolicyCount>`

- **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 list 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 *QosPolicyCount* for more information the information that is stored for each policy.

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.

PublicationMatchedStatus

This status changes every time the *DataWriter* finds a *DataReader* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a *DataReader* that was previously considered to be matched.

List of status data members:

Data Member Name	Type
total_count	`int32_t`
total_count_change	`int32_t`
current_count	`int32_t`
current_count_change	`int32_t`
last_subscription_handle	`InstanceHandle_t`

- **total_count**: Total cumulative count of remote *DataReaders* that have been discovered publishing on the same *Topic* and has a common partition and a compatible QoS. They may not all be matched at the moment.
- **total_count_change**: The change in **total_count** since the last time `on_publication_matched` was called or the status was read. It can only have zero or positive values.
- **current_count**: The number of remote *DataReaders* currently matched to the *DataWriter*.
- **current_count_change**: The change in **current_count** since the last time `on_publication_matched` was called or the status was read. It can have positive or negative values.
- **last_subscription_handle**: Handle to the last *DataReader* that matched the *DataWriter*. If no matching ever happened, it will have value `c_InstanceHandle_Unknown`.

6.6.2 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.

Fig. 4: 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 `set_qos()` member function.

As an Entity, `DomainParticipant` accepts a *DomainParticipantListener* that will be notified of status changes on the `DomainParticipant` instance.

DomainParticipantQos

`DomainParticipantQos` controls the behavior of the *DomainParticipant*. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
UserDataQosPolicy	`user_data()`	Yes
EntityFactoryQosPolicy	`entity_factory()`	Yes
ParticipantResourceLimitsQos	`allocation()`	No
PropertyPolicyQos	`properties()`	No
WireProtocolConfigQos	`wire_protocol()`	No
TransportConfigQos	`transport()`	No

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

The QoS value of a previously created *DomainParticipant* can be modified using the `set_qos()` member function. Trying to modify an immutable `QosPolicy` on an already enabled *DomainParticipant* will result on an error. In such case, no changes will be applied and the *DomainParticipant* will keep its previous `DomainParticipantQos`.

```
// Create a DomainParticipant with default DomainParticipantQos
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
```

(continues on next page)

(continued from previous page)

```
if (nullptr != participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
DomainParticipantQos qos = participant->get_qos();

// Modify QoS attributes
qos.entity_factory().autoenable_created_entities = false;

// Assign the new Qos to the object
participant->set_qos(qos);
```

Default DomainParticipantQos

The default *DomainParticipantQos* refers to the value returned by the `get_default_participant_qos()` member function on the *DomainParticipantFactory* singleton. The special value `PARTICIPANT_QOS_DEFAULT` can be used as QoS argument on `create_participant()` or `set_qos()` member functions to indicate that the current default *DomainParticipantQos* should be used.

When the system starts, the default *DomainParticipantQos* is equivalent to the default constructed value `DomainParticipantQos()`. The default *DomainParticipantQos* can be modified at any time using the `set_default_participant_qos()` member function on the *DomainParticipantFactory* singleton. Modifying the default *DomainParticipantQos* will not affect already existing *DomainParticipant* instances.

```
// Get the current QoS or create a new one from scratch
DomainParticipantQos qos_type1 = DomainParticipantFactory::get_instance()->get_
↪default_participant_qos();

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

// Set as the new default TopicQos
if(DomainParticipantFactory::get_instance()->set_default_participant_qos(qos_type1) !
↪= ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DomainParticipant with the new default DomainParticipantQos.
DomainParticipant* participant_with_qos_type1 =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant_with_qos_type1)
{
    // Error
    return;
}

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

(continues on next page)

(continued from previous page)

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

// Set as the new default TopicQos
if(DomainParticipantFactory::get_instance()->set_default_participant_qos(qos_type2) !
↪= ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Topic with the new default TopicQos.
DomainParticipant* participant_with_qos_type2 =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DomainParticipantQos to the original default constructed␣
↪values
if(DomainParticipantFactory::get_instance()->set_default_participant_qos(PARTICIPANT_
↪QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if(DomainParticipantFactory::get_instance()->set_default_participant_
↪qos(DomainParticipantQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

`set_default_participant_qos()` member function also accepts the value `PARTICIPANT_QOS_DEFAULT` as input argument. This will reset the current default *DomainParticipantQos* to the default constructed value `DomainParticipantQos()`.

```
// Create a custom DomainParticipantQos
DomainParticipantQos custom_qos;

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

// Create a DomainParticipant with a custom DomainParticipantQos

DomainParticipant* participant = DomainParticipantFactory::get_instance()->create_
↪participant(0, custom_qos);
if (nullptr != participant)
{
```

(continues on next page)

(continued from previous page)

```
    // Error
    return;
}

// Set the QoS on the participant to the default
if (participant->set_qos(PARTICIPANT_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if(participant->set_qos(DomainParticipantFactory::get_instance()->get_default_
↪participant_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `PARTICIPANT_QOS_DEFAULT` has different meaning depending on where it is used:

- On `create_participant()` and `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 Listener was able to handle the event.

Additionally, `DomainParticipantListener` adds the following callbacks:

- **on_participant_discovery**: A new *DomainParticipant* is discovered in the same domain, a previously known *DomainParticipant* has been removed, or some *DomainParticipant* has changed its QoS.
- **on_subscriber_discovery**: A new *Subscriber* is discovered in the same domain, a previously known *Subscriber* has been removed, or some *Subscriber* has changed its QoS.
- **on_publisher_discovery**: A new *Publisher* is discovered in the same domain, a previously known *Publisher* has been removed, or some *Publisher* has changed its QoS.
- **on_type_discovery**: A new data Type is discovered in the same domain.
- **on_type_dependencies_reply**: The Type lookup client received a replay to a `getTypeDependencies()` request. This callback can be used to retrieve the new type using the `getTypes()` request and create a new dynamic type using the retrieved type object.

- **on_type_information_received**: A new `TypeInformation` has been received from a newly discovered *DomainParticipant*.
- **onParticipantAuthentication**: Informs about the result of the authentication process of a remote *DomainParticipant* (either on failure or success).

```
class CustomDomainParticipantListener : public DomainParticipantListener
{

public:

    CustomDomainParticipantListener()
    : DomainParticipantListener()
    {
    }

    virtual ~CustomDomainParticipantListener()
    {
    }

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

#if HAVE_SECURITY
    virtual void onParticipantAuthentication(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ParticipantAuthenticationInfo&& info)
    {
        if (info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::AUTHORIZED_PARTICIPANT)
        {
            std::cout << "A participant was authorized" << std::endl;
        }
        else if (info.status ==␣
↪eprosima::fastrtps::rtps::ParticipantAuthenticationInfo::UNAUTHORIZED_PARTICIPANT)
        {
            std::cout << "A participant failed authorization" << std::endl;
        }
    }

#endif

    virtual void on_subscriber_discovery(
```

(continues on next page)

(continued from previous page)

```
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info)
    {
        if (info.status == eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERED_
↪READER)
        {
            std::cout << "New subscriber discovered" << std::endl;
        }
        else if (info.status ==␣
↪eprosima::fastrtps::rtps::ReaderDiscoveryInfo::REMOVED_READER)
        {
            std::cout << "New subscriber lost" << std::endl;
        }
    }

    virtual void on_publisher_discovery(
            DomainParticipant* /*participant*/,
            eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info)
    {
        if (info.status == eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERED_
↪WRITER)
        {
            std::cout << "New publisher discovered" << std::endl;
        }
        else if (info.status ==␣
↪eprosima::fastrtps::rtps::WriterDiscoveryInfo::REMOVED_WRITER)
        {
            std::cout << "New publisher lost" << std::endl;
        }
    }

    virtual void on_type_discovery(
            DomainParticipant* participant,
            const eprosima::fastrtps::rtps::SampleIdentity& request_sample_id,
            const eprosima::fastrtps::string_255& topic,
            const eprosima::fastrtps::types::TypeIdentifier* identifier,
            const eprosima::fastrtps::types::TypeObject* object,
            eprosima::fastrtps::types::DynamicType_ptr dyn_type)
    {
        (void)participant, (void)request_sample_id, (void)topic, (void)identifier,␣
↪(void)object, (void)dyn_type;
        std::cout << "New data type discovered" << std::endl;

    }

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

    virtual void on_type_information_received(
            DomainParticipant* participant,
```

(continues on next page)

(continued from previous page)

```
            const eprosima::fastrtps::string_255 topic_name,
            const eprosima::fastrtps::string_255 type_name,
            const eprosima::fastrtps::types::TypeInformation& type_information)
    {
        (void)participant, (void)topic_name, (void)type_name, (void)type_information;
        std::cout << "New data type information received" << std::endl;
    }
};
```

DomainParticipantFactory

The sole purpose of this class is to allow the creation and destruction of *DomainParticipant* objects. `DomainParticipantFactory` itself has no factory, it is a singleton object that can be accessed through the `get_instance()` static member function on the `DomainParticipantFactory` class.

The behavior of the `DomainParticipantFactory` can be modified with the QoS values specified on *DomainParticipantFactoryQos*. Since the `DomainParticipantFactory` is a singleton, its QoS can only be modified with the `set_qos()` member function.

`DomainParticipantFactory` does not accept any Listener, since it is not an Entity.

DomainParticipantFactoryQos

`DomainParticipantFactory` controls the behavior of the *DomainParticipantFactory*. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
EntityFactoryQosPolicy	`entity_factory()`	Yes

Since the *DomainParticipantFactory* is a singleton, its QoS can only be modified with the `set_qos()` member function.

```
DomainParticipantFactoryQos qos;

// Setting autoenable_created_entities to true makes the created DomainParticipants
// to be enabled upon creation
qos.entity_factory().autoenable_created_entities = true;
if(DomainParticipantFactory::get_instance()->set_qos(qos) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DomainParticipant with the new DomainParticipantFactoryQos.
// The returned DomainParticipant is already enabled
DomainParticipant* enabled_participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != enabled_participant)
{
    // Error
    return;
}
```

(continues on next page)

(continued from previous page)

```
// Setting autoenable_created_entities to false makes the created DomainParticipants
// to be disabled upon creation
qos.entity_factory().autoenable_created_entities = false;
if(DomainParticipantFactory::get_instance()->set_qos(qos) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DomainParticipant with the new DomainParticipantFactoryQos.
// The returned DomainParticipant is disabled and will need to be enabled explicitly
DomainParticipant* disabled_participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != disabled_participant)
{
    // Error
    return;
}
```

Loading profiles from an XML file

To create Entities based on XML profiles, the file containing such profiles must be loaded first.

If the profile is described in one of the default loaded files, it will be automatically available on initialization. Otherwise, `load_XML_profiles_file()` member function can be used to load the profiles in the XML. See section *XML profiles* for more information regarding XML profile format and automatic loading.

Once loaded, the name of the profiles can be used to create Entities that will have QoS settings according to the profile specifications.

```
// Load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Profiles can now be used to create Entities
DomainParticipant* participant_with_profile =
        DomainParticipantFactory::get_instance()->create_participant_with_profile(0,
↪"participant_profile");
if (nullptr != participant_with_profile)
{
    // Error
    return;
}
```

Creating a DomainParticipant

Creation of a *DomainParticipant* is done with the `create_participant()` member function on the *DomainParticipantFactory* singleton, that acts as a factory for the *DomainParticipant*.

Mandatory arguments are:

- The domainId that identifies the domain where the *DomainParticipant* will be created.
- The *DomainParticipantQos* describing the behavior of the *DomainParticipant*. If the provided value is `TOPIC_QOS_DEFAULT`, the value of the *DomainParticipantQos* is used.

Optional arguments are:

- A Listener derived from *DomainParticipantListener*, implementing the callbacks that will be triggered in response to events and state changes on the *DomainParticipant*. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the *DomainParticipantListener*. By default all events are enabled.

`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.
- The name of the profile to be applied to the *DomainParticipant*.

Optional arguments are:

- A Listener derived from *DomainParticipantListener*, implementing the callbacks that will be triggered in response to events and state changes on the *DomainParticipant*. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the *DomainParticipantListener*. By default all events are enabled.

`create_participant_with_profile()` will return a null pointer if there was an error during the operation, e.g if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DomainParticipant using a profile and no Listener
DomainParticipant* participant_with_profile =
        DomainParticipantFactory::get_instance()->create_participant_with_profile(0,
↪"participant_profile");
if (nullptr != participant_with_profile)
{
    // Error
    return;
}

// Create a DomainParticipant using a profile and a custom Listener.
// CustomDomainParticipantListener inherits from DomainParticipantListener.
CustomDomainParticipantListener custom_listener;
DomainParticipant* participant_with_profile_and_custom_listener =
        DomainParticipantFactory::get_instance()->create_participant_with_profile(0,
↪"participant_profile", &custom_listener);
if (nullptr != participant_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a DomainParticipant

A *DomainParticipant* can be deleted with the `delete_participant()` member function on the *DomainParticipantFactory* singleton.

Note: A DomainParticipant can only be deleted if all domain Entities belonging to the participant (Publisher, Subscriber or Topic) have already been deleted. Otherwise, the function will issue an error and the DomainParticipant will not be deleted.

```
// Create a DomainParticipant
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

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

// Delete the DomainParticipant
if (DomainParticipantFactory::get_instance()->delete_participant(participant) !=␣
↪ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

6.6.3 Publisher

A publication is defined by the association of a *DataWriter* to a *Publisher*. To start publishing the values of a data instance, the application creates a new *DataWriter* in a *Publisher*. This *DataWriter* will be bound to the *Topic* that describes the data type that is being transmitted. Remote subscriptions that match with this *Topic* will be able to receive the data value updates from the *DataWriter*.

Publisher

The `Publisher` acts on behalf of one or several *DataWriter* objects that belong to it. It serves as a container that allows grouping different *DataWriter* objects under a common configuration given by the `PublisherQos` of the `Publisher`.

DataWriter objects that belong to the same `Publisher` do not have any other relation among each other beyond the `PublisherQos` of the `Publisher` and act independently otherwise. Specifically, a `Publisher` can host *DataWriter* objects for different topics and data types.

PublisherQos

`PublisherQos` controls the behavior of the *Publisher*. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
PresentationQosPolicy	`presentation()`	Yes
PartitionQosPolicy	`partition()`	Yes
GroupDataQosPolicy	`group_data()`	Yes
EntityFactoryQosPolicy	`entity_factory()`	Yes

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

The QoS value of a previously created *Publisher* can be modified using the `set_qos()` member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Create a Publisher with default PublisherQos
Publisher* publisher =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr != publisher)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
PublisherQos qos = publisher->get_qos();

// 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 `set_qos()` member functions to indicate that the current default *PublisherQos* should be used.

When the system starts, the default *PublisherQos* is equivalent to the default constructed value `PublisherQos()`. The default *PublisherQos* can be modified at any time using the `set_default_publisher_qos()` member function on the *DomainParticipant* instance. Modifying the default *PublisherQos* will not affect already existing *Publisher* instances.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
PublisherQos qos_type1 = participant->get_default_publisher_qos();

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

// Set as the new default PublisherQos
if(participant->set_default_publisher_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Publisher with the new default PublisherQos.
Publisher* publisher_with_qos_type1 =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr != publisher_with_qos_type1)
{
    // Error
    return;
}

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

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

// Set as the new default PublisherQos
if(participant->set_default_publisher_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Publisher with the new default PublisherQos.
Publisher* publisher_with_qos_type2 =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr != publisher_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default PublisherQos to the original default constructed values
if(participant->set_default_publisher_qos(PUBLISHER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
```

(continues on next page)

(continued from previous page)

```
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if(participant->set_default_publisher_qos(PublisherQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

`set_default_publisher_qos()` member function also accepts the special value `PUBLISHER_QOS_DEFAULT` as input argument. This will reset the current default *PublisherQos* to default constructed value `PublisherQos()`.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Create a custom PublisherQos
PublisherQos custom_qos;

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

// Create a publisher with a custom PublisherQos
Publisher* publisher = participant->create_publisher(custom_qos);
if (nullptr != publisher)
{
    // Error
    return;
}

// Set the QoS on the publisher to the default
if (publisher->set_qos(PUBLISHER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if(publisher->set_qos(participant->get_default_publisher_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `PUBLISHER_QOS_DEFAULT` has different meaning depending on where it is used:

- On `create_publisher()` and `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.

`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.
```

(continues on next page)

(continued from previous page)

```
Publisher* publisher_with_default_qos =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
if (nullptr != publisher_with_default_qos)
{
    // Error
    return;
}

// A custom PublisherQos can be provided to the creation method
PublisherQos custom_qos;

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

Publisher* publisher_with_custom_qos =
        participant->create_publisher(custom_qos);
if (nullptr != publisher_with_custom_qos)
{
    // Error
    return;
}

// Create a Publisher with default QoS and a custom Listener.
// CustomPublisherListener inherits from PublisherListener.
// The value PUBLISHER_QOS_DEFAULT is used to denote the default QoS.
CustomPublisherListener custom_listener;
Publisher* publisher_with_default_qos_and_custom_listener =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT, &custom_listener);
if (nullptr != publisher_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a Publisher

Instead of using a *PublisherQos*, the name of a profile can be used to create a *Publisher* with the `create_publisher_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
    return;
}

// Create a Publisher using a profile and a custom Listener.
// CustomPublisherListener inherits from PublisherListener.
CustomPublisherListener custom_listener;
Publisher* publisher_with_profile_and_custom_listener =
        participant->create_publisher_with_profile("publisher_profile", &custom_
↪listener);
if (nullptr != publisher_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a Publisher

A *Publisher* can be deleted with the `delete_publisher()` member function on the *DomainParticipant* instance where the *Publisher* was created.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Create a Publisher
Publisher* publisher =
        participant->create_publisher(PUBLISHER_QOS_DEFAULT);
```

(continues on next page)

(continued from previous page)

```
if (nullptr != publisher)
{
    // Error
    return;
}

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

// Delete the Publisher
if (participant->delete_publisher(publisher) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

DataWriter

A `DataWriter` is attached to exactly one *Publisher* that acts as a factory for it. Additionally, each `DataWriter` is bound to a single *Topic* since its creation. This *Topic* must exist prior to the creation of the `DataWriter`, and must be bound to the data type that the `DataWriter` wants to publish.

The effect of creating a new `DataWriter` in a *Publisher* for a specific *Topic* is to initiate a new publication with the name and data type described by the *Topic*.

Once the `DataWriter` is created, the application can inform of changes in the data value using the `write()` member function on the `DataWriter`. These changes will be transmitted to all subscriptions matched with this publication.

DataWriterQos

`DataWriterQos` controls the behavior of the *DataWriter*. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
DurabilityQosPolicy	`durability()`	No
DurabilityServiceQosPolicy	`durability_service()`	Yes
DeadlineQosPolicy	`deadline()`	Yes
LatencyBudgetQosPolicy	`latency_budget()`	Yes
LivelinessQosPolicy	`liveliness()`	No
ReliabilityQosPolicy	`reliability()`	No (*)
DestinationOrderQosPolicy	`destination_order()`	No
HistoryQosPolicy	`history()`	Yes
ResourceLimitsQosPolicy	`resource_limits()`	Yes
TransportPriorityQosPolicy	`transport_priority()`	Yes
LifespanQosPolicy	`lifespan()`	Yes
UserDataQosPolicy	`user_data()`	Yes
OwnershipQosPolicy	`ownership()`	No
OwnershipStrengthQosPolicy	`ownership_strength()`	Yes
WriterDataLifecycleQosPolicy	`writer_data_lifecycle()`	Yes
PublishModeQosPolicy	`publish_mode()`	Yes
DataRepresentationQosPolicy	`representation()`	Yes
PropertyPolicyQos	`properties()`	Yes
RTPSReliableWriterQos	`reliable_writer_qos()`	Yes
RTPSEndpointQos	`endpoint()`	Yes
WriterResourceLimitsQos	`writer_resource_limits()`	Yes
ThroughputControllerDescriptor	`throughput_controller()`	Yes

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

Note: Reliability kind (whether the publication is reliable or best effort) is not mutable. However, the `max_blocking_time` data member of ReliabilityQosPolicy can be modified any time.

The QoS value of a previously created *DataWriter* can be modified using the `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 `set_qos()` member functions to indicate that the current default *DataWriterQos* should be used.

When the system starts, the default *DataWriterQos* is equivalent to the default constructed value `DataWriterQos()`. The default *DataWriterQos* can be modified at any time using the `set_default_dataWriter_qos()` member function on the *Publisher* instance. Modifying the default *DataWriterQos* will not affect already existing *DataWriter* instances.

```
// Get the current QoS or create a new one from scratch
DataWriterQos qos_type1 = publisher->get_default_datawriter_qos();

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

// Set as the new default DataWriterQos
if(publisher->set_default_datawriter_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DataWriter with the new default DataWriterQos.
DataWriter* data_writer_with_qos_type1 =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr != data_writer_with_qos_type1)
{
    // Error
    return;
}

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

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

// Set as the new default DataWriterQos
if(publisher->set_default_datawriter_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// 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
```

(continues on next page)

(continued from previous page)

```
if(publisher->set_default_datawriter_qos(DATAWRITER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if(publisher->set_default_datawriter_qos(DataWriterQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

`set_default_dataWriter_qos()` member function also accepts the special value `DATAWRITER_QOS_DEFAULT` as input argument. This will reset the current default *DataWriterQos* to default constructed value `DataWriterQos()`.

```
// Create a custom DataWriterQos
DataWriterQos custom_qos;

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

// Create a DataWriter with a custom DataWriterQos
DataWriter* data_writer = publisher->create_datawriter(topic, custom_qos);
if (nullptr != data_writer)
{
    // Error
    return;
}

// Set the QoS on the DataWriter to the default
if (data_writer->set_qos(DATAWRITER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if(data_writer->set_qos(publisher->get_default_datawriter_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `DATAWRITER_QOS_DEFAULT` has different meaning depending on where it is used:

- On `create_datawriter()` and `set_qos()` it refers to the default *DataWriterQos* as returned by `get_default_dataWriter_qos()`.
- On `set_default_dataWriter_qos()` it refers to the default constructed `DataWriterQos()`.

DataWriterListener

`DataWriterListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *DataWriter*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`DataWriterListener` defines the following callbacks:

- **on_publication_matched**: The *DataWriter* has found a *DataReader* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a *DataReader* that was previously considered to be matched.
- **on_offered_deadline_missed**: The *DataWriter* failed to provide data within the deadline period configured on its *DataWriterQos*. It will be called for each deadline period and data instance for which the *DataWriter* failed to provide data.

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

- **on_offered_incompatible_qos**: The *DataWriter* has found a *DataReader* that matches the *Topic* and has a common partition, but with a requested QoS that is incompatible with the one defined on the *DataWriter*.

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

- **on_liveliness_lost**: The *DataWriter* did not respect the liveliness configuration on its *DataWriterQos*, and therefore, *DataReader* entities will consider the *DataWriter* as no longer *active*.

```
class CustomDataWriterListener : public DataWriterListener
{

public:

    CustomDataWriterListener()
    : DataWriterListener()
    {
    }

    virtual ~CustomDataWriterListener()
    {
    }

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

(continues on next page)

(continued from previous page)

```
        {
            std::cout << "Unmatched a remote Subscriber" << std::endl;
        }
    }

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

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

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

Creating a DataWriter

A *DataWriter* always belongs to a *Publisher*. Creation of a *DataWriter* is done with the `create_datawriter()` member function on the *Publisher* instance, that acts as a factory for the *DataWriter*.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- The *DataWriterQos* describing the behavior of the *DataWriter*. If the provided value is `DATAWRITER_QOS_DEFAULT`, the value of the *Default DataWriterQos* is used.

Optional arguments are:

- A Listener derived from *DataWriterListener*, implementing the callbacks that will be triggered in response to events and state changes on the *DataWriter*. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the *DataWriterListener*. By default all events are enabled.

`create_datawriter()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DataWriter with default DataWriterQos and no Listener
// The value DATAWRITER_QOS_DEFAULT is used to denote the default QoS.
DataWriter* data_writer_with_default_qos =
```

(continues on next page)

(continued from previous page)

```
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr != data_writer_with_default_qos)
{
    // Error
    return;
}

// A custom DataWriterQos can be provided to the creation method
DataWriterQos custom_qos;

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

DataWriter* data_writer_with_custom_qos =
        publisher->create_datawriter(topic, custom_qos);
if (nullptr != data_writer_with_custom_qos)
{
    // Error
    return;
}

// Create a DataWriter with default QoS and a custom Listener.
// CustomDataWriterListener inherits from DataWriterListener.
// The value DATAWRITER_QOS_DEFAULT is used to denote the default QoS.
CustomDataWriterListener custom_listener;
DataWriter* data_writer_with_default_qos_and_custom_listener =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT, &custom_listener);
if (nullptr != data_writer_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a DataWriter

Instead of using a *DataWriterQos*, the name of a profile can be used to create a *DataWriter* with the `create_datawriter_with_profile()` member function on the *Publisher* instance.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- A string with the name that identifies the *DataWriter*.

Optional arguments are:

- A Listener derived from *DataWriterListener*, implementing the callbacks that will be triggered in response to events and state changes on the *DataWriter*. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the *DataWriterListener*. By default all events are enabled.

`create_datawriter_with_profile()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DataWriter using a profile and no Listener
DataWriter* data_writer_with_profile =
        publisher->create_datawriter_with_profile(topic, "data_writer_profile");
if (nullptr != data_writer_with_profile)
{
    // Error
    return;
}

// Create a DataWriter using a profile and a custom Listener.
// CustomDataWriterListener inherits from DataWriterListener.
CustomDataWriterListener custom_listener;
DataWriter* data_writer_with_profile_and_custom_listener =
        publisher->create_datawriter_with_profile(topic, "data_writer_profile", &
↪custom_listener);
if (nullptr != data_writer_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a DataWriter

A *DataWriter* can be deleted with the `delete_datawriter()` member function on the *DomainParticipant* instance where the *DataWriter* was created.

```
// Create a DataWriter
DataWriter* data_writer =
        publisher->create_datawriter(topic, DATAWRITER_QOS_DEFAULT);
if (nullptr != data_writer)
{
    // Error
    return;
}

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

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

Publishing data

The user informs of a change in the value of a data instance with the `write()` member function on the *DataWriter*. This change will then be communicated to every *DataReader* matched with the *DataWriter*. As a side effect, this operation asserts liveliness on the *DataWriter* itself, the *Publisher* and the *DomainParticipant*.

The function takes two arguments:

- A pointer to the data instance with the new values.
- The handler to the instance.

An empty (i.e., default constructed `InstanceHandle_t()`) instance handler can be used for the argument handle. This indicates that the identity of the instance should be automatically deduced from the key of the instance data. Alternatively, the member function `write()` is overloaded to take only the pointer to the data instance, which will always deduced the identity from the key of the instance data.

If the handle is not empty, then it must correspond to the value obtained with the `getKey()` of the `TypeSupport` instance. Otherwise the write function will fail with `RETCODE_PRECONDITION_NOT_MET`.

```
// Register the data type in the DomainParticipant.
TypeSupport custom_type_support(new CustomDataType());
custom_type_support.register_type(participant, custom_type_support.get_type_name());

// Create a Topic with the registered type.
Topic* custom_topic =
        participant->create_topic("topic_name", custom_type_support.get_type_name(),␣
↪TOPIC_QOS_DEFAULT);
if (nullptr != custom_topic)
{
    // Error
    return;
}

// Create a DataWriter
DataWriter* data_writer =
        publisher->create_datawriter(custom_topic, DATAWRITER_QOS_DEFAULT);
if (nullptr != data_writer)
{
    // Error
    return;
}

// Get a data instance
void* data = custom_type_support->createData();

// Fill the data values
// (...)

// Publish the new value, deduce the instance handle
if (data_writer->write(data, eprosima::fastrtps::rtps::InstanceHandle_t()) !=␣
↪ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The data instance can be reused to publish new values,
// but delete it at the end to avoid leaks
```

(continues on next page)

(continued from previous page)

```
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`.

6.6.4 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. 5: Subscriber class diagram

Subscriber

The *Subscriber* acts on behalf of one or several *DataReader* objects that belong to it. It serves as a container that allows grouping different *DataReader* objects under a common configuration given by the *SubscriberQos* of the *Subscriber*.

DataReader objects that belong to the same *Subscriber* do not have any other relation among each other beyond the `SubscriberQos` of the `Subscriber` and act independently otherwise. Specifically, a *Subscriber* can host *DataReader* objects for different topics and data types.

SubscriberQos

SubscriberQos controls the behavior of the *Subscriber*. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
PresentationQosPolicy	`presentation`	Yes
PartitionQosPolicy	`partition`	Yes
GroupDataQosPolicy	`group_data`	Yes
EntityFactoryQosPolicy	`entity_factory`	Yes

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

The QoS value of a previously created *Subscriber* can be modified using the `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 `set_qos()` member functions to indicate that the current default *SubscriberQos* should be used.

When the system starts, the default *SubscriberQos* is equivalent to the default constructed value `SubscriberQos()`. The default *SubscriberQos* can be modified at any time using the `set_default_subscriber_qos()` member function on the *DomainParticipant* instance. Modifying the default *SubscriberQos* will not affect already existing *Subscriber* instances.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
SubscriberQos qos_type1 = participant->get_default_subscriber_qos();

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

(continues on next page)

(continued from previous page)

```
// Set as the new default SubscriberQos
if(participant->set_default_subscriber_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Subscriber with the new default SubscriberQos.
Subscriber* subscriber_with_qos_type1 =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr != subscriber_with_qos_type1)
{
    // Error
    return;
}

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

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

// Set as the new default SubscriberQos
if(participant->set_default_subscriber_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Subscriber with the new default SubscriberQos.
Subscriber* subscriber_with_qos_type2 =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr != subscriber_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default SubscriberQos to the original default constructed values
if(participant->set_default_subscriber_qos(SUBSCRIBER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if(participant->set_default_subscriber_qos(SubscriberQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

`set_default_subscriber_qos()` member function also accepts the special value `SUBSCRIBER_QOS_DEFAULT` as input argument. This will reset the current default *SubscriberQos* to de-

fault constructed value `SubscriberQos()`.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Create a custom SubscriberQos
SubscriberQos custom_qos;

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

// Create a subscriber with a custom SubscriberQos
Subscriber* subscriber = participant->create_subscriber(custom_qos);
if (nullptr != subscriber)
{
    // Error
    return;
}

// Set the QoS on the subscriber to the default
if (subscriber->set_qos(SUBSCRIBER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if(subscriber->set_qos(participant->get_default_subscriber_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `SUBSCRIBER_QOS_DEFAULT` has different meaning depending on where it is used:

- On `create_subscriber()` and `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.

Additionally, *SubscriberListener* adds the following callback:

- `on_data_on_readers`: New data is available on any *DataReader* belonging to this *Subscriber*. There is no queuing of invocations to this callback, meaning that if several new data changes are received at once, only one callback invocation may be issued for all of them, instead of one per change. If the application is retrieving the received data on this callback, it must keep *reading data* until no new changes are left.

```
class CustomSubscriberListener : public SubscriberListener
{

public:

    CustomSubscriberListener()
    : SubscriberListener()
    {
    }

    virtual ~CustomSubscriberListener()
    {
    }

    virtual void on_data_on_readers(
            Subscriber* sub)
    {
        (void)sub;
        std::cout << "New data available" << std::endl;
    }
};
```

Creating a Subscriber

A *Subscriber* always belongs to a *DomainParticipant*. Creation of a *Subscriber* is done with the `create_subscriber()` member function on the *DomainParticipant* instance, that acts as a factory for the *Subscriber*.

Mandatory arguments are:

- The *SubscriberQos* describing the behavior of the *Subscriber*. If the provided value is `SUBSCRIBER_QOS_DEFAULT`, the value of the *Default SubscriberQos* is used.

Optional arguments are:

- A Listener derived from *SubscriberListener*, implementing the callbacks that will be triggered in response to events and state changes on the *Subscriber*. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the *SubscriberListener*. By default all events are enabled.

`create_subscriber()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Create a Subscriber with default SubscriberQos and no Listener
// The value SUBSCRIBER_QOS_DEFAULT is used to denote the default QoS.
Subscriber* subscriber_with_default_qos =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT);
if (nullptr != subscriber_with_default_qos)
{
    // Error
    return;
}

// A custom SubscriberQos can be provided to the creation method
SubscriberQos custom_qos;

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

Subscriber* subscriber_with_custom_qos =
        participant->create_subscriber(custom_qos);
if (nullptr != subscriber_with_custom_qos)
{
    // Error
    return;
}

// Create a Subscriber with default QoS and a custom Listener.
// CustomSubscriberListener inherits from SubscriberListener.
// The value SUBSCRIBER_QOS_DEFAULT is used to denote the default QoS.
CustomSubscriberListener custom_listener;
Subscriber* subscriber_with_default_qos_and_custom_listener =
        participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT, &custom_listener);
if (nullptr != subscriber_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a Subscriber

Instead of using a *SubscriberQos*, the name of a profile can be used to create a *Subscriber* with the `create_subscriber_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.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

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

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

// Delete the Subscriber
if (participant->delete_subscriber(subscriber) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

DataReader

A *DataReader* is attached to exactly one *Subscriber* that acts as a factory for it. Additionally, each *DataReader* is bound to a single *Topic* since its creation. This *Topic* must exist prior to the creation of the *DataReader*, and must be bound to the data type that the *DataReader* wants to publish.

The effect of creating a new *DataReader* in a *Subscriber* for a specific *Topic* is to initiate a new subscription with the name and data type described by the *Topic*.

Once the *DataReader* is created, the application will be informed when changes in the data value are received from remote publications. These changes can then be retrieved using the `read()` or `take()` member functions of the *DataReader*.

DataReaderQos

DataReaderQos controls the behavior of the *DataReader*. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor/Mutator	Mutable
DurabilityQosPolicy	`durability()`	No
DurabilityServiceQosPolicy	`durability_service()`	Yes
DeadlineQosPolicy	`deadline()`	Yes
LatencyBudgetQosPolicy	`latency_budget()`	Yes
LivelinessQosPolicy	`liveliness()`	No
ReliabilityQosPolicy	`reliability()`	No (*)
DestinationOrderQosPolicy	`destination_order()`	No
HistoryQosPolicy	`history()`	No
ResourceLimitsQosPolicy	`resource_limits()`	No
LifespanQosPolicy	`lifespan()`	Yes
UserDataQosPolicy	`user_data()`	Yes
OwnershipQosPolicy	`ownership()`	No
PropertyPolicyQos	`properties()`	Yes
RTPSEndpointQos	`endpoint()`	Yes
ReaderResourceLimitsQos	`reader_resource_limits()`	Yes
TimeBasedFilterQosPolicy	`time_based_filter()`	Yes
ReaderDataLifecycleQosPolicy	`reader_data_lifecycle()`	Yes
RTPSReliableReaderQos	`reliable_reader_qos()`	Yes
TypeConsistencyQos	`type_consistency()`	Yes
`boolean`	`expects_inline_qos()`	Yes

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

Note: Reliability kind (whether the publication is reliable or best effort) is not mutable. However, the `max_blocking_time` data member of *ReliabilityQosPolicy* can be modified any time.

The QoS value of a previously created *DataReader* can be modified using the `set_qos()` member function.

```
// Create a DataReader with default DataReaderQos
DataReader* data_reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr != data_reader)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
DataReaderQos qos = data_reader->get_qos();

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

// Assign the new Qos to the object
data_reader->set_qos(qos);
```

Default DataReaderQos

The default *DataReaderQos* refers to the value returned by the `get_default_datareader_qos()` member function on the *Subscriber* instance. The special value `DATAREADER_QOS_DEFAULT` can be used as QoS argument on `create_datareader()` or `set_qos()` member functions to indicate that the current default *DataReaderQos* should be used.

When the system starts, the default *DataReaderQos* is equivalent to the default constructed value `DataReaderQos()`. The default *DataReaderQos* can be modified at any time using the `set_default_datareader_qos()` member function on the *Subscriber* instance. Modifying the default *DataReaderQos* will not affect already existing *DataReader* instances.

```
// Get the current QoS or create a new one from scratch
DataReaderQos qos_type1 = subscriber->get_default_datareader_qos();

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

// Set as the new default DataReaderQos
if(subscriber->set_default_datareader_qos(qos_type1) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DataReader with the new default DataReaderQos.
DataReader* data_reader_with_qos_type1 =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr != data_reader_with_qos_type1)
{
    // Error
    return;
}

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

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

// Set as the new default DataReaderQos
if(subscriber->set_default_datareader_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a DataReader with the new default DataReaderQos.
DataReader* data_reader_with_qos_type2 =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr != data_reader_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default DataReaderQos to the original default constructed values
```

(continues on next page)

(continued from previous page)

```
if(subscriber->set_default_datareader_qos(DATAREADER_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if(subscriber->set_default_datareader_qos(DataReaderQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

`set_default_datareader_qos()` member function also accepts the special value `DATAREADER_QOS_DEFAULT` as input argument. This will reset the current default *DataReaderQos* to default constructed value `DataReaderQos()`.

```
// Create a custom DataReaderQos
DataReaderQos custom_qos;

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

// Create a DataWriter with a custom DataReaderQos
DataReader* data_reader = subscriber->create_datareader(topic, custom_qos);
if (nullptr != data_reader)
{
    // Error
    return;
}

// Set the QoS on the DataWriter to the default
if (data_reader->set_qos(DATAREADER_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if(data_reader->set_qos(subscriber->get_default_datareader_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `DATAREADER_QOS_DEFAULT` has different meaning depending on where it is used:

- On `create_datareader()` and `set_qos()` it refers to the default *DataReaderQos* as returned by `get_default_datareader_qos()`.
- On `set_default_dataReader_qos()` it refers to the default constructed `DataReaderQos()`.

DataReaderListener

DataReaderListener is an abstract class defining the callbacks that will be triggered in response to state changes on the *DataReader*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

DataReaderListener defines the following callbacks:

- `on_data_available`: There is new data available for the application on the *DataReader*. There is no queuing of invocations to this callback, meaning that if several new data changes are received at once, only one callback invocation may be issued for all of them, instead of one per change. If the application is retrieving the received data on this callback, it must keep *reading data* until no new changes are left.
- `on_subscription_matched`: The *DataReader* has found a *DataWriter* that matches the *Topic* and has a common partition and a compatible QoS, or has ceased to be matched with a *DataWriter* that was previously considered to be matched. It is also triggered when a matched *DataWriter* has changed its *DataWriterQos*.
- `on_requested_deadline_missed`: The *DataReader* did not receive data within the deadline period configured on its *DataReaderQos*. It will be called for each deadline period and data instance for which the *DataReader* missed data.

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

- `on_requested_incompatible_qos`: The *DataReader* has found a *DataWriter* that matches the *Topic* and has a common partition, but with a QoS that is incompatible with the one defined on the *DataReader*.

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

- `on_liveliness_changed`: The liveliness status of a matched *DataWriter* has changed. Either a *DataWriter* that was *inactive* has become *active* or the other way around.
- `on_sample_rejected`: A received data sample was rejected.

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

- `on_sample_lost`: A data sample was lost and will never be received.

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

```
class CustomDataReaderListener : public DataReaderListener
{

public:

    CustomDataReaderListener()
    : DataReaderListener()
    {
```

(continues on next page)

(continued from previous page)

```
    }

    virtual ~CustomDataReaderListener()
    {
    }

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

    virtual void on_subscription_matched(
            DataReader* reader,
            const SubscriptionMatchedStatus& info)
    {
        (void)reader;
        if (info.current_count_change == 1)
        {
            std::cout << "Matched a remote DataWriter" << std::endl;
        }
        else if (info.current_count_change == -1)
        {
            std::cout << "Unmatched a remote DataWriter" << std::endl;
        }
    }

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

    virtual void on_liveliness_changed(
            DataReader* reader,
            const eprosima::fastrtps::LivelinessChangedStatus& info)
    {
        (void)reader;
        if (info.alive_count_change == 1)
        {
            std::cout << "A matched DataWriter has become active" << std::endl;
        }
        else if (info.not_alive_count_change == 1)
        {
            std::cout << "A matched DataWriter has become inactive" << std::endl;
        }
    }

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

(continues on next page)

(continued from previous page)

```
    }

    virtual void on_requested_incompatible_qos(
            DataReader* reader,
            const RequestedIncompatibleQosStatus& info)
    {
        (void)reader, (void)info;
        std::cout << "Found a remote Topic with incompatible QoS" << std::endl;
    }

    virtual void on_sample_lost(
            DataReader* reader,
            const SampleLostStatus& info)
    {
        (void)reader, (void)info;
        std::cout << "A data sample was lost and will not be received" << std::endl;
    }
};
```

Creating a DataReader

A *DataReader* always belongs to a *Subscriber*. Creation of a *DataReader* is done with the `create_datareader()` member function on the *Subscriber* instance, that acts as a factory for the *DataReader*.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- The *DataReaderQos* describing the behavior of the *DataReader*. If the provided value is `DATAREADER_QOS_DEFAULT`, the value of the *Default DataReaderQos* is used.

Optional arguments are:

- A Listener derived from *DataReaderListener*, implementing the callbacks that will be triggered in response to events and state changes on the *DataReader*. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the *DataReaderListener*. By default all events are enabled.

`create_datareader()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

```
// Create a DataReader with default DataReaderQos and no Listener
// The value DATAREADER_QOS_DEFAULT is used to denote the default QoS.
DataReader* data_reader_with_default_qos =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr != data_reader_with_default_qos)
{
    // Error
    return;
}

// A custom DataReaderQos can be provided to the creation method
DataReaderQos custom_qos;

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

(continues on next page)

(continued from previous page)

```
DataReader* data_reader_with_custom_qos =
        subscriber->create_datareader(topic, custom_qos);
if (nullptr != data_reader_with_custom_qos)
{
    // Error
    return;
}

// Create a DataReader with default QoS and a custom Listener.
// CustomDataReaderListener inherits from DataReaderListener.
// The value DATAREADER_QOS_DEFAULT is used to denote the default QoS.
CustomDataReaderListener custom_listener;
DataReader* data_reader_with_default_qos_and_custom_listener =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT, &custom_
↪listener);
if (nullptr != data_reader_with_default_qos_and_custom_listener)
{
    // Error
    return;
}
```

Profile based creation of a DataReader

Instead of using a *DataReaderQos*, the name of a profile can be used to create a *DataReader* with the `create_datareader_with_profile()` member function on the *Subscriber* instance.

Mandatory arguments are:

- A *Topic* bound to the data type that will be transmitted.
- A string with the name that identifies the *DataReader*.

Optional arguments are:

- A Listener derived from *DataReaderListener*, implementing the callbacks that will be triggered in response to events and state changes on the *DataReader*. By default empty callbacks are used.
- A `StatusMask` that activates or deactivates triggering of individual callbacks on the *DataReaderListener*. By default all events are enabled.

`create_datareader_with_profile()` will return a null pointer if there was an error during the operation, e.g. if the provided QoS is not compatible or is not supported. It is advisable to check that the returned value is a valid pointer.

Note: XML profiles must have been loaded previously. See *Loading profiles from an XML file*.

```
// First load the XML with the profiles
DomainParticipantFactory::get_instance()->load_XML_profiles_file("profiles.xml");

// Create a DataReader using a profile and no Listener
DataReader* data_reader_with_profile =
        subscriber->create_datareader_with_profile(topic, "data_reader_profile");
if (nullptr != data_reader_with_profile)
{
```

(continues on next page)

(continued from previous page)

```
    // 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;
}
```

Deleting a DataReader

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

```
// Create a DataReader
DataReader* data_reader =
        subscriber->create_datareader(topic, DATAREADER_QOS_DEFAULT);
if (nullptr != data_reader)
{
    // Error
    return;
}

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

// Delete the DataReader
if (subscriber->delete_datareader(data_reader) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

SampleInfo

When a sample is retrieved form the *DataReader*, in addition to the sample data, a *SampleInfo* instance is returned. This object contains additional information that complements the returned data value and helps on it interpretation. For example, if the *valid_data* value is `false`, the *DataReader* is not informing the application about a new value in the data instance, but a change on its status, and the returned data value must be discarded.

Please, refer to the section *Accessing received data* for more information regarding how received data can be accessed on the *DataReader*.

The following sections describe the data members of *SampleInfo* and the meaning of each one in relation to the returned sample data.

- *sample_state*

- *view_state*
- *instance_state*
- *disposed_generation_count*
- *no_writers_generation_count*
- *sample_rank*
- *generation_rank*
- *absolute_generation_rank*
- *source_timestamp*
- *instance_handle*
- *publication_handle*
- *valid_data*
- *sample_identity*
- *related_sample_identity*

sample_state

A *SampleStateKind* enumeration that indicates whether or not the corresponding data sample has already been read previously. It can take one of these values:

- **READ**: This is the first time this data sample has been retrieved.
- **NOT_READ**: The data sample has already been *read* or *taken* previously.

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

view_state

A *ViewStateKind* enumeration that indicates whether or not this is the very first sample of this data instance that the *DataReader* retrieves. It can take one of these values:

- **NEW**: This is the first time a sample of this instance is retrieved.
- **NOT_NEW**: Other samples of this instance have been retrieved previously.

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

instance_state

A *InstanceStateKind* enumeration that indicates whether the instance is currently in existence or it has been disposed. In the latter case, it also provides information about the reason for the disposal. It can take one of these values:

- **ALIVE**: The instance is currently in existence.
- **NOT_ALIVE_DISPOSED**: A remote *DataWriter* disposed the instance.
- **NOT_ALIVE_NO_WRITERS**: The *DataReader* disposed the instance because no remote *DataWriter* that was publishing the instance is *alive*.

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

disposed_generation_count

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

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

no_writers_generation_count

Indicates the number of times the instance had become alive after it was disposed as `NOT_ALIVE_NO_WRITERS`.

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

sample_rank

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

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

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

It holds the time stamp provided by the *DataWriter* when the sample was published.

instance_handle

The instance handle of the local instance.

publication_handle

The instance handle of the *DataWriter* that published the data change.

valid_data

A boolean indicating 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

This 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

This 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 the `read_next_data()` member function. It reads the next, non-previously accessed data value available on the *DataReader*, and stores it in the provided data buffer.
- **Taking** is done with the `take_next_data()` member function. It reads the next, non-previously accessed data value available on the *DataReader*, and stores it in the provided data buffer. Additionally, it also removes the value from the *DataReader*, so it is no longer accessible.

If there is no unread data in the *DataReader*, both operations will return `NO_DATA` and nothing is returned.

In addition to the data value, the data access operations also provide a *SampleInfo* instance with additional information that help interpreting the returned data value, like the originating *DataWriter* or the publication time stamp. Please, refer to the *SampleInfo* section for an extensive description of its contents.

Accessing data on callbacks

When the *DataReader* new data values from any matching *DataWriter*, it informs the application through two Listener callbacks:

- *`on_data_available`*.
- *`on_data_on_readers`*.

These callbacks can be used to retrieve the newly arrived data, as in the following example.

```
class CustomizedDataReaderListener : public DataReaderListener
{

public:

    CustomizedDataReaderListener()
    : DataReaderListener()
    {
    }

    virtual ~CustomizedDataReaderListener()
    {
    }

    virtual void on_data_available(
            DataReader* reader)
    {
        // Create a data and SampleInfo instance
        void* data = reader->type().create_data();
        SampleInfo info;

        // Keep taking data until there is nothing to take
        while (reader->take_next_sample(&data, &info) == ReturnCode_t::RETCODE_OK)
        {
```

(continues on next page)

(continued from previous page)

```
            if (info.instance_state == ALIVE)
            {
                // 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;
            }
        }

        // The data instance can be reused to retrieve new values,
        // but delete it at the end to avoid leaks
        reader->type().delete_data(data);
    }
};
```

Note: If several new data changes are received at once, the callbacks may be triggered just once, instead of once per change. The application must keep *reading* or *taking* until no new changes are available.

Accessing data with a waiting thread

Instead of relying on the Listener to try and get new data values, the application can also dedicate a thread to wait until any new data is available on the *DataReader*. This can be done with the `wait_for_unread_message()` member function, that blocks until a new data sample is available or the given timeout expires. If no new data was available after the timeout expired, it will return with value `false`. This function returning with value `true` means there is new data available on the *DataReaderListener* 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
void* data = data_reader->type().create_data();
SampleInfo info;

//Define a timeout of 5 seconds
eprosima::fastrtps::Duration_t timeout (5,0);

// Loop reading data as it arrives
// This will make the current threat to be dedicated exclusively to
// waiting and reading data until the remote DataWriter dies
while (true)
{
```

(continues on next page)

(continued from previous page)

```
    if (data_reader->wait_for_unread_message(timeout))
    {
        if (data_reader->take_next_sample(&data, &info) == ReturnCode_t::RETCODE_OK)
        {
            if (info.instance_state == ALIVE)
            {
                // Do something with the data
                std::cout << "Received new data value for topic "
                          << topic->get_name()
                          << std::endl;
            }
            else
            {
                // If the remote writer is not alive, we exit the reading loop
                std::cout << "Remote writer for topic "
                          << topic->get_name()
                          << " is dead" << std::endl;
                break;
            }
        }
    }
    else
    {
        std::cout << "No data this time" << std::endl;
    }
}

// The data instance can be reused to retrieve new values,
// but delete it at the end to avoid leaks
data_reader->type().delete_data(data);
```

6.6.5 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. 6: Topic class diagram

Topic

A `Topic` is a specialization of the broader concept of *TopicDescription*. A `Topic` represents a single data flow between *Publisher* and *Subscriber*, providing:

- The name to identify the data flow.
- The data type that is transmitted on that flow.
- The QoS values related to the data itself.

The behavior of the `Topic` can be modified with the QoS values specified on *TopicQos*. The QoS values can be set at the creation of the `Topic`, or modified later with the `set_qos()` member function.

Like other Entities, `Topic` accepts a Listener that will be notified of status changes on the Topic.

TopicQos

`TopicQos` controls the behavior of the *Topic*. Internally it contains the following `QosPolicy` objects:

QosPolicy class	Accessor	Mutable
TopicDataQosPolicy	`topic_data()`	Yes
DurabilityQosPolicy	`durability()`	Yes
DurabilityServiceQosPolicy	`durability_service()`	Yes
DeadlineQosPolicy	`deadline()`	Yes
LatencyBudgetQosPolicy	`latency_budget()`	Yes
LivelinessQosPolicy	`liveliness()`	Yes
ReliabilityQosPolicy	`reliability()`	Yes
DestinationOrderQosPolicy	`destination_order()`	Yes
HistoryQosPolicy	`history()`	Yes
ResourceLimitsQosPolicy	`resource_limits()`	Yes
TransportPriorityQosPolicy	`transport_priority()`	Yes
LifespanQosPolicy	`lifespan()`	Yes
OwnershipQosPolicy	`ownership()`	Yes
DataRepresentationQosPolicy	`representation()`	Yes

Refer to the detailed description of each `QosPolicy` class for more information about their usage and default values.

The QoS value of a previously created *Topic* can be modified using the `set_qos()` member function.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Create a Topic with default TopicQos
Topic* topic =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr != topic)
{
    // Error
```

(continues on next page)

(continued from previous page)

```
    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 `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 `set_default_topict_qos()` member function on the *DomainParticipant* instance. Modifying the default *TopicQos* will not affect already existing *Topic* instances.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Get the current QoS or create a new one from scratch
TopicQos qos_type1 = participant->get_default_topic_qos();

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

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

(continues on next page)

(continued from previous page)

```
TopicQos qos_type2;

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

// Set as the new default TopicQos
if(participant->set_default_topic_qos(qos_type2) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// Create a Topic with the new default TopicQos.
Topic* topic_with_qos_type2 =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr != topic_with_qos_type2)
{
    // Error
    return;
}

// Resetting the default TopicQos to the original default constructed values
if(participant->set_default_topic_qos(TOPIC_QOS_DEFAULT)
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following
if(participant->set_default_topic_qos(TopicQos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

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

(continues on next page)

(continued from previous page)

```
// Create a topic with a custom TopicQos
Topic* topic = participant->create_topic("TopicName", "DataTypeName", custom_qos);
if (nullptr != topic)
{
    // Error
    return;
}

// Set the QoS on the topic to the default
if (topic->set_qos(TOPIC_QOS_DEFAULT) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}

// The previous instruction is equivalent to the following:
if(topic->set_qos(participant->get_default_topic_qos())
        != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

Note: The value `TOPIC_QOS_DEFAULT` has different meaning depending on where it is used:

- On `create_topic()` and `set_qos()` it refers to the default *TopicQos* as returned by `get_default_topic_qos()`.
- On `set_default_topic_qos()` it refers to the default constructed `TopicQos()`.

TopicDescription

`TopicDescription` is an abstract class that serves as the base for all classes describing a data flow. Applications will not create instances of `TopicDescription` directly, they must create instances of one of its specializations instead. At the moment, the only specialization implemented is *Topic*.

TopicListener

`TopicListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the *Topic*. By default, all these callbacks are empty and do nothing. The user should implement a specialization of this class overriding the callbacks that are needed on the application. Callbacks that are not overridden will maintain their empty implementation.

`TopicListener` has the following callback:

- **on_inconsistent_topic**: A remote *Topic* is discovered with the same name but different characteristics as another locally created *Topic*.

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

```
class CustomTopicListener : public TopicListener
{

public:

    CustomTopicListener()
    : TopicListener()
    {
    }

    virtual ~CustomTopicListener()
    {
    }

    virtual void on_inconsistent_topic(
            Topic* topic,
            InconsistentTopicStatus status)
    {
        (void)topic, (void)status;
        std::cout << "Inconsistent topic received discovered" << std::endl;
    }
};
```

Definition of data types

The definition of the data type exchanged in a *Topic* is divided in two classes: the `TypeSupport` and the `TopicDataType`.

`TopicDataType` describes the data type exchanged between a publication and a subscription, i.e., the data corresponding to a *Topic*. The user has to create a specialized class for each specific type that will be used by the application.

Any specialization of `TopicDataType` must be registered in the *DomainParticipant* before it can be used to create *Topic* objects. A `TypeSupport` object encapsulates an instance of `TopicDataType`, providing the functions needed to register the type and interact with the publication and subscription. To register the data type, create a new `TypeSupport` with a `TopicDataType` instance and use the `register_type()` member function on the `TypeSupport`. Then the *Topic* can be created with the registered type name.

Note: Registering two different data types on the same *DomainParticipant* with identical names is not allowed and will issue an error. However, it is allowed to register the same data type within the same *DomainParticipant*, with the same or different names. If the same data type is registered twice on the same *DomainParticipant* with the same name, the second registering will have no effect, but will not issue any error.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Register the data type in the DomainParticipant.
// If nullptr is used as name argument, the one returned by the type itself is used
```

(continues on next page)

(continued from previous page)

```
TypeSupport custom_type_support(new CustomDataType());
custom_type_support.register_type(participant, nullptr);

// The previous instruction is equivalent to the following one
// Even if we are registering the same data type with the same name twice, no error␣
↪will be issued
custom_type_support.register_type(participant, custom_type_support.get_type_name());

// Create a Topic with the registered type.
Topic* topic =
        participant->create_topic("topic_name", custom_type_support.get_type_name(),␣
↪TOPIC_QOS_DEFAULT);
if (nullptr != topic)
{
    // Error
    return;
}

// Create an alias for the same data type using a different name.
custom_type_support.register_type(participant, "data_type_name");

// We can now use the aliased name to If no name is given, it uses the name returned␣
↪by the type itself
Topic* another_topic =
        participant->create_topic("other_topic_name", "data_type_name", TOPIC_QOS_
↪DEFAULT);
if (nullptr != another_topic)
{
    // Error
    return;
}
```

Dynamic data types

Instead of directly writing the specialized `TopicDataType` class, it is possible to dynamically define data types following the OMG Extensible and Dynamic Topic Types for DDS interface. Data types can also be described on an XML file that is dynamically loaded.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Load the XML file with the type description
eprosima::fastrtps::xmlparser::XMLProfileManager::loadXMLFile("example_type.xml");

// Retrieve the an instance of the desired type and register it
eprosima::fastrtps::types::DynamicType_ptr dyn_type =
        eprosima::fastrtps::xmlparser::XMLProfileManager::getDynamicTypeByName(
↪"DynamicType")->build();
```

(continues on next page)

(continued from previous page)

```
TypeSupport dyn_type_support(new eprosima::fastrtps::types::DynamicPubSubType(dyn_
↪type));
dyn_type_support.register_type(participant, nullptr);

// Create a Topic with the registered type.
Topic* topic =
        participant->create_topic("topic_name", dyn_type_support.get_type_name(),␣
↪TOPIC_QOS_DEFAULT);
if (nullptr != topic)
{
    // Error
    return;
}
```

A complete description of the dynamic definition of types can be found on the *Dynamic Topic Types* section.

Data types with a key

Data types that define a set of fields to form a unique key can distinguish different data sets within the same data type. This mechanism is implemented overriding the `getKey()` member function on the TopicDataType to return the appropriate key value according to the field data. Types that do not define a key will not override this member function.

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);
```

(continues on next page)

(continued from previous page)

```
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);
if (nullptr != topic_with_profile_and_custom_listener)
{
    // Error
    return;
}
```

Deleting a Topic

A *Topic* can be deleted with the `delete_topic()` member function on the *DomainParticipant* instance where the *Topic* was created.

```
// Create a DomainParticipant in the desired domain
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(0, PARTICIPANT_
↪QOS_DEFAULT);
if (nullptr != participant)
{
    // Error
    return;
}

// Create a Topic
Topic* topic =
        participant->create_topic("TopicName", "DataTypeName", TOPIC_QOS_DEFAULT);
if (nullptr != topic)
{
    // Error
    return;
}

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

// Delete the Topic
if (participant->delete_topic(topic) != ReturnCode_t::RETCODE_OK)
{
    // Error
    return;
}
```

6.7 RTPS Layer

The lower level RTPS Layer of *eprosima Fast DDS* serves an implementation of the protocol defined in the RTPS standard. This layer provides more control over the internals of the communication protocol than the *DDS Layer*, so advanced users have finer control over the library's functionalities.

6.7.1 Relation to the DDS Layer

Elements of this layer map one-to-one with elements from the *DDS Layer*, with a few additions. This correspondence is shown in the following table:

DDS Layer	*RTPS Layer*
Domain	RTPSDomain
DomainParticipant	RTPSParticipant
DataWriter	RTPSWriter
DataReader	RTPSReader

6.7.2 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 of how to use this layer the distribution comes with while reading this section. They are located in *examples/RTPSTest_as_socket* and in *examples/RTPSTest_registered*

Managing the Participant

Creating a `RTPSParticipant` is done with `RTPSDomain::createParticipant()`. `RTPSParticipantAttributes` structure is used to configure the `RTPSParticipant` upon creation.

```
RTPSParticipantAttributes participant_attr;
participant_attr.setName("participant");
RTPSParticipant* participant = RTPSDomain::createParticipant(0, participant_attr);
```

Managing the Writers and Readers

As the RTPS standard specifies, Writers and Readers 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);
```

The creation of a Reader is similar to that of the Writers. 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 History. Each piece of data is represented by a Change, which *eprosima Fast DDS* implements as `CacheChange_t`. Changes are always managed by the History. As a user, the procedure for interacting with the History is always the same:

1. Request a `CacheChange_t` from the History
2. Use it
3. Release it

You can interact with the History of the Writer to send data. A callback that returns the maximum number of payload bytes is required:

```
//Request a change from the history
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 back 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 a `ReaderListener` callback method as we did in the *DDS Layer*:

```
class MyReaderListener : public ReaderListener
{
public:

    MyReaderListener()
    {
    }

    ~MyReaderListener()
    {
    }

    void onNewCacheChangeAdded(
            RTPSReader* reader,
            const CacheChange_t* const change)
    {
        // The incoming message is enclosed within the `change` in the function␣
↪parameters
        printf("%s\n", change->serializedPayload.data);
        // Once done, remove the change
        reader->getHistory()->remove_change((CacheChange_t*)change);
    }

};
```

6.7.3 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 (see *Persistence*)

To choose your preferred option:

```
writer_attr.endpoint.durabilityKind = TRANSIENT_LOCAL;
```

Because in the *RTPS Layer* you have control over the History, in TRANSIENT_LOCAL and TRANSIENT modes the Writer sends all changes you have not explicitly released from the History.

6.7.4 Configuring the History

The History has its own configuration structure, the `HistoryAttributes`.

Changing the maximum size of the payload

You can choose the maximum size of the Payload that can go into a `CacheChange_t`. Be sure to choose a size that allows it to hold the biggest possible piece of data:

```
history_attr.payloadMaxSize  = 250;//Defaults to 500 bytes
```

Changing the size of the History

You can specify a maximum amount of changes for the History to hold and an initial amount of allocated changes:

```
history_attr.initialReservedCaches = 250; //Defaults to 500
history_attr.maximumReservedCaches = 500; //Defaults to 0 = Unlimited Changes
```

When the initial amount of reserved changes is lower than the maximum, the History will allocate more changes as they are needed until it reaches the maximum size.

6.8 Discovery

Fast DDS, as a Data Distribution Service (DDS) implementation, provides discovery mechanisms that allow for automatically finding and matching `DataWriters` and `DataReaders` across `DomainParticipants` so they can start sharing data. This discovery is performed, for all the mechanisms, in two phases.

6.8.1 Discovery phases

1. **Participant Discovery Phase (PDP)**: During this phase the `DomainParticipants` acknowledge each other's existence. To do that, each `DomainParticipant` sends periodic announcement messages, which specify, among other things, unicast addresses (IP and port) where the `DomainParticipant` is listening for incoming meta and user data traffic. Two given `DomainParticipants` will match when they exist in the same DDS Domain. By default, the announcement messages are sent using well-known multicast addresses and ports (calculated using the `DomainId`). Furthermore, it is possible to specify a list of addresses to send announcements using unicast (see in *Initial peers*). Moreover, is is also possible to configure the periodicity of such announcements (see *Discovery Configuration*).
2. **Endpoint Discovery Phase (EDP)**: During this phase, the `DataWriters` and `DataReaders` acknowledge each other. To do that, the `DomainParticipants` share information about their `DataWriters` and `DataReaders` with each other, using the communication channels established during the PDP. This information contains, among other things, the `Topic` and data type (see *Topic*). For two endpoints to match, their topic and data type must coincide. Once `DataWriter` and `DataReader` have matched, they are ready for sending/receiving user data traffic.

6.8.2 Discovery mechanisms

Fast DDS provides the following discovery mechanisms:

- *Simple Discovery*: This is the default mechanism. It upholds the RTPS standard for both PDP and EDP, and therefore provides compatibility with any other DDS and RTPS implementations.
- *Static Discovery*: This mechanisms uses the Simple Participant Discovery Protocol (SPDP) for the PDP phase (as specified by the RTPS standard), but allows for skipping the Simple Participant Discovery Protocol (SEDP) phase when all the `DataWriters`' and `DataReaders`' IPs and ports, data types, and `Topics` are known beforehand.
- *Server-Client Discovery*: This discovery mechanism uses a centralized discovery architecture, where a DomainParticipant, referred as Server, act as a hub for discovery meta traffic.
- **Manual Discovery**: This mechanism is only compatible with the RTPS layer. It disables the PDP, letting the user to manually match and unmatch `RTPSParticipants`, `RTPSReaders`, and `RTPSWriters` using whatever, external meta-information channel of its choice. Therefore, the user must access the `RTPSParticipant` implemented by the `DomainParticipant` and directly match the RTPS Entities.

6.8.3 Discovery settings

The following sections list and describe the settings available for each of the previously defined discovery mechanisms, as well as how to define the `DomainParticipantListener` discovery callbacks.

General Discovery Settings

Some discovery settings are shared across the different discovery mechanisms. These settings are defined under the `builtin` public data member of the `WireProtocolConfigQos` class. These are:

Name	Description	Type	Default
Discovery Protocol	The discovery protocol to use (see *Discovery mechanisms*).	`DiscoveryProtocol_t`	`SIMPLE`
Ignore Participant flags	Filter discovery traffic for `DomainParticipants` in the same process, in different processes, or in different hosts.	`ParticipantFilteringFlags_t`	`NO_FILTER`
Lease Duration	Indicates for how much time should a remote `DomainParticipant` consider the local `DomainParticipant` to be alive.	`Duration_t`	20 s
Announcement Period	The period for the `DomainParticipant` to send PDP announcements.	`Duration_t`	3 s

Discovery Protocol

Specifies the discovery protocol to use (see *Discovery mechanisms*). The possible values are:

Discovery Mechanism	Possible values	Description
Simple	`SIMPLE`	Simple discovery protocol as specified in RTPS standard.
Static	`STATIC`	SPDP with manual EDP specified in XML files.
Server-Client	`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 send its discovery information to the server, and receives all other discovery information from the server.
	`BACKUP`	Creates a SERVER `DomainParticipant` which has a persistent `sqlite` database. A BACKUP server can load the a database on start. This type of sever makes the Server-Client architecture resilient to server destruction.
Manual	`NONE`	Disables PDP phase, therefore the is no EDP phase. All matching must be done manually through the `addReaderLocator`, `addReaderProxy`, `addWriterProxy` RTPS layer methods.

C++

```
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        eprosima::fastrtps::rtps::DiscoveryProtocol_t::SIMPLE;
```

XML

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

Ignore Participant flags

Defines a filter to ignore some discovery traffic when received. This is useful to add an extra level of *DomainParticipant* isolation. The possible values are:

Possible values	Description
NO_FILTER	All Discovery traffic is processed.
FILTER_DIFFERENT_HOST	Discovery traffic from another host is discarded.
FILTER_DIFFERENT_PROCESS	Discovery traffic from another process on the same host is discarded.
FILTER_SAME_PROCESS	Discovery traffic from *DomainParticipant*'s own process is discarded.
FILTER_DIFFERENT_PROCESS FILTER_SAME_PROCESS	Discovery traffic from *DomainParticipant*'s own host is discarded.

C++

```
pqos.wire_protocol().builtin.discovery_config.ignoreParticipantFlags =
        static_cast<eprosima::fastrtps::rtps::ParticipantFilteringFlags_t>(
    eprosima::fastrtps::rtps::ParticipantFilteringFlags_t::FILTER_DIFFERENT_PROCESS␣
↪|
    eprosima::fastrtps::rtps::ParticipantFilteringFlags_t::FILTER_SAME_PROCESS);
```

XML

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

Lease Duration

Indicates for how much time should a remote *DomainParticipant* consider the local *DomainParticipant* to be alive. If the liveliness of the local *DomainParticipant* has not being asserted within this time, the remote *DomainParticipant* considers the local *DomainParticipant* dead and destroys all the information regarding the local *DomainParticipant* and all its endpoints.

The local *DomainParticipant*'s liveliness is asserted on the remote *DomainParticipant* any time the remote *DomainParticipant* receives any kind of traffic from the local *DomainParticipant*.

The lease duration is specified as a time expressed in seconds and nanosecond using a `Duration_t`.

C++

```
pqos.wire_protocol().builtin.discovery_config.leaseDuration = Duration_t(10, 20);
```

XML

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

Announcement Period

It specifies the periodicity of the `DomainParticipant`'s PDP announcements. For liveliness' sake it is recommend that the announcement period is shorter than the lease duration, so that the `DomainParticipant`'s liveliness is asserted even when there is no data traffic. It is important to note that there is a trade-off involved in the setting of the announcement period, i.e. too frequent announcements will bloat the network with meta traffic, but too scarce ones will delay the discovery of late joiners.

`DomainParticipant`'s announcement period is specified as a time expressed in seconds and nanosecond using a `Duration_t`.

C++

```
pqos.wire_protocol().builtin.discovery_config.leaseDuration_announcementperiod =␣
↪Duration_t(1, 2);
```

XML

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

SIMPLE Discovery Settings

The SIMPLE discovery protocol resolves the establishment of the end-to-end connection between various DDS Entities. Fast DDS implements the SIMPLE discovery protocol to provide compatibility with the RTPS standard. The specification splits up the SIMPLE discovery protocol into two independent protocols:

- **Simple Participant Discovery Protocol (SPDP):** specifies how `DomainParticipants` discover each other in the network; it announces and detects the presence of `DomainParticipants` within the same domain.
- **Simple Endpoint Discovery Protocol (SEDP):** defines the protocol adopted by the discovered `DomainParticipants` for the exchange of information in order to discover the DDS Entities contained in each of them, i.e. the `DataWriter` and `DataReader`.

Name	Description
Initial Announcements	It defines the behavior of the `DomainParticipants` initial announcements.
Simple EDP Attributes	It defines the use of the SIMPLE protocol as a discovery protocol.
Initial peers	A list of `DomainParticipant`'s IP/port pairs to which the SPDP announcements are sent.

Initial Announcements

RTPS standard simple discovery mechanism requires the *DomainParticipants* to send announcements of their presence in the domain. These announcements are not delivered in a reliable fashion, and can be disposed of by the network. In order to avoid the discovery delay induced by message disposal, the initial announcement can be set up to make several shots, in order to increase proper reception chances.

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 `initial_announcements.period`).

Name	Description	Type	Default
count	It defines the number of announcements to send at start-up.	`uint32`	5
period	It defines the specific period for initial announcements.	`Duration_t`	100ms

C++

```
pqos.wire_protocol().builtin.discovery_config.initial_announcements.count = 5;
pqos.wire_protocol().builtin.discovery_config.initial_announcements.period =
↪Duration_t(0, 100000000u);
```

XML

```
<participant profile_name="participant_profile_simple_discovery">
    <rtps>
        <builtin>
            <discovery_config>
                <initialAnnouncements>
                    <count>5</count>
                    <period>
                        <sec>0</sec>
                        <nanosec>100000000</nanosec>
                    </period>
                </initialAnnouncements>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Simple EDP Attributes

Name	Description	Type	Default
SIMPLE EDP	It defines the use of the SIMPLE protocol as a discovery protocol for EDP phase. A *DomainParticipant* may create *DataWriters*, *DataReaders*, both or neither.	bool	true
Publication writer and Subscription reader	It is intended for *DomainParticipants* that implement only one or more *DataWriters*, i.e. do not implement *DataReaders*. It allows the creation of only DataReader discovery related EDP endpoints.	bool	true
Publication reader and Subscription writer	It is intended for *DomainParticipants* that implement only one or more *DataReaders*, i.e. do not implement *DataWriters*. It allows the creation of only *DataWriter* discovery related EDP endpoints.	bool	true

C++

```
pqos.wire_protocol().builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol␣
↪= true;
pqos.wire_protocol().builtin.discovery_config.m_simpleEDP.use_
↪PublicationWriterANDSubscriptionReader = true;
pqos.wire_protocol().builtin.discovery_config.m_simpleEDP.use_
↪PublicationReaderANDSubscriptionWriter = false;
```

XML

```
<participant profile_name="participant_profile_qos_discovery_edp">
    <rtps>
        <builtin>
            <discovery_config>
                <EDP>SIMPLE</EDP>
                <simpleEDP>
                    <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>
                    <PUBREADER_SUBWRITER>false</PUBREADER_SUBWRITER>
                </simpleEDP>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Initial peers

According to the RTPS standard (Section 9.6.1.1), each `RTPSParticipant` must listen for incoming Participant Discovery Protocol (PDP) discovery metatraffic in two different ports, one linked with a multicast address, and another one linked to a unicast address. Fast DDS allows for the configuration of an initial peers list which contains one or more such IP-port address pairs corresponding to remote `DomainParticipants` PDP discovery listening resources, so that the local `DomainParticipant` will not only send its PDP traffic to the default multicast address-port specified by its domain, but also to all the IP-port address pairs specified in the initial peers list.

A `DomainParticipant`'s initial peers list contains the list of IP-port address pairs of all other `DomainParticipants` with which it will communicate. It is a list of addresses that a `DomainParticipant` will use in the unicast discovery mechanism, together or as an alternative to multicast discovery. Therefore, this approach also applies to those scenarios in which multicast functionality is not available.

According to the RTPS standard (Section 9.6.1.1), the RTPSParticipants' discovery traffic unicast listening ports are calculated using the following equation: 7400 + 250 * *domainID* + 10 + 2 * *participantID*. Thus, if for example a `RTPSParticipant` operates in Domain 0 (default domain) and its ID is 1, its discovery traffic unicast listening port would be: 7400 + 250 * 0 + 10 + 2 * 1 = 7412. By default *eProsima Fast DDS* uses as initial peers the Metatraffic Multicast Locators.

The following constitutes an example configuring an Initial Peers list with one peer on host 192.168.10.13 with `DomainParticipant` ID 1 in domain 0.

C++

```
eprosima::fastrtps::rtps::Locator_t initial_peers_locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(initial_peers_locator, "192.168.10.13
↪");
initial_peers_locator.port = 7412;

pqos.wire_protocol().builtin.initialPeersList.push_back(initial_peers_locator);
```

XML

```
<participant profile_name="initial_peers_example_profile" is_default_profile="true">
    <rtps>
        <builtin>
            <initialPeersList>
                <locator>
                    <udpv4>
                        <address>192.168.10.13</address>
                        <port>7412</port>
                    </udpv4>
                </locator>
            </initialPeersList>
        </builtin>
    </rtps>
</participant>
```

STATIC Discovery Settings

Fast DDS allows for the substitution of the SEDP protocol for the EDP phase with a static version that completely eliminates EDP meta traffic. This can become useful when dealing with limited network bandwidth and a well-known schema of `DataWriters` and `DataReaders`. If all `DataWriters` and `DataReaders`, and their `Topics` and data types, are known beforehand, the EDP phase can be replaced with a static configuration of peers. It is important to note that by doing this, no EDP discovery meta traffic will be generated, and only those peers defined in the configuration will be able to communicate. The STATIC discovery related settings are:

Name	Description
STATIC EDP	It activates the STATIC discovery protocol.
STATIC EDP XML Files Specification	Specifies an XML file containing a description of the remote `DataWriters` and `DataReaders`.
Initial Announcements	It defines the behavior of the `DomainParticipant` initial announcements (PDP phase).

STATIC EDP

To activate the STATIC EDP, the SEDP must be disabled on the `WireProtocolConfigQos`. This can be done either by code or using an XML configuration file:

C++

```
pqos.wire_protocol().builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol␣
↪= false;
pqos.wire_protocol().builtin.discovery_config.use_STATIC_EndpointDiscoveryProtocol␣
↪= true;
```

XML

```
<participant profile_name="participant_profile_static_edp">
    <rtps>
        <builtin>
            <discovery_config>
                <EDP>STATIC</EDP>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

STATIC EDP XML Files Specification

Since activating STATIC EDP suppresses all EDP meta traffic, the information about the remote entities (`DataWriters` and `DataReaders`) must be statically specified, which is done using dedicated XML files. A `DomainParticipant` may load several of such configuration files so that the information about different entities can be contained in one file, or split into different files to keep it more organized. Fast DDS provides a Static Discovery example that implements this EDP discovery protocol.

The following table describes all the possible elements of a STATIC EDP XML configuration file. A full example of such file can be found in *STATIC EDP XML Example*.

Name	Description	Values	Default
`<userId>`	Mandatory. Uniquely identifies the `DataReader`/`DataWriter`.	`uint16_t`	0
`<entityID>`	EntityId of the `DataReader`/`DataWriter`.	`uint16_t`	0
`<expectsInlineQos>`	Indicates if QOS is expected inline (`DataReader` **only**).	`bool`	`false`
`<topicName>`	Mandatory. The topic of the remote `DataReader`/`DataWriter`. Should match with one of the topics of the local `DataReaders`/`DataWriters`.	`string_255`	
`<topicDataType>`	Mandatory. The data type of the topic.	`string_255`	
`<topicKind>`	The kind of topic.	`NO_KEY` `WITH_KEY`	`NO_KEY`
`<partition>`	The name of a partition of the remote peer. Repeat to configure several partitions.	`string`	
`<unicastLocator>`	Unicast locator of the `DomainParticipant`. See *Locators definition*.		
`<multicastLocator>`	Multicast locator of the `DomainParticipant`. See *Locators definition*.		
`<reliabilityQos>`	See the *ReliabilityQosPolicy* section.	`BEST_EFFORT_RELIABILITY_QOS` `RELIABLE_RELIABILITY_QOS`	`BEST_EFFORT_RELIABILITY_QOS`

`<durabilityQos>`	See the *DurabilityQosPolicy* section.	`VOLATILE_DURABILITY_QOS`
		`TRANSIENT_LOCAL_DURABILITY_QOS`
		`TRANSIENT_DURABILITY_QOS`
			`VOLATILE_DURABILITY_QOS`

`<ownershipQos>`	See *Ownership QoS*.		
`<livelinessQos>`	Defines the liveliness of the remote peer. See *Liveliness QoS*.		

Locators definition

Locators for remote peers are configured using `<unicastLocator>` and `<multicastLocator>` tags. These take no value, and the locators are defined using tag elements. Locators defined with `<unicastLocator>` and `<multicastLocator>` are accumulative, so they can be repeated to assign several remote endpoints locators to the same peer.

- `address`: a mandatory `string` representing the locator address.
- `port`: an optional `uint16_t` representing a port on that address.

Ownership QoS

The ownership of the topic can be configured using `<ownershipQos>` tag. It takes no value, and the configuration is done using tag elements:

- `kind`: can be one of `SHARED_OWNERSHIP_QOS` or `EXCLUSIVE_OWNERSHIP_QOS`. This element is mandatory withing the tag.
- `strength`: an optional `uint32_t` specifying how strongly the remote `DomainParticipant` owns the `Topic`. This QoS can be set on `DataWriters` **only**. If not specified, default value is zero.

Liveliness QoS

The *LivelinessQosPolicy* of the remote peer is configured using `<livelinessQos>` tag. It takes no value, and the configuration is done using tag elements:

- `kind`: can be any of `AUTOMATIC_LIVELINESS_QOS`, `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS` or `MANUAL_BY_TOPIC_LIVELINESS_QOS`. This element is mandatory withing the tag.
- `leaseDuration_ms`: an optional `uint32` specifying the lease duration for the remote peer. The special value `INF` can be used to indicate infinite lease duration. If not specified, default value is `INF`

STATIC EDP XML Example

The following is a complete example of a configuration XML file for two remote `DomainParticipants`, 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

```xml
<staticdiscovery>
    <participant>
        <name>HelloWorldSubscriber</name>
        <reader>
            <userId>3</userId>
            <entityId>4</entityId>
            <expectsInlineQos>true</expectsInlineQos>
            <topicName>HelloWorldTopic</topicName>
            <topicDataType>HelloWorld</topicDataType>
            <topicKind>WITH_KEY</topicKind>
            <partitionQos>HelloPartition</partitionQos>
            <partitionQos>WorldPartition</partitionQos>
            <unicastLocator address="192.168.0.128" port="5000"/>
            <unicastLocator address="10.47.8.30" port="6000"/>
            <multicastLocator address="239.255.1.1" port="7000"/>
            <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
            <ownershipQos kind="SHARED_OWNERSHIP_QOS"/>
            <livelinessQos kind="AUTOMATIC_LIVELINESS_QOS" leaseDuration_ms="1000"/>
        </reader>
    </participant>
    <participant>
        <name>HelloWorldPublisher</name>
        <writer>
            <unicastLocator address="192.168.0.120" port="9000"/>
            <unicastLocator address="10.47.8.31" port="8000"/>
            <multicastLocator address="239.255.1.1" port="7000"/>
            <userId>5</userId>
            <entityId>6</entityId>
            <topicName>HelloWorldTopic</topicName>
            <topicDataType>HelloWorld</topicDataType>
            <topicKind>WITH_KEY</topicKind>
            <partitionQos>HelloPartition</partitionQos>
            <partitionQos>WorldPartition</partitionQos>
            <reliabilityQos>BEST_EFFORT_RELIABILITY_QOS</reliabilityQos>
            <durabilityQos>VOLATILE_DURABILITY_QOS</durabilityQos>
            <ownershipQos kind="SHARED_OWNERSHIP_QOS" strength="50"/>
            <livelinessQos kind="AUTOMATIC_LIVELINESS_QOS" leaseDuration_ms="1000"/>
        </writer>
    </participant>
</staticdiscovery>
```

Loading STATIC EDP XML Files

Statically discovered remote `DataReaders`/`DataWriters` **must** define a unique *userID* on their profile, whose value **must** agree with the one specified in the discovery configuration XML. This is done by setting the user ID on the `DataReaderQos`/`DataWriterQos`:

C++

```
// Configure the DataWriter
eprosima::fastdds::dds::DataWriterQos wqos;
wqos.endpoint().user_defined_id = 1;

// Configure the DataReader
eprosima::fastdds::dds::DataReaderQos rqos;
rqos.endpoint().user_defined_id = 3;
```

XML

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

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

On the local `DomainParticipant`, loading STATIC EDP configuration files is done by:

C++

```
pqos.wire_protocol().builtin.discovery_config.setStaticEndpointXMLFilename(
↪"RemotePublisher.xml");
pqos.wire_protocol().builtin.discovery_config.setStaticEndpointXMLFilename(
↪"RemoteSubscriber.xml");
```

XML

```
<participant profile_name="participant_profile_static_load_xml">
    <rtps>
        <builtin>
            <discovery_config>
                <staticEndpointXMLFilename>RemotePublisher.xml</
↪staticEndpointXMLFilename>
                <staticEndpointXMLFilename>RemoteSubscriber.xml</
↪staticEndpointXMLFilename>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Server-Client Discovery Settings

This mechanism is based on a client-server discovery paradigm, i.e. the metatraffic (message exchange among `DomainParticipants` to identify each other) is managed by one or several server `DomainParticipants` (left figure), as opposed to simple discovery (right figure), where metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol. A Discovery-Server tool is available to ease client-server setup and testing.

Fig. 7: Comparison of Server-Client discovery and Simple discovery mechanisms

Key concepts

In this architecture there are several key concepts to understand:

- The Server-Client discovery 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 meta-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*.
- *Clients* require a beforehand knowledge of the *servers* to which they want to link. Basically it is reduced to the *server* identity (henceforth called `GuidPrefix`) and a list of locators where the *server* is listening. These locators also define the transport protocol (UDP or TCP) the client will use to contact the *server*.
  - The `GuidPrefix` 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` 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.
- *Servers* do not require any beforehand knowledge of their *clients*, but their `GuidPrefix` and locator list (where they are listening) must match the one provided to the *clients*. In order to gather *client* discovery information, the following handshake strategy is followed:
  - *Clients* send discovery messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement.
  - *Servers* receive discovery messages from the clients, but they do not start processing them until a time interval has elapsed, which starts at the moment the server is instantiated.

In order to clarify this discovery setup, either on compile time (sources) or runtime (XML files), this explanation is divided into two sections: on focusing on the main concepts (*setup by concept*), and another one focusing on the main setting structures and XML tags (*setup by QoS*).

Server-Client setup by concept

Concept	Description
Discovery protocol	Make a participant a *client* or a *server*.
Server unique id	Link a *clients* to *servers*.
Seting up transport	Specify which transport to use and make *servers* reachable.
Pinging period	Fine tune server-client handshake.
Matching period	Fine tune server deliver efficiency.

Choosing between Client and Server

It is set by the *Discovery Protocol* general setting. A participant can only play a role (despite the fact that a *server* may act as a *client* of other server). It's mandatory to fill this value because it defaults to *simple*. The values associated with the Server-Client discovery are specified in *discovery settings section*. The examples below show how to manage the corresponding enum and XML tag.

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos;

pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        eprosima::fastrtps::rtps::DiscoveryProtocol_t::CLIENT;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        eprosima::fastrtps::rtps::DiscoveryProtocol_t::SERVER;
pqos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        eprosima::fastrtps::rtps::DiscoveryProtocol_t::BACKUP;
```

XML

```
<participant profile_name="participant_discovery_protocol_alt" >
  <rtps>
    <builtin>
        <discovery_config>
          <discoveryProtocol>CLIENT</discoveryProtocol>
          <!-- alternatives
          <discoveryProtocol>SERVER</discoveryProtocol>
          <discoveryProtocol>BACKUP</discoveryProtocol>
          -->
        </discovery_config>
    </builtin>
  </rtps>
</participant>
```

The GuidPrefix as the server unique identifier

The `GuidPrefix` 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` as a public data member of the *`WireProtocolConfigQos`* class. In the Server-Client discovery, it has the purpose to link a *server* to its *clients*. It must be mandatorily 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.

```
using namespace eprosima::fastrtps::rtps;

GuidPrefix_t serverGuidPrefix;
serverGuidPrefix.value[0] = octet(0x77);
serverGuidPrefix.value[1] = octet(0x73);
serverGuidPrefix.value[2] = octet(0x71);
serverGuidPrefix.value[3] = octet(0x85);
serverGuidPrefix.value[4] = octet(0x69);
serverGuidPrefix.value[5] = octet(0x76);
serverGuidPrefix.value[6] = octet(0x95);
serverGuidPrefix.value[7] = octet(0x66);
serverGuidPrefix.value[8] = octet(0x65);
serverGuidPrefix.value[9] = octet(0x82);
serverGuidPrefix.value[10] = octet(0x82);
serverGuidPrefix.value[11] = octet(0x79);

eprosima::fastdds::dds::DomainParticipantQos serverQos;
serverQos.wire_protocol().prefix = serverGuidPrefix;
```

C++ - Option 2: Using the >> operator and the `std::ostream` type.

```
eprosima::fastdds::dds::DomainParticipantQos serverQos;
std::istringstream("4d.49.47.55.45.4c.5f.42.41.52.52.4f") >> serverQos.wire_
↪protocol().prefix;
```

XML

```
<participant profile_name="participant_server_guidprefix" >
  <rtps>
        <prefix>
            4D.49.47.55.45.4c.5f.42.41.52.52.4f
        </prefix>
  </rtps>
</participant>
```

Note that a *server* can act as a *client* of other *servers*. Thus, the following section may also apply.

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` must be provided. The *server* list must be populated with `RemoteServerAttributes` objects with a valid `guidPrefix` data member. In XML the server list and its elements are simultaneously specified. Note that `prefix` is an element of the `RemoteServer` tag.

C++

```
eprosima::fastrtps::rtps::RemoteServerAttributes server;
server.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");

eprosima::fastdds::dds::DomainParticipantQos clientQos;
clientQos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(server);
```

XML

```
<participant profile_name="participant_profile_discovery_client_prefix">
    <rtps>
    <builtin>
        <discovery_config>
          <discoveryServersList>
            <RemoteServer prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f">
                  <metatrafficUnicastLocatorList>
                    <locator/>
                  </metatrafficUnicastLocatorList>
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
    </builtin>
  </rtps>
</participant>
```

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++

```
eprosima::fastrtps::rtps::Locator_t locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, 192, 168, 1, 133);
locator.port = 64863;

eprosima::fastdds::dds::DomainParticipantQos serverQos;
serverQos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
```

XML

```
<participant profile_name="participant_profile_discovery_server_server_metatraffic">
       <rtps>
        <builtin>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                    <!-- placeholder server UDP address -->
                        <address>192.168.1.113</address>
                        <port>64863</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>
```

Note that a *server* can act as a client of 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 and 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++

```
eprosima::fastrtps::rtps::Locator_t locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, 192, 168, 1, 133);
locator.port = 64863;
eprosima::fastrtps::rtps::RemoteServerAttributes server;
server.metatrafficUnicastLocatorList.push_back(locator);

eprosima::fastdds::dds::DomainParticipantQos clientQos;
clientQos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(server);
```

XML

```
<participant profile_name="participant_profile_discovery_server_client_metatraffic">
       <rtps>
        <builtin>
            <discovery_config>
                <discoveryServersList>
                    <RemoteServer prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f">
                        <metatrafficUnicastLocatorList>
                            <locator>
                                <udpv4>
                                <!-- placeholder server UDP address -->
                                    <address>192.168.1.113</address>
                                    <port>64863</port>
                                </udpv4>
                            </locator>
                        </metatrafficUnicastLocatorList>
                    </RemoteServer>
                </discoveryServersList>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

Client ping period

As explained *above* the *clients* send discovery messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement.

C++

```
eprosima::fastdds::dds::DomainParticipantQos clientQos;
clientQos.wire_protocol().builtin.discovery_config.discoveryServer_client_
↪syncperiod =
        Duration_t(0, 250000000);
```

XML

```
<participant profile_name="participant_profile_client_ping" >
  <rtps>
    <builtin>
        <discovery_config>
          <clientAnnouncementPeriod>
            <!-- change default to 250 ms -->
            <nanosec>250000000</nanosec>
          </clientAnnouncementPeriod>
        </discovery_config>
    </builtin>
  </rtps>
</participant>
```

Server match period

As explained *above*, the *servers* receive discovery messages from new clients to join the communication. However, the *servers* do not start processing them until a time interval, defined by this period, has elapsed, which starts at the moment the server is instantiated. Therefore, this member specifies a time interval in which the server's `DataReader` is disabled and incoming messages are not processed. It is a time interval intended to allow the server to initialize its resources.

C++

```
eprosima::fastdds::dds::DomainParticipantQos serverQos;
serverQos.wire_protocol().builtin.discovery_config.discoveryServer_client_
↪syncperiod =
        Duration_t(0, 250000000);
```

XML

```
<participant profile_name="participant_profile_server_ping" >
  <rtps>
    <builtin>
        <discovery_config>
          <clientAnnouncementPeriod>
            <!-- change default to 250 ms -->
            <nanosec>250000000</nanosec>
          </clientAnnouncementPeriod>
        </discovery_config>
    </builtin>
  </rtps>
</participant>
```

Server-Client setup by Qos

The settings related with server-client discovery are:

Name	Description
WireProtocolConfigQos (`wire_protocol()`)	Specifies wire protocol settings for a `DomainParticipant`. Some of it data members must be modified in order to properly configure a Server. An example is the `prefix` data member.
RTPS BuiltinAttributes (`builtin`)	It is a public data member of the above `WireProtocolConfigQos` class. Allows to specify some mandatory server discovery settings like the addresses were it listens for clients discovery information.
DiscoverySettings	It is a member of the above `BuiltinAttributes` structure. Allows to specify some mandatory and optional Server-Client discovery settings such as whether the `DomainParticipant` is a client or a server, the list of servers it is linked to, the client-ping, and the server-match frequencies.

WireProtocolConfigQos

The `prefix` data member of the `wire_protocol()` class specifies the server's identity. This member has only significance if `discovery_config.discoveryProtocol` is **SERVER** or **BACKUP**.

C++

```
using namespace eprosima::fastrtps::rtps;

GuidPrefix_t serverGuidPrefix;
serverGuidPrefix.value[0] = octet(0x77);
serverGuidPrefix.value[1] = octet(0x73);
serverGuidPrefix.value[2] = octet(0x71);
serverGuidPrefix.value[3] = octet(0x85);
serverGuidPrefix.value[4] = octet(0x69);
serverGuidPrefix.value[5] = octet(0x76);
serverGuidPrefix.value[6] = octet(0x95);
serverGuidPrefix.value[7] = octet(0x66);
serverGuidPrefix.value[8] = octet(0x65);
serverGuidPrefix.value[9] = octet(0x82);
serverGuidPrefix.value[10] = octet(0x82);
serverGuidPrefix.value[11] = octet(0x79);

eprosima::fastdds::dds::DomainParticipantQos serverQos;
serverQos.wire_protocol().prefix = serverGuidPrefix;
```

XML

```
<participant profile_name="participant_profile_discovery_client_prefix">
    <rtps>
    <builtin>
        <discovery_config>
          <discoveryServersList>
            <RemoteServer prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f">
                  <metatrafficUnicastLocatorList>
                    <locator/>
                  </metatrafficUnicastLocatorList>
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
    </builtin>
  </rtps>
</participant>
```

RTPS BuiltinAttributes

All discovery related information is gathered in the *DiscoverySettings* `discovery_config` data member.

In order to receive client metatraffic, `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList` must be populated with the addresses (IP and port) that were given to the clients.

C++

```
eprosima::fastrtps::rtps::Locator_t locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, 192, 168, 1, 133);
locator.port = 64863;

eprosima::fastdds::dds::DomainParticipantQos serverQos;
serverQos.wire_protocol().builtin.metatrafficUnicastLocatorList.push_back(locator);
```

XML

```
<participant profile_name="participant_profile_discovery_server_metatraffic">
       <rtps>
        <builtin>
            <discovery_config>
              <discoveryProtocol>SERVER</discoveryProtocol>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                    <!-- placeholder server UDP address -->
                        <address>192.168.1.113</address>
                        <port>64863</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>
```

DiscoverySettings

The `discoveryProtocol` enum data member (*Discovery Protocol*) specifies the participant's discovery kind. As was explained before, to setup the Server-Client discovery it may be:

enum value	Description
CLIENT	Generates a client `DomainParticipant`, which relies on a server (or servers) to be notified of other clients presence. This `DomainParticipant` can create `DataWriters` and `DataReaders` of any topic (static or dynamic) as ordinary `DomainParticipants` do.
SERVER	Generates a server `DomainParticipant`, which receives, manages and spreads its matched client's metatraffic assuring any single one is aware of the others. This `DomainParticipant` can create `DataWriters` and `DataReaders` of any topic (static or dynamic) as ordinary `DomainParticipants` do. Servers can link to other servers in order to share its clients information.
BACKUP	Generates a server `DomainParticipant` with additional functionality over **SERVER**. Specifically, it uses a database to backup its client information, so that this information can be automatically restored at any moment and continue spreading metatraffic to late joiners. A **SERVER** in the same scenario ought to collect client information again, introducing a recovery delay.

A `DiscoveryServers` that lists the servers linked to a client `DomainParticipant`. This member has only significance if *Discovery Protocol* is **CLIENT**, **SERVER** or **BACKUP**. These member elements are `RemoteServerAttributes` objects that identify each server and report where the servers can be reached:

Data members	Description
`GuidPrefix_t guidPrefix`	Is the RTPS unique identifier of the remote server `DomainParticipant`.
`metatrafficUnicastLocatorList` `metatrafficMulticastLocatorList`	Are ordinary `LocatorList_t` (see *LocatorListType*) where the server's locators must be specified. At least one of them should be populated.
`Duration_t discoveryServerPingPeriod`	Has only significance if *Discovery Protocol* is **CLIENT**, **SERVER** or **BACKUP**. For a *client* it specifies the pinging period as explained in *key concepts*. When a client has not yet established a reliable connection to a server it *pings* until the server notices him and establishes the connection. For a *server* it specifies the match period as explained in *key concepts*. When a *server* discovers new *clients* it only starts exchanging information with them at regular intervals as a mechanism to bundle discovery information and optimize delivery. The default value is half a second.

C++

```
eprosima::fastrtps::rtps::RemoteServerAttributes server;
server.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");

eprosima::fastrtps::rtps::Locator_t locator;
eprosima::fastrtps::rtps::IPLocator::setIPv4(locator, 192, 168, 1, 133);
locator.port = 64863;
server.metatrafficUnicastLocatorList.push_back(locator);

eprosima::fastdds::dds::DomainParticipantQos clientQos;
clientQos.wire_protocol().builtin.discovery_config.discoveryProtocol =
        eprosima::fastrtps::rtps::DiscoveryProtocol_t::CLIENT;
clientQos.wire_protocol().builtin.discovery_config.m_DiscoveryServers.push_
↪back(server);
clientQos.wire_protocol().builtin.discovery_config.discoveryServer_client_
↪syncperiod =
        Duration_t(0, 250000000);
```

XML

```
<participant profile_name="participant_profile_client" >
  <rtps>
    <builtin>
        <discovery_config>
          <discoveryProtocol>CLIENT</discoveryProtocol>
          <discoveryServersList>
            <RemoteServer prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f">
              <metatrafficUnicastLocatorList>
                <locator>
                  <udpv4>
                  <!-- placeholder server UDP address -->
                    <address>192.168.1.113</address>
                    <port>64863</port>
                  </udpv4>
                </locator>
              </metatrafficUnicastLocatorList>
            </RemoteServer>
          </discoveryServersList>
          <clientAnnouncementPeriod>
            <!-- change default to 250 ms -->
            <nanosec>250000000</nanosec>
          </clientAnnouncementPeriod>
        </discovery_config>
    </builtin>
  </rtps>
</participant>
```

DomainParticipantListener Discovery Callbacks

As stated in *DomainParticipantListener*, the `DomainParticipantListener` is an abstract class defining the callbacks that will be triggered in response to state changes on the DomainParticipant. Fast DDS defines four callbacks attached to events that may occur during discovery: `on_participant_discovery`, `on_subscriber_discovery`, `on_publisher_discovery`, `on_type_discovery`. Further information about the `DomainParticipantListener` is provided in the *DomainParticipantListener* section. The following is an example of the implementation of `DomainParticipantListener` discovery callbacks.

```
class DiscoveryDomainParticipantListener : public DomainParticipantListener
{
    /* Custom Callback on_participant_discovery */
    virtual void on_participant_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::ParticipantDiscoveryInfo&& info)
    {
        (void)participant;
        switch (info.status){
            case eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::DISCOVERED_
↪PARTICIPANT:
                /* Process the case when a new DomainParticipant was found in the␣
↪domain */
                std::cout << "New DomainParticipant '" << info.info.m_participantName
↪<<
                    "' with ID '" << info.info.m_guid.entityId << "' and GuidPrefix '
↪" <<
                    info.info.m_guid.guidPrefix << "' discovered." << std::endl;
                break;
            case eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::CHANGED_QOS_
↪PARTICIPANT:
                /* Process the case when a DomainParticipant changed its QOS */
                break;
            case eprosima::fastrtps::rtps::ParticipantDiscoveryInfo::REMOVED_
↪PARTICIPANT:
                /* Process the case when a DomainParticipant was removed from the␣
↪domain */
                std::cout << "New DomainParticipant '" << info.info.m_participantName
↪<<
                    "' with ID '" << info.info.m_guid.entityId << "' and GuidPrefix '
↪" <<
                    info.info.m_guid.guidPrefix << "' left the domain." << std::endl;
                break;
        }
    }

    /* Custom Callback on_subscriber_discovery */
    virtual void on_subscriber_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info)
    {
        (void)participant;
        switch (info.status){
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERED_READER:
                /* Process the case when a new subscriber was found in the domain */
                std::cout << "New DataReader subscribed to topic '" << info.info.
↪topicName() <<
                "' of type '" << info.info.typeName() << "' discovered";
```

(continues on next page)

(continued from previous page)

```
                break;
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::CHANGED_QOS_READER:
                /* Process the case when a subscriber changed its QOS */
                break;
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::REMOVED_READER:
                /* Process the case when a subscriber was removed from the domain */
                std::cout << "New DataReader subscribed to topic '" << info.info.
↪topicName() <<
                    "' of type '" << info.info.typeName() << "' left the domain.";
                break;
        }
    }

    /* Custom Callback on_publisher_discovery */
    virtual void on_publisher_discovery(
            DomainParticipant* participant,
            eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info)
    {
        (void)participant;
        switch (info.status){
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::DISCOVERED_WRITER:
                /* Process the case when a new publisher was found in the domain */
                std::cout << "New DataWriter publishing under topic '" << info.info.
↪topicName() <<
                    "' of type '" << info.info.typeName() << "' discovered";
                break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::CHANGED_QOS_WRITER:
                /* Process the case when a publisher changed its QOS */
                break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo::REMOVED_WRITER:
                /* Process the case when a publisher was removed from the domain */
                std::cout << "New DataWriter publishing under topic '" << info.info.
↪topicName() <<
                    "' of type '" << info.info.typeName() << "' left the domain.";
                break;
        }
    }

    /* Custom Callback on_type_discovery */
    virtual void on_type_discovery(
            DomainParticipant* participant,
            const eprosima::fastrtps::rtps::SampleIdentity& request_sample_id,
            const eprosima::fastrtps::string_255& topic,
            const eprosima::fastrtps::types::TypeIdentifier* identifier,
            const eprosima::fastrtps::types::TypeObject* object,
            eprosima::fastrtps::types::DynamicType_ptr dyn_type)
    {
        (void)participant, (void)request_sample_id, (void)topic, (void)identifier,␣
↪(void)object, (void)dyn_type;
        std::cout << "New data type of topic '" << topic << "' discovered." <<␣
↪std::endl;
    }
};
```

To use the previously implemented discovery callbacks in `DiscoveryDomainParticipantListener` class, which inherits from the *`DomainParticipantListener`*, an object of this class is created and registered as a listener of the *`DomainParticipant`*.

```
// Create a custom user DomainParticipantListener
DiscoveryDomainParticipantListener* plistener = new␣
↪DiscoveryDomainParticipantListener();
// Pass the listener on DomainParticipant creation.
DomainParticipant* participant =
        DomainParticipantFactory::get_instance()->create_participant(
                0, PARTICIPANT_QOS_DEFAULT, plistener);
```

6.9 Transports

eProsima Fast RTPS implements an architecture of pluggable transports. Current version implements five transports: UDPv4, UDPv6, TCPv4, TCPv6 and SHM (shared memory). By default, when a `Participant` is created, one built-in UDPv4 transport is configured.

You can add custom transports using the attribute `rtps.userTransports`.

C++

```
    //Create a descriptor for the new transport.
    auto custom_transport = std::make_shared<UDPv4TransportDescriptor>();
    custom_transport->sendBufferSize = 9216;
    custom_transport->receiveBufferSize = 9216;

    //Disable the built-in Transport Layer.
    participant_attr.rtps.useBuiltinTransports = false;

    //Link the Transport Layer to the Participant.
    participant_attr.rtps.userTransports.push_back(custom_transport);
```

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>
```

All Transport configuration options can be found in the section *Transport descriptors*.

6.9.1 Shared memory Transport (SHM)

The shared memory transport enables fast communications between entities running in the same processing unit/machine, relying on the shared memory mechanisms provided by the host operating system.

SHM transport provides better performance than other 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. 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.

When two participants on the same machine have SHM transport enabled, all communications between them are automatically performed by SHM transport only. The rest of the enabled transports are not used between those two participants.

To enable SHM transport in a Participant, you need to add the SharedMemTransportDescriptor to the `rtps.userTransports` attribute (C++ code) or define a transport_descriptor of type SHM in the XML file (see examples below).

C++

```
    // Create a descriptor for the new transport.
    std::shared_ptr<SharedMemTransportDescriptor> shm_transport = std::make_shared
↪<SharedMemTransportDescriptor>();

    // Link the Transport Layer to the Participant.
    participant_attr.rtps.userTransports.push_back(shm_transport);
```

XML

```
<transport_descriptors>
    <!-- Create a descriptor for the new transport -->
    <transport_descriptor>
        <transport_id>shm_transport</transport_id>
        <type>SHM</type>
    </transport_descriptor>
</transport_descriptors>

<participant profile_name="SHMParticipant">
    <rtps>
        <!-- Link the Transport Layer to the Participant -->
        <userTransports>
            <transport_id>shm_transport</transport_id>
        </userTransports>
    </rtps>
</participant>
```

SHM configuration parameters:

- `segment_size`: The size of the shared memory segment in bytes. A shared memory segment is created by each participant. Participant's writers copy their messages into the segment and send a message reference to the destination readers.
- `port_queue_capacity`: Each participant with SHM transport enabled listens on a queue (port) for incoming SHM message references. This parameter specifies the queue size (in messages).
- `healthy_check_timeout_ms`: With SHM, Readers and writers use a queue to exchange messages (called Port). If one of the processes involved crashes while using the port, the structure can be left inoperative. For this reason, every time a port is opened, a healthy check is performed. If the attached listeners don't respond in `healthy_check_timeout_ms` milliseconds, the port is destroyed and created again.
- `rtps_dump_file`: Full path, including the file name, of the protocol dump_file. When this string parameter is not empty, all the participant's SHM traffic (sent and received) is traced to a file. The output file format is *tcpdump* text hex, and can be read with protocol analyzer applications such as Wireshark.

6.9.2 TCP Transport

Unlike UDP, TCP transport is connection oriented and for that Fast-RTPS must establish a TCP connection before sending the RTPS messages. Therefore TCP transport can have two behaviors, acting as a server (**TCP Server**) or as a client (**TCP Client**). The server opens a TCP port listening for incoming connections and the client tries to connect to the server. The server and the client concepts are independent from the RTPS concepts: **Publisher**, **Subscriber**, **Writer**, and **Reader**. Any of them can operate as a **TCP Server** or a **TCP Client** because these entities are used only to establish the TCP connection and the RTPS protocol works over it.

To use TCP transports you need to define some more configurations:

You must create a new TCP transport descriptor, for example TCPv4. This transport descriptor has a field named `listening_ports` that indicates to Fast-RTPS in which physical TCP ports our participant will listen for input connections. If omitted, 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. The transport must be added to the `userTransports` list of the participant attributes. The field `wan_addr` can be used to allow incoming connections using the public IP in a WAN environment or the Internet. See *WAN or Internet Communication over TCP/IPv4* for more information about how to configure a TCP Transport to allow or connect to WAN connections.

C++

```
//Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<TCPv4TransportDescriptor>();
tcp_transport->add_listener_port(5100);
tcp_transport->set_WAN_address("80.80.99.45");

//Disable the built-in Transport Layer.
participant_attr.rtps.useBuiltinTransports = false;

//Link the Transport Layer to the Participant.
participant_attr.rtps.userTransports.push_back(tcp_transport);
```

XML

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>tcp_transport</transport_id>
        <type>TCPv4</type>
        <listening_ports>
            <port>5100</port>
        </listening_ports>
        <wan_addr>80.80.99.45</wan_addr>
    </transport_descriptor>
</transport_descriptors>

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

To configure the participant to connect to another node through TCP, you must configure and add a Locator to its `initialPeersList` that points to the remote *listening port*.

C++

```cpp
auto tcp2_transport = std::make_shared<TCPv4TransportDescriptor>();

//Disable the built-in Transport Layer.
participant_attr.rtps.useBuiltinTransports = false;

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

//Link the Transport Layer to the Participant.
participant_attr.rtps.userTransports.push_back(tcp2_transport);
```

XML

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

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

A TCP version of helloworld example can be found in this link.

WAN or Internet Communication over TCP/IPv4

Fast-RTPS 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 should add the rules to allow the communication.

For example, to allow incoming connections through our NAT, Fast-RTPS must be configured as a **TCP Server** listening to incoming TCP connections. To allow incoming connections through a WAN, the TCP descriptor associated must indicate its public IP through its field `wan_addr`.

C++

```
//Create a descriptor for the new transport.
auto tcp_transport = std::make_shared<TCPv4TransportDescriptor>();
tcp_transport->add_listener_port(5100);
tcp_transport->set_WAN_address("80.80.99.45");

//Disable the built-in Transport Layer.
participant_attr.rtps.useBuiltinTransports = false;

//Link the Transport Layer to the Participant.
participant_attr.rtps.userTransports.push_back(tcp_transport);
```

XML

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>tcp_transport</transport_id>
        <type>TCPv4</type>
        <listening_ports>
            <port>5100</port>
        </listening_ports>
        <wan_addr>80.80.99.45</wan_addr>
    </transport_descriptor>
</transport_descriptors>

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

In this case, configuring the router (which public IP is `80.80.99.45`) is mandatory to allow the incoming traffic to reach the **TCP Server**. Typically a NAT routing with the `listening_port 5100` to our machine is enough. Any existing firewall should be configured as well.

In the client side, it is needed to specify the public IP of the **TCP Server** with its `listening_port` as `initial_peer`.

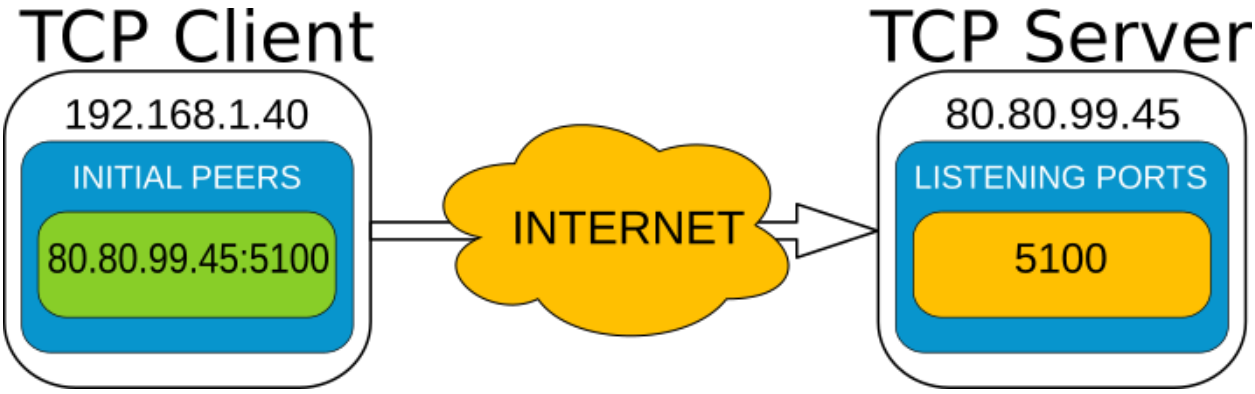
C++

```
auto tcp2_transport = std::make_shared<TCPv4TransportDescriptor>();

//Disable the built-in Transport Layer.
participant_attr.rtps.useBuiltinTransports = false;

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

//Link the Transport Layer to the Participant.
participant_attr.rtps.userTransports.push_back(tcp2_transport);
```

XML

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

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

The combination of the above configurations in both **TCP Server** and **TCP Client** allows a scenario similar to the represented by the following image.



IPLocator

IPLocator is an auxiliary static class that offers methods to ease the management of IP based locators, as UDP or TCP. In TCP, the port field of the locator is divided into physical and logical port. The physical port is the port used by the network device, the real port that the operating system understands. The logical port can be seen as RTPS port, or UDP's equivalent port (physical ports of UDP, are logical ports in TCP). Logical ports normally are not necessary to manage explicitly, but you can do it through IPLocator class. Physical ports instead, must be set to explicitly use certain ports, to allow the communication through a NAT, for example.

```
Locator_t locator;
// Get & Set Physical Port
uint16_t physical_port = IPLocator::getPhysicalPort(locator);
IPLocator::setPhysicalPort(locator, 5555);

// Get & Set Logical Port
uint16_t logical_port = IPLocator::getLogicalPort(locator);
IPLocator::setLogicalPort(locator, 7400);

// Set WAN Address
IPLocator::setWan(locator, "80.88.75.55");
```

NOTE

TCP doesn't support multicast scenarios, so you must plan carefully your network architecture.

TLS over TCP

Fast-RTPS allows configuring a TCP Transport to use TLS (Transport Layer Security) by setting up **TCP Server** and **TCP Client** properly.

TCP Server

C++

```
auto tls_transport = std::make_shared<TCPv4TransportDescriptor>();

using TLSOptions = 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);
```

XML

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>tls_transport_server</transport_id>
        <type>TCPv4</type>
        <tls>
            <password>test</password>
            <private_key_file>serverkey.pem</private_key_file>
            <cert_chain_file>server.pem</cert_chain_file>
            <tmp_dh_file>dh2048.pem</tmp_dh_file>
            <options>
                <option>DEFAULT_WORKAROUNDS</option>
                <option>SINGLE_DH_USE</option>
                <option>NO_SSLV2</option>
            </options>
        </tls>
    </transport_descriptor>
</transport_descriptors>
```

TCP Client

C++

```
        auto tls_transport = std::make_shared<TCPv4TransportDescriptor>();

        using TLSOptions = TCPTransportDescriptor::TLSConfig::TLSOptions;
        using TLSVerifyMode = TCPTransportDescriptor::TLSConfig::TLSVerifyMode;
        tls_transport->apply_security = true;
        tls_transport->tls_config.verify_file = "ca.pem";
        tls_transport->tls_config.verify_mode = TLSVerifyMode::VERIFY_PEER;
        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);
```

XML

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

More TLS related options can be found in the section *Transport descriptors*.

6.9.3 Listening locators

eProsima Fast RTPS divides listening locators into four categories:

- Metatraffic Multicast Locators: these locators are used to receive metatraffic information using multicast. They usually are used by built-in endpoints, like the discovery of built-in endpoints. You can set your own locators using attribute `rtps.builtin.metatrafficMulticastLocatorList`.

```
        // 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;

        participant_attr.rtps.builtin.metatrafficMulticastLocatorList.push_
↪back(locator);
```

- Metatraffic Unicast Locators: these locators are used to receive metatraffic information using unicast. They usually are used by built-in endpoints, like the discovery of built-in endpoints. You can set your own locators using attribute `rtps.builtin.metatrafficUnicastLocatorList`.

```
        // This locator will open a socket to listen network messages on UDPv4␣
↪port 22223 over network interface 192.168.0.1
        eprosima::fastrtps::rtps::Locator_t locator;
        IPLocator::setIPv4(locator, 192, 168, 0, 1);
        locator.port = 22223;

        participant_attr.rtps.builtin.metatrafficUnicastLocatorList.push_
↪back(locator);
```

- User Multicast Locators: these locators are used to receive user information using multicast. They are used by user endpoints. You can set your own locators using attribute `rtps.defaultMulticastLocatorList`.

```
        // 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;

        participant_attr.rtps.defaultMulticastLocatorList.push_back(locator);
```

- User Unicast Locators: these locators are used to receive user information using unicast. They are used by user endpoints. You can set your own locators using attributes `rtps.defaultUnicastLocatorList`.

```
        // This locator will open a socket to listen network messages on UDPv4␣
↪port 22225 over network interface 192.168.0.1
        eprosima::fastrtps::rtps::Locator_t locator;
        IPLocator::setIPv4(locator, 192, 168, 0, 1);
        locator.port = 22225;

        participant_attr.rtps.defaultUnicastLocatorList.push_back(locator);
```

By default *eProsima Fast RTPS* calculates the listening locators for the built-in UDPv4 network transport using well-known ports. These well-known ports are calculated using the following predefined rules:

Table 1: Ports used

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*

These predefined rules use some values explained here:

- DG: DomainId Gain. You can set this value using attribute `rtps.port.domainIDGain`.
- PG: ParticipantId Gain. You can set this value using attribute `rtps.port.participantIDGain`. The default value is `2`.
- PB: Port Base number. You can set this value using attribute `rtps.port.portBase`. The default value is `7400`.
- offsetd0, offsetd1, offsetd2, offsetd3: Additional offsets. You can set these values using attributes `rtps.port.offsetdN`. Default values are: `offsetd0 = 0`, `offsetd1 = 10`, `offsetd2 = 1`, `offsetd3 = 11`.

Both UDP and TCP unicast locators support to have a null address. In that case, *eProsima Fast RTPS* understands to get local network addresses and use them.

Both UDP and TCP locators support to have a zero port. In that case, *eProsima Fast RTPS* understands to calculate well-known port for that type of traffic.

6.9.4 Initial peers

According to the RTPS standard (Section 9.6.1.1), each participant must listen for incoming Participant Discovery Protocol (PDP) discovery metatraffic in two different ports, one linked with a multicast address, and another one linked to a unicast address (see *Discovery*). Fast-RTPS allows for the configuration of an initial peers list which contains one or more such address-port pairs corresponding to remote participants PDP discovery listening resources, so that the local participant will not only send its PDP traffic to the default multicast address-port specified by its domain, but also to all the address-port pairs specified in the initial-peers list.

A participant's initial peers list contains the list of address-port pairs of all other participants with which it will communicate. It is a list of addresses that a participant 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 participants' discovery traffic unicast listening ports are calculated using the following equation: 7400 + 250 * *domainID* + 10 + 2 * *participantID*. Thus, if for example a participant 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 RTPS* 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 participant ID 1 in domain 0.

C++

```
Locator_t initial_peers_locator;
IPLocator::setIPv4(initial_peers_locator, "192.168.10.13");
initial_peers_locator.port = 7412;
participant_attr.rtps.builtin.initialPeersList.push_back(initial_peers_locator);
```

XML

```
<participant profile_name="initial_peers_example_profile" is_default_profile="true">
    <rtps>
        <builtin>
            <initialPeersList>
                <locator>
                    <udpv4>
                        <address>192.168.10.13</address>
                        <port>7412</port>
                    </udpv4>
                </locator>
            </initialPeersList>
        </builtin>
    </rtps>
</participant>
```

6.9.5 Whitelist Interfaces

There could be situations where you want to block some network interfaces to avoid connections or sending data through them. This can be managed using the field *interface whitelist* in the transport descriptors, and with them, you can set the interfaces you want to use to send or receive packets. The values on this list should match the IPs of your machine in that networks. For example:

C++

```
UDPv4TransportDescriptor descriptor;
descriptor.interfaceWhiteList.emplace_back("127.0.0.1");
```

XML

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>CustomTransport</transport_id>
        <type>UDPv4</type>
        <interfaceWhiteList>
            <address>127.0.0.1</address>
        </interfaceWhiteList>
    </transport_descriptor>
<transport_descriptor>
```

6.9.6 Tips

Disabling all multicast traffic

C++

```
    // Metatraffic Multicast Locator List will be empty.
    // Metatraffic Unicast Locator List will contain one locator, with null address␣
↪and null port.
    // Then eProsima Fast RTPS will use all network interfaces to receive network␣
↪messages using a well-known port.
    Locator_t default_unicast_locator;
    participant_attr.rtps.builtin.metatrafficUnicastLocatorList.push_back(default_
↪unicast_locator);

    // Initial peer will be UDPv4 addresss 192.168.0.1. The port will be a well-
↪known port.
    // Initial discovery network messages will be sent to this UDPv4 address.
    Locator_t initial_peer;
    IPLocator::setIPv4(initial_peer, 192, 168, 0, 1);
    participant_attr.rtps.builtin.initialPeersList.push_back(initial_peer);
```

XML

```
<participant profile_name="disable_multicast" is_default_profile="true">
    <rtps>
        <builtin>
            <metatrafficUnicastLocatorList>
                <locator/>
            </metatrafficUnicastLocatorList>
            <initialPeersList>
                <locator>
                    <udpv4>
                        <address>192.168.0.1</address>
                    </udpv4>
                </locator>
            </initialPeersList>
        </builtin>
    </rtps>
</participant>
```

Non-blocking write on sockets

For UDP transport, it is possible to configure whether to use non-blocking write calls on the sockets.

C++

```
//Create a descriptor for the new transport.
auto non_blocking_UDP_transport = std::make_shared<UDPv4TransportDescriptor>();
non_blocking_UDP_transport->non_blocking_send = false;

//Disable the built-in Transport Layer.
participant_attr.rtps.useBuiltinTransports = false;

//Link the Transport Layer to the Participant.
participant_attr.rtps.userTransports.push_back(non_blocking_UDP_transport);
```

XML

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>non_blocking_transport</transport_id>
        <type>UDPv4</type>
        <non_blocking_send>false</non_blocking_send>
    </transport_descriptor>
</transport_descriptors>

<participant profile_name="non_blocking_transport">
    <rtps>
        <userTransports>
            <transport_id>non_blocking_transport</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
    </rtps>
</participant>
```

XML Configuration

The *XML profiles* section contains the full information about how to setup *Fast RTPS* through an *XML file*.

6.10 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 *Installation from Sources*.

Security plugins can be activated through the `DomainParticipantQos` properties. A `Property` is defined by its name (`std::string`) and its value (`std::string`).

Warning: For the full understanding of this documentation it is required the user to have basic knowledge of network security since terms like Certificate Authority (CA), Public Key Infrastructure (PKI), and Diffie-Hellman encryption protocol are not explained in detail. However, it is possible to configure basic system security settings, i.e. authentication, access control and encryption, to Fast DDS without this knowledge.

The following sections describe how to configure each of these properties to set up the Fast DDS security plugins.

6.10.1 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` `Property` `dds.sec.auth.plugin` with the value `builtin.PKI-DH`. The following table outlines the properties used for the DDS:Auth:PKI-DH plugin configuration.

Property name	Property value
identity_ca	URI to the X.509 v3 certificate of the Identity CA in PEM format. Supported URI schemes: file.
identity_certificate	URI to an X.509 v3 certificate signed by the Identity CA in PEM format containing the signed public key for the Participant. Supported URI schemes: file.
identity_crl *(optional)*	URI to a X.509 Certificate Revocation List (CRL). Supported URI schemes: file.
private_key	URI to access the private Private Key for the Participant. Supported URI schemes: file.
password *(optional)*	A password used to decrypt the *private_key*. If the *password* property is not present, then the value supplied in the *private_key* property must contain the decrypted private key.

Note: All listed properties have "dds.sec.auth.builtin.PKI-DH." prefix. For example: `dds.sec.auth.builtin.PKI-DH.identity_ca`.

The following is an example of how to set the properties of *DomainParticipantQos* for the DDS:Auth:PKI-DH plugin configuration.

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos;

// Activate DDS:Auth:PKI-DH plugin
pqos.properties().properties().emplace_back("dds.sec.auth.plugin",
    "builtin.PKI-DH");

// Configure DDS:Auth:PKI-DH plugin
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_ca",
    "file://maincacert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_certificate",
    "file://partcert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.identity_crl",
    "file://crl.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.private_key",
    "file://partkey.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.auth.builtin.PKI-DH.password",
    "domainParticipantPassword");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_auth_plugin_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.plugin</name>
                    <value>builtin.PKI-DH</value>
                </property>
                <!-- Configure DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_ca</name>
                    <value>file://maincacert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_certificate</name>
                    <value>file://partcert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_crl</name>
                    <value>file://crl.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.private_key</name>
                    <value>file://partkey.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.password</name>
                    <value>domainParticipantPassword</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Generation of X.509 certificates

An X.509 digital certificate is a document that has been encrypted and/or digitally signed according to RFC 5280. The X.509 certificate refers to the Public Key Infrastructure (PKI) certificate of the IETF , and specifies the standard formats for public-key certificates and a certification route validation algorithm. A simple way to generate these certificates for a proprietary PKI structure is through the OpenSSL toolkit. This section explains how to build a certificate infrastructure from the trusted CA certificate to the end-entity certificate, i.e. the `DomainParticipant`.

Generating the CA certificate for self-signing

First, since multiple certificates will need to be issued, one for each of the `DomainParticipants`, a dedicated CA is set up, and the CA's certificate is installed as the root key of all `DomainParticipants`. Thus, the `DomainParticipants` will accept all certificates issued by our own CA. To create a proprietary CA certificate, a configuration file must first be written with the CA information. An example of the CA configuration file is shown below. The OpenSSL commands shown in this example are compatible with both Linux and Windows Operating Systems (OS). However, all other commands are only compatible with Linux OS.

```
# File: maincaconf.cnf
# OpenSSL example Certificate Authority configuration file

####################################################################
[ ca ]
default_ca = CA_default # The default ca section

####################################################################
[ CA_default ]

dir = . # Where everything is kept
certs = $dir/certs # Where the issued certs are kept
crl_dir = $dir/crl # Where the issued crl are kept
database = $dir/index.txt # database index file.
unique_subject = no # Set to 'no' to allow creation of
                    # several ctificates with same subject.
new_certs_dir = $dir

certificate = $dir/maincacert.pem # The CA certificate
serial = $dir/serial # The current serial number
crlnumber = $dir/crlnumber # the current crl number
                           # must be commented out to leave a V1 CRL
crl = $dir/crl.pem # The current CRL
private_key = $dir/maincakey.pem # The private key
RANDFILE = $dir/private/.rand # private random number file

name_opt = ca_default # Subject Name options
cert_opt = ca_default # Certificate field options

default_days= 1825 # how long to certify for
default_crl_days = 30 # how long before next CRL
default_md = sha256 # which md to use.
preserve = no # keep passed DN ordering

policy = policy_match

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

(continues on next page)

(continued from previous page)

```
stateOrProvinceName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

[ req ]
prompt = no
#default_bits = 1024
#default_keyfile = privkey.pem
distinguished_name= req_distinguished_name
#attributes = req_attributes
#x509_extensions = v3_ca # The extentions to add to the self signed cert
string_mask = utf8only

[ req_distinguished_name ]
countryName = ES
stateOrProvinceName = MA
localityName = Tres Cantos
0.organizationName = eProsima
commonName = eProsima Main Test CA
emailAddress = mainca@eprosima.com
```

After writing the configuration file, next commands generate the certificate using the Elliptic Curve Digital Signature Algorithm (ECDSA).

```
openssl ecparam -name prime256v1 > ecdsaparam

openssl req -nodes -x509 \
  -days 3650 \
  -newkey ec:ecdsaparam \
  -keyout maincakey.pem \
  -out maincacert.pem \
  -config maincaconf.cnf
```

Generating the DomainParticipant certificate

As was done for the CA, a `DomainParticipant` certificate configuration file needs to be created first.

```
# File: partconf.cnf

prompt = no
string_mask = utf8only
distinguished_name = req_distinguished_name

[ req_distinguished_name ]
countryName = ES
stateOrProvinceName = MA
localityName = Tres Cantos
organizationName = eProsima
emailAddress = example@eprosima.com
commonName = DomainParticipantName
```

After writing the `DomainParticipant` certificate configuration file, next commands generate the X.509 certificate, using ECDSA, for a `DomainParticipant`.

```
openssl ecparam -name prime256v1 > ecdsaparam

openssl req -nodes -new \
  -newkey ec:ecdsaparam \
  -config partconf.cnf \
  -keyout partkey.pem \
  -out partreq.pem

openssl ca -batch -create_serial \
  -config maincaconf.cnf \
  -days 3650 \
  -in partreq.pem \
  -out partcert.pem
```

Generating the Certificate Revocation List (CRL)

Finally, the CRL is created. This is a list of the X.509 certificates revoked by the certificate issuing CA before they reach their expiration date. Any certificate that is on this list will no longer be trusted. To create a CRL using OpenSSL just run the following commands.

```
echo -ne '00' > crlnumber

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

As an example, below is shown how to add the X.509 certificate of a `DomainParticipant` to the CRL.

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

(continues on next page)

(continued from previous page)

```
  -revoke partcert.pem

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

6.10.2 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` `Property` `dds.sec.auth.plugin` with the value `builtin.Access-Permissions`. The following table outlines the properties used for the DDS:Access:Permissions plugin configuration.

Property name	Property value
permissions_ca	URI to the X509 certificate of the Permissions CA. Supported URI schemes: file. The file schema shall refer to an X.509 v3 certificate in PEM format.
governance	URI to shared Governance Document signed by the Permissions CA in S/MIME format. Supported URI schemes: file.
permissions	URI to the Participant permissions document signed by the Permissions CA in S/MIME format. Supported URI schemes: file.

Note: All listed properties have "dds.sec.access.builtin.Access-Permissions." prefix. For example: `dds.sec.access.builtin.Access-Permissions.permissions_ca`.

The following is an example of how to set the properties of `DomainParticipantQos` for the DDS:Access:Permissions configuration.

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos;

// Activate DDS:Access:Permissions plugin
pqos.properties().properties().emplace_back("dds.sec.access.plugin",
    "builtin.Access-Permissions");

// Configure DDS:Access:Permissions plugin
pqos.properties().properties().emplace_back(
    "dds.sec.access.builtin.Access-Permissions.permissions_ca",
    "file://certs/maincacert.pem");
pqos.properties().properties().emplace_back(
    "dds.sec.access.builtin.Access-Permissions.governance",
    "file://certs/governance.smime");
pqos.properties().properties().emplace_back(
    "dds.sec.access.builtin.Access-Permissions.permissions",
    "file://certs/permissions.smime");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_access_control_plugin_xml_
↪profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.plugin</name>
                    <value>builtin.Access-Permissions</value>
                </property>
                <!-- Configure DDS:Access:Permissions plugin -->
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.permissions_ca</
↪name>
                    <value>file://maincacet.pem</value>
                </property>
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.governance</
↪name>
                    <value>file://governance.smime</value>
                </property>
                <property>
                    <name>dds.sec.access.builtin.Access-Permissions.permissions</
↪name>
                    <value>file://permissions.smime</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Permissions CA Certificate

This is an X.509 certificate that contains the Public Key of the CA that will be used to sign the *Domain Governance Document* and the *DomainParticipant Permissions Document*.

Domain Governance Document

Domain Governance document is an XML document that specifies the mechanisms to secure the DDS Domain. It shall be signed by the Permissions CA in S/MIME format. The XML scheme of this document is defined in *Domain Governance XSD*. The following is an example of the Domain Governance XML file contents.

```
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="omg_shared_ca_domain_governance.xsd">
    <domain_access_rules>
        <domain_rule>
            <domains>
                <id_range>
                    <min>0</min>
                    <max>230</max>
                </id_range>
            </domains>
            <allow_unauthenticated_participants>false</allow_unauthenticated_
↪participants>
            <enable_join_access_control>true</enable_join_access_control>
            <discovery_protection_kind>ENCRYPT</discovery_protection_kind>
            <liveliness_protection_kind>ENCRYPT</liveliness_protection_kind>
            <rtps_protection_kind>ENCRYPT</rtps_protection_kind>
            <topic_access_rules>
                <topic_rule>
                    <topic_expression>HelloWorldTopic</topic_expression>
                    <enable_discovery_protection>true</enable_discovery_protection>
                    <enable_liveliness_protection>false</enable_liveliness_protection>
                    <enable_read_access_control>true</enable_read_access_control>
                    <enable_write_access_control>true</enable_write_access_control>
                    <metadata_protection_kind>ENCRYPT</metadata_protection_kind>
                    <data_protection_kind>ENCRYPT</data_protection_kind>
                </topic_rule>
            </topic_access_rules>
        </domain_rule>
    </domain_access_rules>
</dds>
```

The Governance XSD file and the Governance XML example can also be downloaded from the eProsima Fast DDS Github repository.

Domain Rules

It allows the application of rules to the DDS Domain. The domain rules define aspects of the DDS Domain such as:

- Whether the discovery data should be protected and the type of protection: MAC only or encryption followed by MAC.
- Whether the whole RTPS message should be encrypted.
- Whether the liveliness of the messages should be protected.

- Whether a non-authenticated `DomainParticipant` can access or not to the unprotected discovery metatraffic and unprotected Topics.
- Whether an authenticated `DomainParticipant` can access the domain without evaluating the access control policies.
- Whether discovery information on a certain Topic should be sent with secure `DataWriters`.
- Whether or not the access to Topics should be restricted to `DomainParticipants` with the appropriate permission to read them.
- Whether the metadata sent on a certain Topic should be protected and the type of protection.
- Whether payload data on a certain Topic should be protected and the type of protection.

The domain rules are evaluated in the same order as they appear in the document. A rule only applies to a particular `DomainParticipant` if the domain section matches the DDS `Domain_Id` to which the `DomainParticipant` belongs. If multiple rules match, the first rule that matches is the only one that applies. Each domain rule is delimited by the `<domain_rule>` XML element tag.

Some domain rules may have an additional configuration if enabled. This configuration defines the level of protection that the rule applies to the domain:

- **NONE**: no cryptographic transformation is applied.
- **SIGN**: cryptographic transformation based on Message Authentication Code (MAC) is applied, without additional encryption.
- **ENCRYPT**: the data is encrypted and followed by a MAC computed on the ciphertext, also known as Encrypt-then-MAC.

The following table summarizes the elements and sections that each domain rule may contain.

Type	Name	XML element tag	Values
Element	*Domains*	`<domains>`	`false`
			`true`
	Allow Unauthenticated Participants	`<allow_unauthenticated_participants>`	`false`
			`true`
	Enable Join Access Control	`<enable_join_access_control>`	`SIGN`
			`ENCRYPT`
			`NONE`
	Discovery Protection Kind	`<discovery_protection_kind>`	`SIGN`
			`ENCRYPT`
			`NONE`
	Liveliness Protection Kind	`<liveliness_protection_kind>`	`SIGN`
			`ENCRYPT`
			`NONE`
	RTPS Protection Kind	`<rtps_protection_kind>`	`SIGN`
			`ENCRYPT`
			`NONE`
Section	Topic Access Rules	`<topic_access_rules>`	`<topic_rule>`

The following describes the possible configurations of each of the elements and sections listed above that are contained in the domain rules.

Domains

This element is delimited by the `<domains>` XML element tag. The value in this element identifies the collection of DDS Domains to which the rule applies. The `<domains>` element can contain:

- A single domain identifier:

```
<domains>
    <id>1</id>
</domains>
```

- A range of domain identifiers:

```
<domains>
    <id_range>
        <min>1</min>
        <max>10</max>
    </id_range>
</domains>
```

Or a combination of both, a list of domain identifiers and ranges of domain identifiers.

Allow Unauthenticated Participants

This element is delimited by the `<allow_unauthenticated_participants>` XML element tag. It indicates whether the matching of a *`DomainParticipant`* with a remote *`DomainParticipant`* requires authentication. The possible values for this element are:

- `false`: the *`DomainParticipant`* shall enforce the authentication of remote *`DomainParticipants`* and disallow matching those that cannot be successfully authenticated.
- `true`: the *`DomainParticipant`* shall allow matching other *`DomainParticipants`* (event if the remote *`DomainParticipant`* cannot authenticate) as long as there is not an already valid authentication with the same *`DomainParticipant`*'s GUID.

Enable Join Access Control

This element is delimited by the `<enable_join_access_control>` XML element tag. Indicates whether the matching of the participant with a remote DomqainParticipant requires authorization by the DDS:Access:Permissions plugin. Its possible values are:

- `false`: the *`DomainParticipant`* shall not check the permissions of the authenticated remote *`DomainParticipant`*.
- `true`: the *`DomainParticipant`* shall check the permissions of the authenticated remote *`DomainParticipant`*.

Discovery Protection Kind

This element is delimited by the `<discovery_protection_kind>` XML element tag. Indicates whether the secure channel of the endpoint discovery phase needs to be encrypted. The possible values are:

- `NONE`: the secure channel shall not be protected.
- `SIGN`: the secure channel shall be protected by MAC.
- `ENCRYPT`: the secure channel shall be encrypted.

Liveliness Protection Kind

This element is delimited by the `<liveliness_protection_kind>` XML element tag. Indicates whether the secure channel of the liveliness mechanism needs to be encrypted. The possible values are:

- `NONE`: the secure channel shall not be protected.
- `SIGN`: the secure channel shall be protected by MAC.
- `ENCRYPT`: the secure channel shall be encrypted.

RTPS Protection Kind

This element is delimited by the `<rtps_protection_kind>` XML element tag. Indicates whether the whole RTPS Message needs to be encrypted. The possible values are:

- `NONE`: whole RTPS Messages shall not be protected.
- `SIGN`: whole RTPS Messages shall be protected by MAC.
- `ENCRYPT`: whole RTPS Messages shall be encrypted.

Topic Rule

This element is delimited by the `<topic_rule>` XML element tag and appears within the Topic Access Rules Section whose XML element tag is `<topic_access_rules>`. The following table summarizes the elements and sections that each domain rule may contain.

Elements	XML element tag	Values
Topic expression	`<topic_expression>`	Topic name
Enable Discovery Protection	`<enable_discovery_protection>`	`false`
		`true`
Enable Liveliness Protection	`<enable_liveliness_protection>`	`false`
		`true`
Enable Read Access Control	`<enable_read_access_control>`	`false`
		`true`
Enable Write Access Control	`<enable_write_access_control>`	`false`
		`true`
Metadata protection Kind	`<metadata_protection_kind>`	`true`
		`false`
Data protection Kind	`<data_protection_kind>`	`true`
		`false`

The topic expression within the rules selects a set of Topic names. The rule applies to any *DataReader* or *DataWriter* associated with a *Topic* whose name matches the *Topic* expression name. The topic access rules are evaluated in the same order as they appear within the `<topic_access_rules>` section. If multiple rules match, the first rule that matches is the only one that applies.

Topic expression

This element is delimited by the `<topic_expression>` XML element tag. The value in this element identifies the set of Topic names to which the rule applies. The rule applies to any *DataReader* or *DataWriter* associated with a *Topic* whose name matches the value.

The Topic name expression syntax and matching shall use the syntax and rules of the POSIX `fnmatch()` function as specified in IEEE 1003.1-2017.

Enable Discovery Protection

This element is delimited by the `<enable_discovery_protection>` XML element tag. Indicates whether the entity related discovery information shall go through the secure channel of endpoint discovery phase.

- `false`: the entity discovery information shall be sent by an unsecured channel of discovery.
- `true`: the information shall be sent by the secure channel.

Enable Liveliness Protection

This element is delimited by the `<enable_liveliness_protection>` XML element tag. Indicates whether the entity related liveliness information shall go through the secure channel of liveliness mechanism.

- `false`: the entity liveliness information shall be sent by an unsecured channel of liveliness.
- `true`: the information shall be sent by the secure channel.

Enable Read Access Control

This element is delimited by the `<enable_read_access_control>` XML element tag. Indicates whether read access to the Topic is protected.

- `false`: then local Subscriber creation and remote Subscriber matching can proceed without further access-control mechanisms imposed.
- `true`: they shall be checked using Access control plugin.

Enable Write Access Control

This element is delimited by the `<enable_write_access_control>` XML element tag. Indicates whether write access to the Topic is protected.

- `false`: then local Publisher creation and remote Publisher matching can proceed without further access-control mechanisms imposed.
- `true`: they shall be checked using Access control plugin.

Metadata Protection Kind

This element is delimited by the `<metadata_protection_kind>` XML element tag. Indicates whether the entity's RTPS submessages shall be encrypted by the Cryptographic plugin.

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

Data Protection Kind

This element is delimited by the `<data_protection_kind>` XML element tag. Indicates whether the data payload shall be encrypted by the Cryptographic plugin.

- `false`: the data payload shall not be encrypted.
- `true`: the data payload shall be encrypted.

Domain Governance XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="dds" type="DomainAccessRulesNode" />
    <xs:complexType name="DomainAccessRulesNode">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="domain_access_rules"
                type="DomainAccessRules" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DomainAccessRules">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="domain_rule" type="DomainRule" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DomainRule">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="domains" type="DomainIdSet" />
            <xs:element name="allow_unauthenticated_participants"
                type="xs:boolean" />
            <xs:element name="enable_join_access_control"
                type="xs:boolean" />
            <xs:element name="discovery_protection_kind"
                type="ProtectionKind" />
            <xs:element name="liveliness_protection_kind"
                type="ProtectionKind" />
            <xs:element name="rtps_protection_kind"
                type="ProtectionKind" />
            <xs:element name="topic_access_rules"
                type="TopicAccessRules" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DomainIdSet">
        <xs:choice minOccurs="1" maxOccurs="unbounded">
            <xs:element name="id" type="DomainId" />
```

(continues on next page)

(continued from previous page)

```
            <xs:element name="id_range" type="DomainIdRange" />
        </xs:choice>
    </xs:complexType>
    <xs:simpleType name="DomainId">
        <xs:restriction base="xs:nonNegativeInteger" />
    </xs:simpleType>
    <xs:complexType name="DomainIdRange">
        <xs:choice>
            <xs:sequence>
                <xs:element name="min" type="DomainId" />
                <xs:element name="max" type="DomainId" minOccurs="0" />
            </xs:sequence>
            <xs:element name="max" type="DomainId" />
        </xs:choice>
    </xs:complexType>
    <xs:simpleType name="ProtectionKind">
        <xs:restriction base="xs:string">
            <xs:enumeration value="ENCRYPT_WITH_ORIGIN_AUTHENTICATION" />
            <xs:enumeration value="SIGN_WITH_ORIGIN_AUTHENTICATION" />
            <xs:enumeration value="ENCRYPT" />
            <xs:enumeration value="SIGN" />
            <xs:enumeration value="NONE" />
        </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="BasicProtectionKind">
        <xs:restriction base="ProtectionKind">
            <xs:enumeration value="ENCRYPT" />
            <xs:enumeration value="SIGN" />
            <xs:enumeration value="NONE" />
        </xs:restriction>
    </xs:simpleType>
    <xs:complexType name="TopicAccessRules">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="topic_rule" type="TopicRule" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TopicRule">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="topic_expression" type="TopicExpression" />
            <xs:element name="enable_discovery_protection"
                type="xs:boolean" />
            <xs:element name="enable_liveliness_protection"
                type="xs:boolean" />
            <xs:element name="enable_read_access_control"
                type="xs:boolean" />
            <xs:element name="enable_write_access_control"
                type="xs:boolean" />
            <xs:element name="metadata_protection_kind"
                type="ProtectionKind" />
            <xs:element name="data_protection_kind"
                type="BasicProtectionKind" />
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="TopicExpression">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
</xs:schema>
```

Back to the *Domain Governance Document*.

DomainParticipant Permissions Document

The permissions document is an XML file which contains the permissions of a `DomainParticipant` and binds them to the `DomainParticipant` distinguished name defined in the DDS:Auth:PKI-DH plugin. The permissions document shall be signed by the Permissions CA in S/MIME format. The XML scheme of this document is defined in *DomainParticipant Permissions XSD*. The following is an example of the `DomainParticipant` Permissions XML file contents.

```
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://www.omg.org/spec/DDS-Security/20170801/omg_shared_ca_permissions.xsd">
    <permissions>
        <grant name="PublisherPermissions">
            <subject_name>emailAddress=mainpub@eprosima.com, CN=Main Publisher, OU=eProsima, O=eProsima, ST=MA, C=ES</subject_name>
            <validity>
                <not_before>2013-06-01T13:00:00</not_before>
                <not_after>2038-06-01T13:00:00</not_after>
            </validity>
            <allow_rule>
                <domains>
                    <id_range>
                        <min>0</min>
                        <max>230</max>
                    </id_range>
                </domains>
                <publish>
                    <topics>
                        <topic>HelloWorldTopic</topic>
                    </topics>
                </publish>
            </allow_rule>
            <default>DENY</default>
        </grant>
        <grant name="SubscriberPermissions">
            <subject_name> emailAddress=mainsub@eprosima.com, CN=Main Subscriber, OU=eProsima, O=eProsima, ST=MA, C=ES</subject_name>
            <validity>
                <not_before>2013-06-01T13:00:00</not_before>
                <not_after>2038-06-01T13:00:00</not_after>
            </validity>
            <allow_rule>
                <domains>
                    <id_range>
                        <min>0</min>
                        <max>230</max>
                    </id_range>
                </domains>
                <subscribe>
                    <topics>
                        <topic>HelloWorldTopic</topic>
                    </topics>
                </subscribe>
            </allow_rule>
            <default>DENY</default>
```

(continues on next page)

(continued from previous page)

```
        </grant>
    </permissions>
</dds>
```

The Permissions XSD file and the Permissions XML example can also be downloaded from the eProsima Fast DDS Github repository.

Grant Section

This section is delimited by the `<grant>` XML element tag. Each grant section contains three sections:

- Subject name
- Validity
- Rules

Subject name

This section is delimited by XML element `<subject_name>`. The subject name identifies the *`DomainParticipant`* to which the permissions apply. Each subject name can only appear in a single `<permissions>` section within the XML Permissions document. The contents of the subject name element shall be the X.509 subject name of the *`DomainParticipant`* that was given in the authorization X.509 Certificate.

Validity

This section is delimited by the XML element `<validity>`. It reflects the valid dates for the permissions.

Rules

This section contains the permissions assigned to the *`DomainParticipant`*. The rules are applied in the same order that appears in the document. If the criteria for the rule matched the Domain join, publish or subscribe operation that is being attempted, then the *allow* or *deny* decision is applied. If the criteria for a rule does not match the operation being attempted, the evaluation shall proceed to the next rule. If all rules have been examined without a match, then the decision specified by the `<default>` rule is applied. The default rule, if present, must appear after all *allow* and *deny* rules. If the default rule is not present, the implied default decision is `DENY`.

For the grant to match there shall be a match of the topics and partitions criteria.

Allow rules are delimited by the XML element `<allow_rule>`. Deny rules are delimited by the XML element``<deny_rule>``. Both contain the same element children.

Domains Section

This section is delimited by the XML element `<domains>`. The value in this element identifies the collection of DDS Domains to which the rule applies. The syntax is the same as for the *Domains* of the *Domain Governance Document*.

Format of the Allowed/Denied Actions sections

The sections for each of the three actions have a similar format. The only difference is the name of the XML element used to delimit the action:

Action	XML element tag
Allow/Deny Publish	`<publish>`
Allow/Deny Subscribe	`<subscribe>`
Allow/Deny Relay	`<relay>`

Each action contains two conditions.

- Allowed/Denied *Topics Condition*
- Allowed/Denied *Partitions Condition*

Topics Condition

This section is delimited by the `<topics>` XML element. It defines the Topic names that must be matched for the allow/deny rule to apply. Topic names may be given explicitly or by means of Topic name expressions. Each explicit topic name or Topic name expressions appears separately in a `<topic>` sub-element within the `<topics>` element.

The Topic name expression syntax and matching shall use the syntax and rules of the POSIX `fnmatch()` function as specified in

```
<topics>
    <topic>Plane</topic>
    <topic>Hel*</topic>
</topics>
```

Partitions Condition

This section is delimited by the `<partitions>` XML element. It limits the set Partitions names that may be associated with the (publish, subscribe, relay) action for the rule to apply. Partition names expression syntax and matching shall use the syntax and rules of the POSIX `fnmatch()` function as specified in IEEE 1003.1-2017. If there is no `<partitions>` section within a rule, then the default "empty string" partition is assumed.

```
<partitions>
    <partition>A</partition>
    <partition>B*</partition>
</partitions>
```

DomainParticipant Permissions XSD

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="dds" type="PermissionsNode" />
    <xs:complexType name="PermissionsNode">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="permissions" type="Permissions" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Permissions">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="grant" type="Grant" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Grant">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="subject_name" type="xs:string" />
            <xs:element name="validity" type="Validity" />
            <xs:sequence minOccurs="1" maxOccurs="unbounded">
                <xs:choice minOccurs="1" maxOccurs="1">
                    <xs:element name="allow_rule" minOccurs="0" type="Rule" />
                    <xs:element name="deny_rule" minOccurs="0" type="Rule" />
                </xs:choice>
            </xs:sequence>
            <xs:element name="default" type="DefaultAction" />
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required" />
    </xs:complexType>
    <xs:complexType name="Validity">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="not_before" type="xs:dateTime" />
            <xs:element name="not_after" type="xs:dateTime" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Rule">
        <xs:sequence minOccurs="1" maxOccurs="1">
            <xs:element name="domains" type="DomainIdSet" />
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="publish" type="Criteria" />
            </xs:sequence>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="subscribe" type="Criteria" />
            </xs:sequence>
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="relay" type="Criteria" />
            </xs:sequence>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="DomainIdSet">
        <xs:choice minOccurs="1" maxOccurs="unbounded">
            <xs:element name="id" type="DomainId" />
            <xs:element name="id_range" type="DomainIdRange" />
        </xs:choice>
    </xs:complexType>
    <xs:simpleType name="DomainId">
```

(continues on next page)

(continued from previous page)

```
        <xs:restriction base="xs:nonNegativeInteger" />
    </xs:simpleType>
    <xs:complexType name="DomainIdRange">
        <xs:choice>
            <xs:sequence>
                <xs:element name="min" type="DomainId" />
                <xs:element name="max" type="DomainId" minOccurs="0" />
            </xs:sequence>
            <xs:element name="max" type="DomainId" />
        </xs:choice>
    </xs:complexType>
    <xs:complexType name="Criteria">
        <xs:all minOccurs="1">
            <xs:element name="topics" minOccurs="1"
                type="TopicExpressionList" />
            <xs:element name="partitions" minOccurs="0"
                type="PartitionExpressionList" />
            <xs:element name="data_tags" minOccurs="0" type="DataTags" />
        </xs:all>
    </xs:complexType>
    <xs:complexType name="TopicExpressionList">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="topic" type="TopicExpression" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="PartitionExpressionList">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="partition" type="PartitionExpression" />
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="TopicExpression">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
    <xs:simpleType name="PartitionExpression">
        <xs:restriction base="xs:string" />
    </xs:simpleType>
    <xs:complexType name="DataTags">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="tag" type="TagNameValuePair" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="TagNameValuePair">
        <xs:sequence minOccurs="1" maxOccurs="unbounded">
            <xs:element name="name" type="xs:string" />
            <xs:element name="value" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
    <xs:simpleType name="DefaultAction">
        <xs:restriction base="xs:string">
            <xs:enumeration value="ALLOW" />
            <xs:enumeration value="DENY" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Back to the *DomainParticipant Permissions Document*.

Signing documents using x509 certificate

Domain Governance Document and *DomainParticipant Permissions Document* have to be signed using an X.509 certificate. Generation of an X.509 certificate is explained in *Generation of X.509 certificates*. Next commands sign the necessary documents for its use by the DDS:Access:Permissions plugin.

```
# Governance document: governance.xml
openssl smime -sign -in governance.xml -text -out governance.smime -signer maincacert.
↪pem -inkey maincakey.pem

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

6.10.3 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` `Property` `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* plugin. The DDS:Crypto:AES-GCM-GMAC plugin is configured using the *Access control plugin: DDS:Access:Permissions* plugin, i.e the cryptography plugin is configured through the properties and configuration files of the access control plugin. If the *Access control plugin: DDS:Access:Permissions* plugin will not be used, you can configure the DDS:Crypto:AES-GCM-GMAC plugin manually with the properties outlined in the following table.

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

The following is an example of how to set the properties of `DomainParticipantQos` for the DDS:Crypto:AES-GCM-GMAC configuration.

C++

```
eprosima::fastdds::dds::DomainParticipantQos pqos;

// Activate DDS:Crypto:AES-GCM-GMAC plugin
pqos.properties().properties().emplace_back("dds.sec.crypto.plugin",
    "builtin.AES-GCM-GMAC");

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

XML

```
<participant profile_name="secure_domainparticipant_conf_crypto_plugin_xml_profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Crypto:AES-GCM-GMAC plugin -->
                <property>
                    <name>dds.sec.crypto.plugin</name>
                    <value>builtin.AES-GCM-GMAC</value>
                </property>
                <!-- Only if DDS:Access:Permissions plugin is not enabled -->
                <!-- Configure DDS:Crypto:AES-GCM-GMAC plugin -->
                <property>
                    <name>rtps.participant.rtps_protection_kind</name>
                    <value>ENCRYPT</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

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

C++

```
eprosima::fastdds::dds::DataWriterQos wqos;

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

XML

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

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

C++

```
eprosima::fastdds::dds::DataWriterQos rqos;

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

XML

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

6.10.4 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` `Property` `dds.sec.log.plugin` with the value `builtin.DDS_LogTopic`. The following table outlines the properties used for the DDS:Logging:DDS_LogTopic plugin configuration.

Property name	Property value	
	Value	Definition
logging_level	`EMERGENCY_LEVEL`	System is unusable. Should not continue use.
	`ALERT_LEVEL`	Should be corrected immediately.
	`CRITICAL_LEVEL`	A failure in primary application.
	`ERROR_LEVEL`	General error conditions. Default value.
	`WARNING_LEVEL`	May indicate future error if action not taken.
	`NOTICE_LEVEL`	Unusual, but nor erroneous event or condition.
	`INFORMATIONAL_LEVEL`	Normal operational. Requires no action.
	`DEBUG_LEVEL`	Normal operational.
log_file	Path of the file in which the log messages are to be saved.	

Note: All listed properties have "dds.sec.log.builtin.DDS_LogTopic." prefix. For example: `dds.sec.log.builtin.DDS_LogTopic.logging_level`.

The following is an example of how to set the properties of *`DomainParticipantQos`* for the DDS:Logging:DDS_LogTopic plugin configuration.

C++

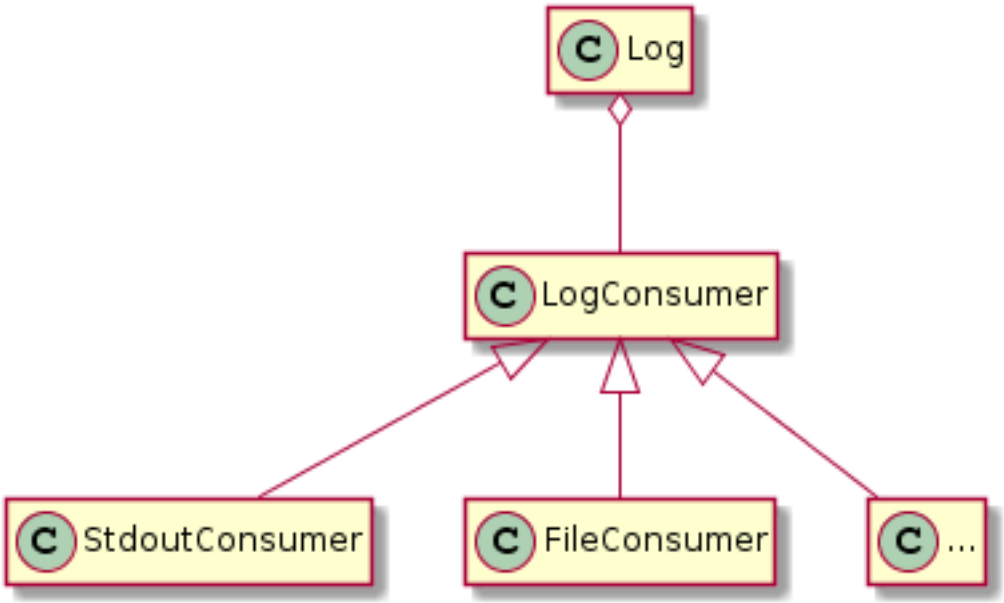
```
eprosima::fastdds::dds::DomainParticipantQos pqos;

// Activate DDS:Logging:DDS_LogTopic plugin
pqos.properties().properties().emplace_back("dds.sec.log.plugin",
    "builtin.DDS_LogTopic");

// Configure DDS:Logging:DDS_LogTopic plugin
pqos.properties().properties().emplace_back(
    "dds.sec.log.builtin.DDS_LogTopic.logging_level",
    "EMERGENCY_LEVEL");
pqos.properties().properties().emplace_back(
    "dds.sec.log.builtin.DDS_LogTopic.log_file",
    "myLogFile.log");
```

XML

```
<participant profile_name="secure_domainparticipant_conf_logging_plugin_xml_profile
↪">
    <rtps>
        <propertiesPolicy>
            <properties>
                <!-- Activate DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.log.plugin</name>
                    <value>builtin.DDS_LogTopic</value>
                </property>
                <!-- Configure DDS:Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.log.builtin.DDS_LogTopic.logging_level</name>
                    <value>EMERGENCY_LEVEL</value>
                </property>
                <property>
                    <name>dds.sec.log.builtin.DDS_LogTopic.log_file</name>
                    <value>myLogFile.log</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

6.11 Logging

Fast RTPS includes an extensible logging system with the following class hierarchy:



`Log` is the entry point of the Logging system. It exposes three macro definitions to ease its usage:

```
logInfo(INFO_MSG, "This is an info message");
logWarning(WARN_MSG, "This is a warning message");
logError(ERROR_MSG, "This is an error message");
```

In all cases, `INFO_MSG`, `WARN_MSG` and `ERROR_MSG` will be used as category for the log entry as a preprocessor string, so you can use define any category inline.

```
logInfo(NEW_CATEGORY, "This log message belong to NEW_CATEGORY category.");
```

You can control the verbosity of the log system and filter it by category:

```
Log::SetVerbosity(Log::Kind::Warning);
std::regex my_regex("NEW_CATEGORY");
Log::SetCategoryFilter(my_regex);
```

The possible verbosity levels are `Log::Kind::Info`, `Log::Kind::Warning` and `Log::Kind::Error`.

When selecting one of them, you also select the ones with more priority.

- Selecting `Log::Kind::Error`, you will only receive error messages.
- Selecting `Log::Kind::Warning` you select `Log::Kind::Error` too.
- Selecting `Log::Kind::Info` will select all of them

To filter by category, you must provide a valid `std::regex` expression that will be applied to the category. The categories that matches the expression, will be logged.

By default, the verbosity is set to `Log::Kind::Error` and without category filtering.

There are some others configurable parameters:

```
    //! Enables the reporting of filenames in log entries. Disabled by default.
    RTPS_DllAPI static void ReportFilenames(bool);
    //! Enables the reporting of function names in log entries. Enabled by default␣
↪when supported.
```

(continues on next page)

(continued from previous page)

```
        RTPS_DllAPI static void ReportFunctions(bool);
        //! Sets the verbosity level, allowing for messages equal or under that␣
↪priority to be logged.
        RTPS_DllAPI static void SetVerbosity(Log::Kind);
        //! Returns the current verbosity level.
        RTPS_DllAPI static Log::Kind GetVerbosity();
        //! Sets a filter that will pattern-match against log categories, dropping any␣
↪unmatched categories.
        RTPS_DllAPI static void SetCategoryFilter    (const std::regex&);
        //! Sets a filter that will pattern-match against filenames, dropping any␣
↪unmatched categories.
        RTPS_DllAPI static void SetFilenameFilter    (const std::regex&);
        //! Sets a filter that will pattern-match against the provided error string,␣
↪dropping any unmatched categories.
        RTPS_DllAPI static void SetErrorStringFilter (const std::regex&);
```

6.11.1 LogConsumers

LogConsumers are classes that implement how to manage the log information. They must be registered into the Log system to be called with the log messages (after filtering).

Currently there are two LogConsumer implementations:

- **`StdoutConsumer`:** Default consumer, it prints the logging messages to the standard output. It has no configuration available.
- **`FileConsumer`:** It prints the logging messages to a file. It has two configuration parameters:
  - `filename` that defines the file where the consumer will write the log messages.
  - `append` that indicates to the consumer if the output file must be opened to append new content.

  By default, `filename` is **output.log** and `append` is equals to **false**.

If you want to add a consumer to manage the logs, you must call the `RegisterConsumer` method of the Log. To remove all consumers, including the default one, you should call the `ClearConsumers` method. If you want to reset the Log configuration to its defaults, including recovering the default consumer, you can call to its `Reset` method.

```
    Log::ClearConsumers(); // Deactivate StdoutConsumer

    // Add FileConsumer consumer
    std::unique_ptr<FileConsumer> fileConsumer(new FileConsumer("append.log", true));
    Log::RegisterConsumer(std::move(fileConsumer));

    // Back to its defaults: StdoutConsumer will be enable and FileConsumer removed.
    Log::Reset();
```

6.11.2 XML Log configuration

You can configure the logging system through xml with the tag `<log>` under the `<dds>` tag, or as an standalone file (without the `<dds>` tag, just `<log>` as root). You can set `<use_default>` and a set of `<consumer>`. Each `<consumer>` is defined by its `<class>` and a set of `<property>`.

```
<log>
    <use_default>FALSE</use_default>
    <consumer>
        <class>FileConsumer</class>
        <property>
            <name>filename</name>
            <value>test1.log</value>
        </property>
        <property>
            <name>append</name>
            <value>TRUE</value>
        </property>
    </consumer>
</log>
```

`<use_default>` indicates if we want to use the default consumer `StdoutConsumer`.

Each `<consumer>` defines a consumer that will be added to the consumers list of the Log. `<class>` indicates which consumer class to instantiate and the set of `<property>` configures it. `StdoutConsumer` has no properties to be configured, but `FileConsumer` has `filename` and `append`.

This marks the end of this document. We recommend you to take a look at the Doxygen API reference and the embedded examples that come with the distribution. If you need more help, send us an email to *support@eprosima.com*.

6.12 XML profiles

This section shows how to configure entity attributes using XML profiles, explaining each field with its available values and how to compound the complete XML files.

eProsima Fast RTPS permits to load several XML files, each one containing XML profiles. In addition to the API functions to load user XML files, at initialization *eProsima Fast RTPS* tries to locate and load several default XML files. *eProsima Fast RTPS* offers the following options to use default XML files:

- Using an XML file with the name *DEFAULT_FASTRTPS_PROFILES.xml* and located in the current execution path.
- Using an XML file which location is defined in the environment variable *FASTRTPS_DEFAULT_PROFILES_FILE*.

An XML profile is defined by a unique name (or `<transport_id>` label in the *Transport descriptors* case) that is used to reference the XML profile during the creation of a Fast RTPS entity, *Transports*, or *Dynamic Topic Types*.

6.12.1 Making an XML

An XML file can contain several XML profiles. The available profile types are *Transport descriptors*, *XML Dynamic Types*, *Participant profiles*, *Publisher profiles*, and *Subscriber profiles*.

```
<transport_descriptor>
    <transport_id>TransportProfile</transport_id>
    <type>UDPv4</type>
    <!-- ... -->
</transport_descriptor>

<types>
    <type>
        <struct name="struct_profile">
            <!-- ... -->
        </struct>
    </type>
</types>

<participant profile_name="participant_profile">
    <rtps>
    <!-- ... -->
    </rtps>
</participant>

<publisher profile_name="publisher_profile">
    <!-- ... -->
</publisher>

<subscriber profile_name="subscriber_profile">
    <!-- ... -->
</subscriber>
```

The Fast-RTPS XML format uses some structures along several profiles types. For readability, the *Common* section groups these common structures.

Finally, The *Example* section shows an XML file that uses all the possibilities. This example is useful as a quick reference to look for a particular property and how to use it. This XSD file can be used as a quick reference too.

Loading and applying profiles

Before creating any entity, it's required to load XML files using `Domain::loadXMLProfilesFile` function. `createParticipant`, `createPublisher` and `createSubscriber` have a version that expects the profile name as an argument. *eProsima Fast RTPS* searches the XML profile using this profile name and applies the XML profile to the entity.

```
    eprosima::fastrtps::Domain::loadXMLProfilesFile("my_profiles.xml");

    Participant* participant = Domain::createParticipant("participant_xml_profile");
    Publisher* publisher = Domain::createPublisher(participant, "publisher_xml_profile
↪");
    Subscriber* subscriber = Domain::createSubscriber(participant, "subscriber_xml_
↪profile");
```

To load dynamic types from its declaration through XML see the *Usage* section of *XML Dynamic Types*.

6.12.2 Library settings

This section is devoted to general settings that are not constraint to specific entities (like participants, subscribers, publishers) or functionality (like transports or types). All of them are gathered under the `library_settings` profile.

```
    <library_settings>
        <intraprocess_delivery>FULL</intraprocess_delivery> <!-- OFF | USER_DATA_ONLY␣
↪| FULL -->
    </library_settings>
```

Currently only the *Intra-process delivery* feature is comprised here.

6.12.3 Transport descriptors

This section allows creating transport descriptors to be referenced by the *Participant profiles*. Once a well-defined transport descriptor is referenced by a **Participant profile**, every time that profile is instantiated it will use or create the related transport.

The following XML code shows the complete list of configurable parameters:

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>TransportId1</transport_id> <!-- string -->
        <type>UDPv4</type> <!-- string -->
        <sendBufferSize>8192</sendBufferSize> <!-- uint32 -->
        <receiveBufferSize>8192</receiveBufferSize> <!-- uint32 -->
        <TTL>250</TTL> <!-- uint8 -->
        <non_blocking_send>false</non_blocking_send> <!-- boolean -->
        <maxMessageSize>16384</maxMessageSize> <!-- uint32 -->
        <maxInitialPeersRange>100</maxInitialPeersRange> <!-- uint32 -->
        <interfaceWhiteList>
            <address>192.168.1.41</address> <!-- string -->
            <address>127.0.0.1</address> <!-- string -->
        </interfaceWhiteList>
        <wan_addr>80.80.55.44</wan_addr> <!-- string -->
        <output_port>5101</output_port> <!-- uint16 -->
        <keep_alive_frequency_ms>5000</keep_alive_frequency_ms> <!-- uint32 -->
        <keep_alive_timeout_ms>25000</keep_alive_timeout_ms> <!-- uint32 -->
        <max_logical_port>9000</max_logical_port> <!-- uint16 -->
        <logical_port_range>100</logical_port_range> <!-- uint16 -->
        <logical_port_increment>2</logical_port_increment> <!-- uint16 -->
        <listening_ports>
            <port>5100</port> <!-- uint16 -->
            <port>5200</port> <!-- uint16 -->
        </listening_ports>
        <calculate_crc>false</calculate_crc> <!-- boolean -->
        <check_crc>false</check_crc> <!-- boolean -->
        <enable_tcp_nodelay>false</enable_tcp_nodelay> <!-- boolean -->
        <tls><!-- TLS Section --></tls>
    </transport_descriptor>
```

The XML label `<transport_descriptors>` can hold any number of `<transport_descriptor>`.

Name	Description	Values	Default
`<transport_id>`	Unique name to identify each transport descriptor.	`string`	
`<type>`	Type of the transport descriptor.	`UDPv4`, `UDPv6`, `TCPv4`, `TCPv6`, `SHM`	`UDPv4`
`<sendBufferSize>`	Size in bytes of the socket send buffer. If the value is zero then FastRTPS will use the default size from the configuration of the sockets, using a minimum size of 65536 bytes.	`uint32`	0
`<receiveBufferSize>`	Size in bytes of the socket receive buffer. If the value is zero then FastRTPS will use the default size from the configuration of the sockets, using a minimum size of 65536 bytes.	`uint32`	0
`<TTL>`	*Time To Live*, **only** for UDP transports .	`uint8`	1
`<non_blocking>`	Whether to set the non-blocking send mode on the socket	`bool`	false
`<maxMessageSize>`	The maximum size in bytes of the transport's message buffer.	`uint32`	65500
`<maxInitialPeersRange>`	The maximum number of guessed initial peers to try to connect.	`uint32`	4
`<interfaceWhitelist>`	Allows defining *Whitelist Interfaces*.	*Whitelist Interfaces*	
`<wan_addr>`	Public WAN address when using **TCPv4 transports**. This field is optional if the transport doesn't need to define a WAN address.	`string` with IPv4 Format `XXX.XXX.XXX.XXX`.	
`<output_port>`	Port used for output bound. If this field isn't defined, the output port will be random (UDP **only**).	`uint16`	0
`<keep_alive_frequency_ms>`	Frequency in milliseconds for sending *RTCP* keep-alive requests (TCP **only**).	`uint32`	50000
`<keep_alive_timeout_ms>`	Time in milliseconds since sending the last keep-alive request to consider a connection as broken. (TCP **only**).	`uint32`	10000
`<max_logical_port>`	The maximum number of logical ports to try during *RTCP* negotiations. (TCP **only**)	`uint16`	100
`<logical_port_range>`	The maximum number of logical ports per request to try during *RTCP* negotiations. (TCP **only**)	`uint16`	20
`<logical_port_increment>`	Increment between logical ports to try during *RTCP* negotiation. (TCP **only**)	`uint16`	2
`<listening_ports>`	Local port to work as TCP acceptor for input connections. If not set, the transport will work as TCP client only (TCP **only**).	`List <uint16>`	
`<tls>`	Allows to define TLS related parameters and options (TCP **only**).	*TLS Configuration*	
`<segment_size>`	Size (in bytes) of the shared-memory segment. (OPTIONAL, SHM **only**).	`uint32`	262144
`<port_queue_capacity>`	Capacity (in number of messages) available to every Listener (OPTIONAL, SHM **only**).	`uint32`	512
`<healthy_check_timeout_ms>`	Maximum time-out (in milliseconds) used when checking whether a Listener is alive & OK (OPTIONAL, SHM **only**).	`uint32`	1000
`<rtps_dump_file>`	Complete path (including file) where RTPS messages will be stored for debugging purposes. An empty string indicates no trace will be performed (OPTIONAL, SHM **only**).	`string`	empty

RTCP is the control protocol for communications with RTPS over TCP/IP connections.

There are more examples of transports descriptors in *Transports*.

TLS Configuration

Fast-RTPS allows configuring TLS parameters through the `<tls>` tag of its Transport Descriptor. The full list of options is listed here:

```
<transport_descriptors>
    <transport_descriptor>
        <transport_id>Test</transport_id>
        <type>TCPv4</type>
        <tls>
            <password>Password</password>
            <private_key_file>Key_file.pem</private_key_file>
            <rsa_private_key_file>RSA_file.pem</rsa_private_key_file>
            <cert_chain_file>Chain.pem</cert_chain_file>
            <tmp_dh_file>DH.pem</tmp_dh_file>
            <verify_file>verify.pem</verify_file>
            <verify_mode>
                <verify>VERIFY_PEER</verify>
            </verify_mode>
            <options>
                <option>NO_TLSV1</option>
                <option>NO_TLSV1_1</option>
            </options>
            <verify_paths>
                <verify_path>Path1</verify_path>
                <verify_path>Path2</verify_path>
                <verify_path>Path3</verify_path>
            </verify_paths>
            <verify_depth>55</verify_depth>
            <default_verify_path>true</default_verify_path>
            <handshake_role>SERVER</handshake_role>
        </tls>
    </transport_descriptor>
</transport_descriptors>
```

Name	Description	Values	Default
`<password>`	Password of the private_key_file if provided (or RSA).	`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 CA (Certification- Authority) file.	`string`	
`<verify_mode>`	Establishes the verification mode mask.	`VERIFY_NONE`, `VERIFY_PEER`, `VERIFY_FAIL_IF_NO_PEER_CERT`, `VERIFY_CLIENT_ONCE`	
`<options>`	Establishes the SSL Context options mask	`DEFAULT_WORKAROUNDS`, `NO_COMPRESSION`, `NO_SSLV2`, `NO_SSLV3`, `NO_TLSV1`, `NO_TLSV1_1`, `NO_TLSV1_2`, `NO_TLSV1_3`, `SINGLE_DH_USE`	
`<verify_paths>`	Paths where the system will look for verification files.	`string`	
`<verify_depth>`	Maximum allowed depth for verify intermediate certificates.	`uint32`	
`<default_verify_paths>`	Default paths where the system will look for verification files.	`boolean`	`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`

6.12.4 XML Dynamic Types

XML Dynamic Types allows creating *eProsima Fast RTPS Dynamic Types* directly defining them through XML. It allows any application to change TopicDataTypes without modifying its source code.

XML Structure

The XML Types definition (`<types>` tag) can be placed similarly to the profiles tag inside the XML file. It can be a stand-alone XML Types file or be a child of the Fast-RTPS XML root tag (`<dds>`). Inside the types tag, there must be one or more type tags (`<type>`).

Stand-Alone:

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

Rooted:

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

Finally, each `<type>` tag can contain one or more *Type definitions*. Defining several types inside a `<type>` tag or defining each type in its `<type>` tag has the same result.

Type definition

Enum

The `<enum>` type is defined by its `name` and a set of `enumerators`, each of them with its `name` and its (optional) `value`.

Example:

XML	C++

`<enum name="MyEnum">`
`  <enumerator name="A" value="0"/>`
`  <enumerator name="B" value="1"/>`
`  <enumerator name="C" value="2"/>`
`</enum>`
	`  DynamicTypeBuilder_ptr enum_builder = DynamicTypeBuilderFactory::get_instance()->create_enum_builder();`
	`  enum_builder->set_name("MyEnum");`
	`  enum_builder->add_empty_member(0, "A");`
	`  enum_builder->add_empty_member(1, "B");`
	`  enum_builder->add_empty_member(2, "C");`
	`  DynamicType_ptr enum_type = DynamicTypeBuilderFactory::get_instance()->create_type(enum_builder.get());`

Typedef

The `<typedef>` type is defined by its `name` and its `value` or an inner element for complex types. `<typedef>` corresponds to `Alias` in Dynamic Types glossary.

Example:

XML	C++

`<typedef name="MyAliasEnum" type="nonBasic" nonBasicTypeName="MyEnum"/>`
 	
`<typedef name="MyAliasArray" type="int32" arrayDimension="2,2"/>`
	`DynamicTypeBuilder_ptr alias1_builder = DynamicTypeBuilderFactory::get_instance()->create_alias_builder(enum_builder.get(), "MyAlias1");`
	`DynamicType_ptr alias1_type = DynamicTypeBuilderFactory::get_instance()->create_type(alias1_builder.get());`
	
	`std::vector<uint32_t> sequence_lengths = { 2, 2 };`
	`DynamicTypeBuilder_ptr int_builder = DynamicTypeBuilderFactory::get_instance()->create_int32_builder();`
	`DynamicTypeBuilder_ptr array_builder = DynamicTypeBuilderFactory::get_instance()->create_array_builder(int_builder.get(), sequence_lengths);`
	`DynamicTypeBuilder_ptr alias2_builder = DynamicTypeBuilderFactory::get_instance()->create_alias_builder(array_builder.get(), "MyAlias2");`
	`DynamicType_ptr alias2_type = DynamicTypeBuilderFactory::get_instance()->create_type(alias2_builder.get());`

Struct

The `<struct>` type is defined by its `name` and inner *members*.

Example:

XML	C++

`<struct name="MyStruct">`
`  <member name="first" type="int32"/>`
`  <member name="second" type="int64"/>`
`</struct>`
	`  DynamicTypeBuilder_ptr long_builder = DynamicTypeBuilderFactory::get_instance()->create_int32_builder();`
	`  DynamicTypeBuilder_ptr long_long_builder = DynamicTypeBuilderFactory::get_instance()->create_int64_builder();`
	`  DynamicTypeBuilder_ptr struct_builder = DynamicTypeBuilderFactory::get_instance()->create_struct_builder();`
	
	`  struct_builder->set_name("MyStruct");`
	`  struct_builder->add_member(0, "first", long_builder.get());`
	`  struct_builder->add_member(1, "second", long_long_builder.get());`
	`  DynamicType_ptr struct_type = DynamicTypeBuilderFactory::get_instance()->create_type(struct_builder.get());`

Structs can inherit from another structs:

XML	C++

`<struct name="ParentStruct">`
`  <member name="first" type="int32"/>`
`  <member name="second" type="int64"/>`
`</struct>`
`<struct name="ChildStruct" baseType="ParentStruct">`
`  <member name="third" type="int32"/>`
`  <member name="fourth" type="int64"/>`
`</struct>`
	`  DynamicTypeBuilder_ptr long_builder = DynamicTypeBuilderFactory::get_instance()->create_int32_builder();`
	`  DynamicTypeBuilder_ptr long_long_builder = DynamicTypeBuilderFactory::get_instance()->create_int64_builder();`
	`  DynamicTypeBuilder_ptr struct_builder = DynamicTypeBuilderFactory::get_instance()->create_struct_builder();`
	
	`  struct_builder->set_name("ParentStruct");`
	`  struct_builder->add_member(0, "first", long_builder.get());`
	`  struct_builder->add_member(1, "second", long_long_builder.get());`
	`  DynamicType_ptr struct_type = DynamicTypeBuilderFactory::get_instance()->create_type(struct_builder.get());`
	
	`  DynamicTypeBuilder_ptr child_builder = DynamicTypeBuilderFactory::get_instance()->create_child_struct_builder(struct_builder.get());`
	
	`  child_builder->set_name("ChildStruct");`
	`  child_builder->add_member(0, "third", long_builder.get());`
	`  child_builder->add_member(1, "fourth", long_long_builder.get());`
	`  DynamicType_ptr child_struct_type = DynamicTypeBuilderFactory::get_instance()->create_type(child_builder.get());`

Union

The `<union>` type is defined by its `name`, a `discriminator` and a set of `cases`. Each `case` has one or more `caseDiscriminator` and a `member`.

Example:

XML	C++

`<union name="MyUnion">`
`  <discriminator type="byte"/>`
`  <case>`
`  <caseDiscriminator value="0"/>`
`  <caseDiscriminator value="1"/>`
`  <member name="first" type="int32 ↪"/>`
`  </case>`
`  <case>`
`  <caseDiscriminator value="2"/>`
`  <member name="second" type= ↪"nonBasic" nonBasicTypeName="MyStruct"/ ↪>`
`  </case>`
`  <case>`
`  <caseDiscriminator value="default ↪"/>`
`  <member name="third" type= ↪"nonBasic" nonBasicTypeName="int64"/>`
`  </case>`
`</union>`
	`  DynamicTypeBuilder_ptr long_ ↪builder = ↪DynamicTypeBuilderFactory::get_ ↪instance()->create_int32_builder();`
	`  DynamicTypeBuilder_ptr long_long_ ↪builder = ↪DynamicTypeBuilderFactory::get_ ↪instance()->create_int64_builder();`
	`  DynamicTypeBuilder_ptr struct_ ↪builder = ↪DynamicTypeBuilderFactory::get_ ↪instance()->create_struct_builder();`
	`  DynamicTypeBuilder_ptr octet_ ↪builder = ↪DynamicTypeBuilderFactory::get_ ↪instance()->create_byte_builder();`
	`  DynamicTypeBuilder_ptr union_ ↪builder = ↪DynamicTypeBuilderFactory::get_ ↪instance()->create_union_builder( octet_builder.get());`
	
	`  union_builder->set_name("MyUnion ↪");`
	`  union_builder->add_member(0, ↪"first", long_builder.get(), "", { 0, ↪1 }, false);`
	`  union_builder->add_member(1, ↪"second", struct_builder.get(), "", { ↪2 }, false);`
	`  union_builder->add_member(2, ↪"third", long_long_builder.get(), "", ↪{ }, true);`
	`  DynamicType_ptr union_type = ↪DynamicTypeBuilderFactory::get_ ↪instance()->create_type(union_builder. ↪get());`

Bitset

The `<bitset>` type is defined by its `name` and inner *bitfields*.

Example:

XML	C++

`<bitset name="MyBitSet">`
`  <bitfield name="a" bit_bound="3"/>`
`  <bitfield name="b" bit_bound="1"/>`
`  <bitfield bit_bound="4"/>`
`  <bitfield name="c" bit_bound="10"/>`
`  <bitfield name="d" bit_bound="12" type="int16"/>`
`</bitset>`
	`DynamicTypeBuilderFactory* m_factory = DynamicTypeBuilderFactory::get_instance();`
	`DynamicTypeBuilder_ptr builder_ptr = m_factory->create_bitset_builder();`
	`builder_ptr->add_member(0, "a", m_factory->create_byte_builder()->build());`
	`builder_ptr->add_member(1, "b", m_factory->create_bool_builder()->build());`
	`builder_ptr->add_member(3, "c", m_factory->create_uint16_builder()->build());`
	`builder_ptr->add_member(4, "d", m_factory->create_int16_builder()->build());`
	`builder_ptr->apply_annotation_to_member(0, ANNOTATION_BIT_BOUND_ID, "value", "3");`
	`builder_ptr->apply_annotation_to_member(0, ANNOTATION_POSITION_ID, "value", "0");`
	`builder_ptr->apply_annotation_to_member(1, ANNOTATION_BIT_BOUND_ID, "value", "1");`
	`builder_ptr->apply_annotation_to_member(1, ANNOTATION_POSITION_ID, "value", "3");`
	`builder_ptr->apply_annotation_to_member(3, ANNOTATION_BIT_BOUND_ID, "value", "10");`
	`builder_ptr->apply_annotation_to_member(3, ANNOTATION_POSITION_ID, "value", "8"); // 4 empty`
	`builder_ptr->apply_annotation_to_member(4, ANNOTATION_BIT_BOUND_ID, "value", "12");`
	`builder_ptr->apply_annotation_to_member(4, ANNOTATION_POSITION_ID, "value", "18");`
	`builder_ptr->set_name("MyBitSet");`

A bitfield without name is an inaccessible set of bits. Bitfields can specify their management type to ease their modification and access. The bitfield's bit_bound is mandatory and cannot be bigger than 64.

Bitsets can inherit from another bitsets:

XML	C++

`<bitset name="ParentBitSet">`
`  <bitfield name="a" bit_bound="3"/>`
`  <bitfield name="b" bit_bound="1"/>`
`</bitset>`
 	
`<bitset name="ChildBitSet" baseType="ParentBitSet">`
`  <bitfield name="c" bit_bound="30"/>`
`  <bitfield name="d" bit_bound="13"/>`
`</bitset>`
	`DynamicTypeBuilderFactory* m_factory = DynamicTypeBuilderFactory::get_instance();`
	`DynamicTypeBuilder_ptr builder_ptr = m_factory->create_bitset_builder();`
	`builder_ptr->add_member(0, "a", m_factory->create_byte_builder()->build());`
	`builder_ptr->add_member(1, "b", m_factory->create_bool_builder()->build());`
	`builder_ptr->apply_annotation_to_member(0, ANNOTATION_BIT_BOUND_ID, "value", "3");`
	`builder_ptr->apply_annotation_to_member(0, ANNOTATION_POSITION_ID, "value", "0");`
	`builder_ptr->apply_annotation_to_member(1, ANNOTATION_BIT_BOUND_ID, "value", "1");`
	`builder_ptr->apply_annotation_to_member(1, ANNOTATION_POSITION_ID, "value", "3");`
	`builder_ptr->set_name("ParentBitSet");`
	
	`DynamicTypeBuilder_ptr child_ptr = m_factory->create_child_struct_builder(builder_ptr.get());`
	`child_ptr->add_member(3, "c", m_factory->create_uint16_builder()->build());`
	`child_ptr->add_member(4, "d", m_factory->create_int16_builder()->build());`
	`child_ptr->apply_annotation_to_member(3, ANNOTATION_BIT_BOUND_ID, "value", "10");`
	`child_ptr->apply_annotation_to_member(3, ANNOTATION_POSITION_ID, "value", "8"); // 4 empty`
	`child_ptr->apply_annotation_to_member(4, ANNOTATION_BIT_BOUND_ID, "value", "12");`
	`child_ptr->apply_annotation_to_member(4, ANNOTATION_POSITION_ID, "value", "18");`
	`child_ptr->set_name("ChildBitSet");`

Bitmask

The `<bitmask>` type is defined by its `name` and inner *bit_values*.

Example:

XML	C++

`<bitmask name="MyBitMask" bit_bound="8">`
`  <bit_value name="flag0" position="0"/>`
`  <bit_value name="flag1"/>`
`  <bit_value name="flag2" position="2"/>`
`  <bit_value name="flag5" position="5"/>`
`</bitmask>`
	`DynamicTypeBuilderFactory* m_factory = DynamicTypeBuilderFactory::get_instance();`
	`DynamicTypeBuilder_ptr builder_ptr = m_factory->create_bitmask_builder(8);`
	`builder_ptr->add_empty_member(0, "flag0");`
	`builder_ptr->add_empty_member(1, "flag1");`
	`builder_ptr->add_empty_member(2, "flag2");`
	`builder_ptr->add_empty_member(5, "flag5");`
	`builder_ptr->set_name("MyBitMask");`

The bitmask can specify its bit_bound, this is, the number of bits that the type will manage. Internally will be converted to the minimum type that allows to store them. The maximum allowed bit_bound is 64. Bit_values can define their position inside the bitmask.

Member types

Member types are any type that can belong to a `<struct>` or a `<union>`, or be aliased by a `<typedef>`.

Basic types

The identifiers of the available basic types are:

boolean	int64	float128
byte	uint16	string
char	uint32	wstring
wchar	uint64	
int16	float32	
int32	float64	

All of them are defined as follows:

XML	C++

`<member name="my_long" type="int64"/>`	`DynamicTypeBuilder_ptr long_long_builder = DynamicTypeBuilderFactory::get_instance()->create_int64_builder();`
	`long_long_builder->set_name("my_long");`
	`DynamicType_ptr long_long_type = DynamicTypeBuilderFactory::get_instance()->create_type(long_long_builder.get());`

Arrays

Arrays are defined in the same way as any other member type but add the attribute `arrayDimensions`. The format of this dimensions attribute is the size of each dimension separated by commas.

Example:

XML	C++

`<member name="long_array" type="int32" arrayDimensions="2,3,4"/>`	`std::vector<uint32_t> lengths = { 2, 3, 4 };`
	`DynamicTypeBuilder_ptr long_builder = DynamicTypeBuilderFactory::get_instance()->create_int32_builder();`
	`DynamicTypeBuilder_ptr array_builder = DynamicTypeBuilderFactory::get_instance()->create_array_builder(long_builder.get(), lengths);`
	`array_builder->set_name("long_array");`
	`DynamicType_ptr array_type = DynamicTypeBuilderFactory::get_instance()->create_type(array_builder.get());`

It's IDL analog would be:

```
long long_array[2][3][4];
```

Sequences

Sequences are defined by its `name`, its content `type`, and its `sequenceMaxLength`. The type of its content should be defined by its `type` attribute.

Example:

XML	C++

`<typedef name="my_sequence_inner" type="int32" sequenceMaxLength="2"/>`
`<struct name="SeqSeqStruct">`
`  <member name="my_sequence_sequence" type="nonBasic" nonBasicTypeName="my_sequence_inner" sequenceMaxLength="3"/>`
`</struct>`
	`uint32_t child_len = 2;`
	`DynamicTypeBuilder_ptr long_builder = DynamicTypeBuilderFactory::get_instance()->create_int32_builder();`
	`DynamicTypeBuilder_ptr seq_builder = DynamicTypeBuilderFactory::get_instance()->create_sequence_builder(`
	`  long_builder.get(),`
	`  child_len);`
	`uint32_t length = 3;`
	`DynamicTypeBuilder_ptr seq_seq_builder = DynamicTypeBuilderFactory::get_instance()->create_sequence_builder(`
	`  seq_builder.get(), length);`
	`seq_seq_builder->set_name("my_sequence_sequence");`
	`DynamicType_ptr seq_type = DynamicTypeBuilderFactory::get_instance()->create_type(seq_seq_builder.get());`

The example shows a sequence with `sequenceMaxLength` 3 of sequences with `sequenceMaxLength` 2 with `<int32>` contents. As IDL would be:

```
sequence<sequence<long,2>,3> my_sequence_sequence;
```

Note that the inner sequence has been defined before.

Maps

Maps are similar to sequences, but they need to define two types instead of one. One type defines its `key_type`, and the other type defines its elements types. Again, both types can be defined as attributes or as members, but when defined as members, they should be contained in another XML element (`<key_type>` and `<type>` respectively).

Example:

XML	C++

`<typedef name="my_map_inner" type="int32" key_type="int32" mapMaxLength="2"/>`
`<struct name="MapMapStruct">`
`  <member name="my_map_map" type="nonBasic" nonBasicTypeName="my_map_inner" key_type="int32" mapMaxLength="2"/>`
`</struct>`
	`uint32_t length = 2;`
	`DynamicTypeBuilder_ptr long_builder = DynamicTypeBuilderFactory::get_instance()->create_int32_builder();`
	`DynamicTypeBuilder_ptr map_builder = DynamicTypeBuilderFactory::get_instance()->create_map_builder(`
	`  long_builder.get(),`
	`  long_builder.get(), length);`
	
	`DynamicTypeBuilder_ptr map_map_builder = DynamicTypeBuilderFactory::get_instance()->create_map_builder(`
	`  long_builder.get(),`
	`  map_builder.get(), length);`
	`map_map_builder->set_name("my_map_map");`
	`DynamicType_ptr map_type = DynamicTypeBuilderFactory::get_instance()->create_type(map_map_builder.get());`

Is equivalent to the IDL:

```
map<long,map<long,long,2>,2> my_map_map;
```

Complex types

Once defined, complex types can be used as members in the same way a basic or array type would be.

Example:

```
<struct name="OtherStruct">
    <member name="my_enum" type="nonBasic" nonBasicTypeName="MyEnum"/>
    <member name="my_struct" type="nonBasic" nonBasicTypeName="MyStruct" arrayDimensions="5"/>
</struct>
```

Usage

In the application that will make use of *XML Types*, it's mandatory to load the XML file that defines the types before trying to instantiate *DynamicPubSubTypes* of these types. It's important to remark that only `<struct>` types generate usable *DynamicPubSubType* instances.

```
        // Load the XML File
        XMLP_ret ret = XMLProfileManager::loadXMLFile("types.xml");
        // Create the "MyStructPubSubType"
        DynamicPubSubType* pbType = XMLProfileManager::CreateDynamicPubSubType("MyStruct");
```

(continues on next page)

(continued from previous page)

```
        // Create a "MyStruct" instance
        DynamicData* data = DynamicDataFactory::get_instance()->create_data(pbType->
↪GetDynamicType());
```

6.12.5 Participant profiles

Participant profiles allow declaring *DomainParticipantQos* from an XML file. All the configuration options for the participant, except from `domainId`, belong to the `<rtps>` label. The attribute `profile_name` will be the name that the `Domain` will associate to the profile to load it as shown in *Loading and applying profiles*.

```
<participant profile_name="part_profile_name">
    <domainId>80</domainId>

    <rtps>
        <name>Participant Name</name> <!-- String -->

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

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

        <sendSocketBufferSize>8192</sendSocketBufferSize> <!-- uint32 -->

        <listenSocketBufferSize>8192</listenSocketBufferSize>  <!-- uint32 -->

        <builtin>
            <!-- BUILTIN -->
        </builtin>

        <port>
            <portBase>7400</portBase> <!-- uint16 -->
            <domainIDGain>200</domainIDGain> <!-- uint16 -->
            <participantIDGain>10</participantIDGain> <!-- uint16 -->
            <offsetd0>0</offsetd0> <!-- uint16 -->
            <offsetd1>1</offsetd1> <!-- uint16 -->
            <offsetd2>2</offsetd2> <!-- uint16 -->
            <offsetd3>3</offsetd3> <!-- uint16 -->
        </port>

        <participantID>99</participantID>   <!-- int32 -->

        <throughputController>
            <bytesPerPeriod>8192</bytesPerPeriod> <!-- uint32 -->
            <periodMillisecs>1000</periodMillisecs> <!-- uint32 -->
        </throughputController>
```

(continues on next page)

(continued from previous page)

```
        <userTransports>
            <transport_id>TransportId1</transport_id> <!-- string -->
            <transport_id>TransportId2</transport_id> <!-- string -->
        </userTransports>

        <useBuiltinTransports>false</useBuiltinTransports> <!-- boolean -->

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

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

    </rtps>
</participant>
```

List with the configuration tags:

Name	Description	Values	Default
`<domainId>`	DomainId to be used by the RTPSParticipant.	`UInt32`	0
`<rtps>`	RTPS parameters. Explained in *RTPS*.	*RTPS*	

RTPS Configuration

List with the possible configuration parameter:

Name	Description	Values	Default
`<name>`	Participant's name. It's not the same field that `profile_name`.	`string_255`	
`<defaultUnicastLocator>`	List of default input unicast locators. It expects a *LocatorListType*.	`LocatorListType`	
`<defaultMulticastLocator>`	List of default input multicast locators. It expects a *LocatorListType*.	`LocatorListType`	
`<sendSocketBufferSize>`	Size in bytes of the output socket buffer. If the value is zero then FastRTPS will use the default size from the configuration of the sockets, using a minimum size of 65536 bytes.	`uint32`	0
`<listenSocketBufferSize>`	Size in bytes of the input socket buffer. If the value is zero then FastRTPS will use the default size from the configuration of the sockets, using a minimum size of 65536 bytes.	`uint32`	0
`<builtin>`	Built-in parameters. Explained in the *Built-in parameters* section.	*Built-in parameters*	
`<port>`	Allows defining the port parameters and gains related to the RTPS protocol. Explained in the *Port* section.	*Port*	
`<participantID>`	Participant's identifier. Typically it will be automatically generated by the `Domain`.	`int32`	0
`<throughputController>`	Allows defining a maximum throughput. Explained in the *Throughput* section.	*Throughput*	
`<userTransports>`	Transport descriptors to be used by the participant.	`List <string>`	
`<useBuiltinTransports>`	Boolean field to indicate to the system that the participant will use the default builtin transport independently of its `<userTransports>`.	`bool`	true
`<propertiesPolicy>`	Additional configuration properties. It expects a *PropertiesPolicyType*.	*PropertiesPolicyType*	
`<allocation>`	Configuration regarding allocation behavior. It expects a *Participant allocation parameters*	*Participant allocation parameters*	

Note:

- `LOCATOR_LIST` means it expects a *LocatorListType*.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- `DURATION` means it expects a *DurationType*.
- For `BUILTIN` details, please refer to *Built-in parameters*.
- For `ALLOCATION` details, please refer to *Participant allocation parameters*.

Port Configuration

Name	Description	Values	Default
`<portBase>`	Base `port`.	`uint16`	7400
`<domainIDGain>`	Gain in `domainId`.	`uint16`	250
`<participantIDGain>`	Gain in `participantId`.	`uint16`	2
`<offsetd0>`	Multicast metadata offset.	`uint16`	0
`<offsetd1>`	Unicast metadata offset.	`uint16`	10
`<offsetd2>`	Multicast user data offset.	`uint16`	1
`<offsetd3>`	Unicast user data offset.	`uint16`	11

Participant allocation parameters

This section of the `Participant's rtps` configuration allows defining parameters related with allocation behavior on the participant.

```
<allocation>
    <remote_locators>
        <max_unicast_locators>4</max_unicast_locators> <!-- uint32 -->
        <max_multicast_locators>1</max_multicast_locators> <!-- uint32 -->
    </remote_locators>
    <total_participants>
        <initial>0</initial>
        <maximum>0</maximum>
        <increment>1</increment>
    </total_participants>
    <total_readers>
        <initial>0</initial>
        <maximum>0</maximum>
        <increment>1</increment>
    </total_readers>
    <total_writers>
        <initial>0</initial>
        <maximum>0</maximum>
        <increment>1</increment>
    </total_writers>
    <max_partitions>256</max_partitions>
    <max_user_data>256</max_user_data>
    <max_properties>512</max_properties>
</allocation>
```

Name	Description	Values	Default
`<max_unicast_locators>`	Maximum number of unicast locators expected on a remote entity. It is recommended to use the maximum number of network interfaces found on any machine the participant will connect to.	`UInt32`	4
`<max_multicast_locators>`	Maximum number of multicast locators expected on a remote entity. May be set to zero to disable multicast traffic.	`UInt32`	1
`<total_participants>`	Participant *Allocation Configuration* related to the total number of participants in the domain (local and remote).	*Allocation Configuration*	
`<total_readers>`	Participant *Allocation Configuration* related to the total number of readers on each participant (local and remote).	*Allocation Configuration*	
`<total_writers>`	Participant *Allocation Configuration* related to the total number of writers on each participant (local and remote).	*Allocation Configuration*	
`<max_partitions>`	Maximum size of the partitions submessage. Zero for no limit. See *Submessage Size Limit*.	`UInt32`	
`<max_user_data>`	Maximum size of the user data submessage. Zero for no limit. See *Submessage Size Limit*.	`UInt32`	
`<max_properties>`	Maximum size of the properties submessage. Zero for no limit. See *Submessage Size Limit*.	`UInt32`	

Built-in parameters

This section of the `Participant's rtps` configuration allows defining built-in parameters.

```
<builtin>
    <discovery_config>

        <discoveryProtocol>NONE</discoveryProtocol> <!-- DiscoveryProtocol enum -->

        <ignoreParticipantFlags>FILTER_DIFFERENT_HOST</ignoreParticipantFlags> <!--␣
↪ParticipantFlags enum -->

        <EDP>SIMPLE</EDP>  <!-- string -->

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

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

        <initialAnnouncements>
```

(continues on next page)

(continued from previous page)

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

        <simpleEDP>
            <PUBWRITER_SUBREADER>true</PUBWRITER_SUBREADER>  <!-- boolean -->
            <PUBREADER_SUBWRITER>true</PUBREADER_SUBWRITER>  <!-- boolean -->
        </simpleEDP>

        <staticEndpointXMLFilename>filename.xml</staticEndpointXMLFilename>  <!--␣
↪string -->

    </discovery_config>

    <avoid_builtin_multicast>true</avoid_builtin_multicast>

    <use_WriterLivelinessProtocol>false</use_WriterLivelinessProtocol>  <!-- boolean -
↪->

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

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

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

    <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</readerHistoryMemoryPolicy>
    <readerPayloadSize>512</readerPayloadSize>

    <writerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</writerHistoryMemoryPolicy>
    <writerPayloadSize>512</writerPayloadSize>

    <mutation_tries>55</mutation_tries>
</builtin>
```

Name	Description	Values	Default
`<discovery_config>`	This is the main tag where discovery-related settings can be configured.	*discovery_config*	
`<avoid_builtin_multicast>`	Restricts metatraffic multicast traffic to PDP only.	`Boolean`	`true`
`<use_WriterLiveliness>`	Indicates to use the WriterLiveliness protocol.	`Boolean`	`true`
`<metatrafficUnicastLocatorList>`	Metatraffic Unicast Locator List	List of *LocatorListType*	
`<metatrafficMulticastLocatorList>`	Metatraffic Multicast Locator List	List of *LocatorListType*	
`<initialPeersList>`	Initial peers.	List of *LocatorListType*	
`<readerHistoryMemoryPolicy>`	Memory policy for builtin readers.	*historyMemoryPolicy*	`PREALLOCATED_WITH_REALLOC`
`<writerHistoryMemoryPolicy>`	Memory policy for builtin writers.	*historyMemoryPolicy*	`PREALLOCATED_WITH_REALLOC`
`<readerPayloadSize>`	Maximum payload size for builtin readers.	`UInt32`	512
`<writerPayloadSize>`	Maximum payload size for builtin writers.	`UInt32`	512
`<mutation_tries>`	Number of different ports to try if reader's physical port is already in use.	`UInt32`	100

discovery_config

Name	Description	Values	Default
`<discoveryProtocol>`	Indicates which kind of PDP protocol the participant must use.	`SIMPLE`, `CLIENT`, `SERVER`, `BACKUP`	`SIMPLE`
`<ignoreParticipantFlags>`	Restricts metatraffic using several filtering criteria.	*ignoreParticipantFlags*	`NO_FILTER`
`<EDP>`	• If set to `SIMPLE`, `<simpleEDP>` would be used. • If set to `STATIC`, StaticEDP based on an XML file would be used with the contents of `<staticEndpointXMLFilename>`.	`SIMPLE`, `STATIC`	`SIMPLE`
`<simpleEDP>`	Attributes of the SimpleEDP protocol	*simpleEDP*	
`<leaseDuration>`	Indicates how long this RTPSParticipant should consider remote RTPSParticipants alive.	*DurationType*	20 s
`<leaseAnnouncement>`	The period for the RTPSParticipant to send its Discovery Message to all other discovered RTPSParticipants as well as to all Multicast ports.	*DurationType*	3 s
`<initialAnnouncements>`	Allows the user to configure the number and period of the initial RTPSparticipant's Discovery messages.	*Initial Announcements*	
`<staticEndpointXMLFilename>`	StaticEDP XML filename. Only necessary if `<EDP>` is set to `STATIC`	`string`	

ignoreParticipantFlags

Possible values	Description
`NO_FILTER`	All Discovery traffic is processed
`FILTER_DIFFERENT_HOST`	Discovery traffic from another host is discarded
`FILTER_DIFFERENT_PROCESS`	Discovery traffic from another process on the same host is discarded
`FILTER_SAME_PROCESS`	Discovery traffic from participant's own process is discarded.
`FILTER_DIFFERENT_PROCESS | FILTER_SAME_PROCESS`	Discovery traffic from participant's own host is discarded.

simpleEDP

Name	Description	Values	Default
`<PUBWRITER_SUBREADER>`	Indicates if the participant must use Publication Writer and Subscription Reader.	`Boolean`	`true`
`<PUBREADER_SUBWRITER>`	Indicates if the participant must use Publication Reader and Subscription Writer.	`Boolean`	`true`

Initial Announcements

Name	Description	Values	Default
`<count>`	Number of Discovery Messages to send at the period specified by `<period>`. After these announcements, the RTPSParticipant will continue sending its Discovery Messages at the `<leaseAnnouncement>` rate.	`Uint32`	25
`<period>`	The period for the RTPSParticipant to send its first `<count>` Discovery Messages.	*DurationType*	100 ms

6.12.6 Publisher profiles

Publisher profiles allow declaring *DataWriterQos* from an XML file. The attribute `profile_name` is the name that the `Domain` associates to the profile to load it as shown in the *Loading and applying profiles* section.

```
<publisher profile_name="pub_profile_name">
    <topic>
        <!-- TOPIC_TYPE -->
    </topic>

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

    <times> <!-- writerTimesType -->
        <initialHeartbeatDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>12</nanosec>
        </initialHeartbeatDelay>
        <heartbeatPeriod> <!-- DURATION -->
            <sec>3</sec>
            <nanosec>0</nanosec>
        </heartbeatPeriod>
        <nackResponseDelay> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>5</nanosec>
        </nackResponseDelay>
        <nackSupressionDuration> <!-- DURATION -->
            <sec>0</sec>
            <nanosec>0</nanosec>
        </nackSupressionDuration>
    </times>

    <unicastLocatorList>
        <!-- LOCATOR_LIST -->
```

(continues on next page)

(continued from previous page)

```
        <locator>
            <udpv4/>
        </locator>
    </unicastLocatorList>

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

    <throughputController>
        <bytesPerPeriod>8192</bytesPerPeriod> <!-- uint32 -->
        <periodMillisecs>1000</periodMillisecs> <!-- uint32 -->
    </throughputController>

    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

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

    <userDefinedID>55</userDefinedID> <!-- Int16 -->

    <entityID>66</entityID> <!-- Int16 -->

    <matchedSubscribersAllocation>
        <initial>0</initial> <!-- uint32 -->
        <maximum>0</maximum> <!-- uint32 -->
        <increment>1</increment> <!-- uint32 -->
    </matchedSubscribersAllocation>

</publisher>
```

Note:

- `LOCATOR_LIST` means it expects a *LocatorListType*.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- `DURATION` means it expects a *DurationType*.
- For `QOS` details, please refer to *QOS*.
- `TOPIC_TYPE` is detailed in section *Topic Type*.

Name	Description	Values	Default
`<topic>`	*Topic Type* configuration of the publisher.	*Topic Type*	
`<qos>`	Publisher *QOS* configuration.	*QOS*	
`<times>`	It allows configuring some time related parameters of the publisher.	*Times*	
`<unicastLocatorList>`	List of input unicast locators. It expects a *LocatorListType*.	List of *LocatorListType*	
`<multicastLocatorList>`	List of input multicast locators. It expects a *LocatorListType*.	List of *LocatorListType*	
`<throughputController>`	Limits the output bandwidth of the publisher.	*Throughput*	
`<historyMemoryPolicy>`	Memory allocation kind for publisher's history.	*historyMemoryPolicy*	`PREALLOCATED`
`<propertiesPolicy>`	Additional configuration properties.	*PropertiesPolicyType*	
`<userDefinedID>`	Used for StaticEndpointDiscovery.	`Int16`	-1
`<entityID>`	EntityId of the *endpoint*.	`Int16`	-1
`<matchedSubscribersAllocation>`	Publisher *Allocation Configuration* related to the number of matched subscribers.	*Allocation Configuration*	

Times

Name	Description	Values	Default
`<initialHeartbeatDelay>`	Initial heartbeat delay.	*DurationType*	~45 ms
`<heartbeatPeriod>`	Periodic HB period.	*DurationType*	3 s
`<nackResponseDelay>`	Delay to apply to the response of a ACKNACK message.	*DurationType*	~45 ms
`<nackSupressionDuration>`	This time allows the RTPSWriter to ignore nack messages too soon after the data has been sent.	*DurationType*	0 ms

6.12.7 Subscriber profiles

Subscriber profiles allow declaring *DataReaderQos* from an XML file. The attribute `profile_name` is the name that the `Domain` associates to the profile to load it as shown in *Loading and applying profiles*.

```
<subscriber profile_name="sub_profile_name">
    <topic>
        <!-- TOPIC_TYPE -->
    </topic>

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

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

(continues on next page)

(continued from previous page)

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

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

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

    <expectsInlineQos>true</expectsInlineQos> <!-- boolean -->

    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

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

    <userDefinedID>55</userDefinedID> <!-- Int16 -->

    <entityID>66</entityID> <!-- Int16 -->

    <matchedPublishersAllocation>
        <initial>0</initial> <!-- uint32 -->
        <maximum>0</maximum> <!-- uint32 -->
        <increment>1</increment> <!-- uint32 -->
    </matchedPublishersAllocation>

</subscriber>
```

Note:

- `LOCATOR_LIST` means it expects a *LocatorListType*.
- `PROPERTIES_POLICY` means that the label is a *PropertiesPolicyType* block.
- `DURATION` means it expects a *DurationType*.
- For `QOS` details, please refer to *QOS*.
- `TOPIC_TYPE` is detailed in section *Topic Type*.

Name	Description	Values	Default
`<topic>`	*Topic Type* configuration of the subscriber.	*Topic Type*	
`<qos>`	Subscriber *QOS* configuration.	*QOS*	
`<times>`	It allows configuring some time related parameters of the subscriber.	*Times*	
`<unicastLocatorList>`	List of input unicast locators. It expects a *LocatorListType*.	List of *LocatorListType*	
`<multicastLocatorList>`	List of input multicast locators. It expects a *LocatorListType*.	List of *LocatorListType*	
`<expectsInlineQos>`	It indicates if QOS is expected inline.	`Boolean`	`false`
`<historyMemoryPolicy>`	Memory allocation kind for subscriber's history.	*historyMemoryPolicy*	`PREALLOCATED`
`<propertiesPolicy>`	Additional configuration properties.	*PropertiesPolicyType*	
`<userDefinedID>`	Used for StaticEndpointDiscovery.	`Int16`	-1
`<entityID>`	EntityId of the *endpoint*.	`Int16`	-1
`<matchedPublishersAllocation>`	Subscriber *Allocation Configuration* related to the number of matched publishers.	*Allocation Configuration*	

Times

Name	Description	Values	Default
`<initialAcknackDelay>`	Initial AckNack delay.	*DurationType*	~45 ms
`<heartbeatResponseDelay>`	Delay to be applied when a heartbeat message is received.	*DurationType*	~4.5 ms

6.12.8 Common

In the above profiles, some types are used in several different places. To avoid too many details, some of that places have a tag like `LocatorListType` that indicates that field is defined in this section.

LocatorListType

It represents a list of `Locator_t`. LocatorListType is normally used as an anonymous type, this is, it hasn't its own label. Instead, it is used inside other configuration parameter labels that expect a list of locators and give it sense, for example, in `<defaultUnicastLocatorList>`. The locator kind is defined by its own tag and can take the values `<udpv4>`, `<tcpv4>`, `<udpv6>`, and `<tcpv6>`:

```
<defaultUnicastLocatorList>
    <locator>
        <udpv4>
            <!-- Access as physical, typical UDP usage -->
            <port>7400</port> <!-- uint32 -->
            <address>192.168.1.41</address>
        </udpv4>
    </locator>
    <locator>
        <tcpv4>
            <!-- Both physical and logical (port), useful in TCP transports -->
```

(continues on next page)

(continued from previous page)

```
            <physical_port>5100</physical_port> <!-- uint16 -->
            <port>7400</port> <!-- uint16 -->
            <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
            <wan_address>80.80.99.45</wan_address>
            <address>192.168.1.55</address>
        </tcpv4>
    </locator>
    <locator>
        <udpv6>
            <port>8844</port>
            <address>::1</address>
        </udpv6>
    </locator>
    <locator>
        <tcpv6>
            <!-- Both physical and logical (port), useful in TCP transports -->
            <physical_port>5100</physical_port> <!-- uint16 -->
            <port>7400</port> <!-- uint16 -->
            <address>fe80::55e3:290:165:5af8</address>
        </tcpv6>
    </locator>
</defaultUnicastLocatorList>
```

In this example, there are one locator of each kind in `<defaultUnicastLocatorList>`.

Let's see each possible Locator's field in detail:

Name	Description	Values	Default
`<port>`	RTPS port number of the locator. *Physical port* in UDP, *logical port* in TCP.	`Uint32`	0
`<physical_port>`	TCP's *physical port*.	`Uint32`	0
`<address>`	IP address of the locator.	`string` with expected format	""
`<unique_lan_id>`	The LAN ID uniquely identifies the LAN the locator belongs to (**TCPv4 only**).	`string` (16 bytes)	
`<wan_address>`	WAN IPv4 address (**TCPv4 only**).	`string` with IPv4 Format	`0.0.0.0`

PropertiesPolicyType

PropertiesPolicyType (XML label `<propertiesPolicy>`) allows defining a set of generic properties. It's useful at defining extended or custom configuration parameters.

```
<propertiesPolicy>
    <properties>
        <property>
            <name>Property1Name</name> <!-- string -->
            <value>Property1Value</value> <!-- string -->
            <propagate>false</propagate> <!-- boolean -->
        </property>
        <property>
            <name>Property2Name</name> <!-- string -->
            <value>Property2Value</value> <!-- string -->
```

(continues on next page)

(continued from previous page)

```
            <propagate>true</propagate> <!-- boolean -->
        </property>
    </properties>
</propertiesPolicy>
```

Name	Description	Values	Default
`<name>`	Name to identify the property.	`string`	
`<value>`	Property's value.	`string`	
`<propagate>`	Indicates if it is going to be serialized along with the object it belongs to.	`Boolean`	`false`

DurationType

DurationType expresses a period of time and it's commonly used as an anonymous type, this is, it hasn't its own label. Instead, it is used inside other configuration parameter labels that give it sense, like `<leaseAnnouncement>` or `<leaseDuration>`.

```
<discovery_config>
    <leaseDuration>
        <sec>DURATION_INFINITY</sec> <!-- string -->
    </leaseDuration>

    <leaseDuration>
        <sec>500</sec> <!-- int32 -->
        <nanosec>0</nanosec> <!-- uint32 -->
    </leaseDuration>

    <leaseAnnouncement>
        <sec>1</sec> <!-- int32 -->
        <nanosec>856000</nanosec> <!-- uint32 -->
    </leaseAnnouncement>
</discovery_config>
```

Duration time can be defined through `<sec>` plus `<nanosec>` labels (see table below). An infinite value can be specified by using the values `DURATION_INFINITY`, `DURATION_INFINITE_SEC` and `DURATION_INFINITE_NSEC`.

Name	Description	Values	Default
`<sec>`	Number of seconds.	`Int32`	0
`<nanosec>`	Number of nanoseconds.	`UInt32`	0

Topic Type

The *Topic* name and data type are used as meta-data to determine whether Publishers and Subscribers can exchange messages.

```
<topic>
    <kind>NO_KEY</kind> <!-- string -->
    <name>TopicName</name> <!-- string -->
    <dataType>TopicDataTypeName</dataType> <!-- string -->
    <historyQos>
        <kind>KEEP_LAST</kind> <!-- string -->
        <depth>20</depth> <!-- uint32 -->
    </historyQos>
    <resourceLimitsQos>
        <max_samples>5</max_samples> <!-- unint32 -->
        <max_instances>2</max_instances> <!-- unint32 -->
        <max_samples_per_instance>1</max_samples_per_instance> <!-- unint32 -->
        <allocated_samples>20</allocated_samples> <!-- unint32 -->
    </resourceLimitsQos>
</topic>
```

Name	Description	Values	Default
`<kind>`	It defines the Topic's kind	`NO_KEY`, `WITH_KEY`	`NO_KEY`
`<name>`	It defines the Topic's name. Must be unique.	`string_255`	
`<dataType>`	It references the Topic's data type.	`string_255`	
`<historyQos>`	It controls the behavior of *Fast RTPS* when the value of an instance changes before it is finally communicated to some of its existing DataReader entities.	*HistoryQos*	
`<resourceLimitsQos>`	It controls the resources that *Fast RTPS* can use in order to meet the requirements imposed by the application and other QoS settings.	*ResourceLimitsQos*	

HistoryQoS

It controls the behavior of *Fast RTPS* when the value of an instance changes before it is finally communicated to some of its existing DataReader entities.

Name	Description	Values	Default
`<kind>`	See description below.	`KEEP_LAST`, `KEEP_ALL`	`KEEP_LAST`
`<depth>`		`UInt32`	1000

If the `<kind>` is set to `KEEP_LAST`, then *Fast RTPS* will only attempt to keep the latest values of the instance and discard the older ones.

If the `<kind>` is set to `KEEP_ALL`, then *Fast RTPS* will attempt to maintain and deliver all the values of the instance to existing subscribers.

The setting of `<depth>` must be consistent with the *ResourceLimitsQos* `<max_samples_per_instance>`. For these two QoS to be consistent, they must verify that `depth <= max_samples_per_instance`.

ResourceLimitsQos

It controls the resources that *Fast RTPS* can use in order to meet the requirements imposed by the application and other QoS settings.

Name	Description	Values	Default
`<max_samples>`	It must verify that `max_samples >= max_samples_per_instance`.	`UInt32`	5000
`<max_instances>`	It defines the maximum number of instances.	`UInt32`	10
`<max_samples_per_instance>`	It must verify that *HistoryQos* `depth <= max_samples_per_instance`.	`UInt32`	400
`<allocated_samples>`	It controls the maximum number of samples to be stored.	`UInt32`	100

QOS

The quality of service (QoS) handles the restrictions applied to the application.

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

    <liveliness>
        <kind>AUTOMATIC</kind> <!-- string -->
        <lease_duration>
          <sec>1</sec>
        </lease_duration>
        <announcement_period>
          <sec>1</sec>
        </announcement_period>
    </liveliness>

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

    <partition>
        <names>
            <name>part1</name> <!-- string -->
            <name>part2</name> <!-- string -->
        </names>
    </partition>

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

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

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

Name	Description	Values	Default
`<durability>`	It is defined in *Setting the data durability kind* section.	`VOLATILE`, `TRANSIENT_LOCAL` `TRANSIENT`	`VOLATILE`
`<liveliness>`	Defines the liveliness of the publisher.	*LivelinessQosPolicy*	
`<reliability>`	It is defined in *ReliabilityQosPolicy* section.	`RELIABLE`, `BEST_EFFORT`	`RELIABLE`
`<partition>`	It allows the introduction of a logical partition concept inside the *physical* partition induced by a domain.		`List <string>`
`<deadline>`	It is defined in *DeadlineQosPolicy* section.	Deadline period as a *DurationType*	`c_TimeInfinite`
`<lifespan>`	It is defined in *LifespanQosPolicy* section.	Lifespan duration as a *DurationType*	`c_TimeInfinite`
`<disablePositiveACKs>`	It is defined in section *DisablePositiveACKsQosPolicy*		It is disabled by default and `duration` is set to `c_TimeInfinite`

Throughput Configuration

Throughput Configuration allows to limit the output bandwidth.

Name	Description	Values	Default
`<bytesPerPeriod>`	Packet size in bytes that this controller will allow in a given period.	`UInt32`	4294967295
`<periodMillisecs>`	Window of time in which no more than `<bytesPerPeriod>` bytes are allowed.	`UInt32`	0

Allocation Configuration

Allocation Configuration 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 publisher which depend on the number of subscribers matching with it.

See *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`	0
`<maximum>`	Maximum number of elements for which space will be allocated.	`UInt32`	0 (means no limit)
`<increment>`	Number of new elements that will be allocated when more space is necessary.	`UInt32`	1

Submessage Size Limit

While some submessages have a fixed size (for example, SequenceNumber), others have a variable size depending on the data they contain. Processing a submessage requires having a memory chunk large enough to contain a copy of the submessage data. That is easy to handle when dealing with fixed variable submessages, as size is known and memory can be allocated beforehand. For variable size submessages on the other hand, two different strategies can be used:

- Set a maximum size for the data container, which will be allocated beforehand during the participant's setup. This avoids dynamic allocations during message communication. However, any submessages with a larger payload than the defined maximum will not fit in, and will therefore be discarded.
- Do not set any maximum size for the data container, and instead allocate the required memory dynamically upon submessage arrival (according to the size declared on the submessage header). This allows for any size of submessages, at the cost of dynamic allocations during message decoding.

History Memory Policy Configuration

Controls the allocation behavior of the change histories.

- **PREALLOCATED**: As the history gets larger, memory is allocated in chunks. Each chunk accommodates a number of changes, and no more allocations are done until that chunk is full. Provides minimum number of dynamic allocations at the cost of increased memory footprint. Maximum payload size of changes must be appropriately configured, as history will not be able to accommodate changes with larger payload after the allocation.
- **PREALLOCATED_WITH_REALLOC**: Like PREALLOCATED, but preallocated memory can be reallocated to accommodate changes with larger payloads than the defined maximum.
- **DYNAMIC**: Every change gets a fresh new allocated memory of the correct size. It minimizes the memory footprint, at the cost of increased number of dynamic allocations.
- **DYNAMIC_REUSABLE**: Like DYNAMIC, but instead of deallocating the memory when the change is removed from the history, it is reused for a future change, reducing the amount of dynamic allocations. If the new change has larger payload, it will be reallocated to accommodate the new size.

6.12.9 Example

In this section, there is a full XML example with all possible configuration. It can be used as a quick reference, but it may not be valid due to incompatibility or exclusive properties. Don't take it as a working example.

```
<profiles>
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>ExampleTransportId1</transport_id>
            <type>TCPv4</type>
            <sendBufferSize>8192</sendBufferSize>
            <receiveBufferSize>8192</receiveBufferSize>
            <TTL>250</TTL>
            <maxMessageSize>16384</maxMessageSize>
            <maxInitialPeersRange>100</maxInitialPeersRange>
            <interfaceWhiteList>
                <address>192.168.1.41</address>
                <address>127.0.0.1</address>
            </interfaceWhiteList>
            <wan_addr>80.80.55.44</wan_addr>
            <keep_alive_frequency_ms>5000</keep_alive_frequency_ms>
```

(continues on next page)

(continued from previous page)

```
            <keep_alive_timeout_ms>25000</keep_alive_timeout_ms>
            <max_logical_port>200</max_logical_port>
            <logical_port_range>20</logical_port_range>
            <logical_port_increment>2</logical_port_increment>
            <listening_ports>
                <port>5100</port>
                <port>5200</port>
            </listening_ports>
        </transport_descriptor>
        <transport_descriptor>
            <transport_id>ExampleTransportId2</transport_id>
            <type>UDPv6</type>
        </transport_descriptor>
        <!-- SHM sample transport descriptor -->
        <transport_descriptor>
            <transport_id>SHM_SAMPLE_DESCRIPTOR</transport_id>
                <type>SHM</type> <!-- REQUIRED -->
                <maxMessageSize>524288</maxMessageSize> <!-- OPTIONAL uint32 valid of␣
↪all transports-->
                <segment_size>1048576</segment_size> <!-- OPTIONAL uint32 SHM only-->
                <port_queue_capacity>1024</port_queue_capacity> <!-- OPTIONAL uint32␣
↪SHM only-->
                <healthy_check_timeout_ms>250</healthy_check_timeout_ms> <!--␣
↪OPTIONAL uint32 SHM only-->
                <rtps_dump_file>test_file.dump</rtps_dump_file> <!-- OPTIONAL string␣
↪SHM only-->
        </transport_descriptor>
    </transport_descriptors>

    <types>
        <type> <!-- Types can be defined in its own type of tag or sharing the same␣
↪tag -->
            <enum name="MyAloneEnumType">
                <enumerator name="A" value="0"/>
                <enumerator name="B" value="1"/>
                <enumerator name="C" value="2"/>
            </enum>
        </type>
        <type>
            <enum name="MyEnumType">
                <enumerator name="A" value="0"/>
                <enumerator name="B" value="1"/>
                <enumerator name="C" value="2"/>
            </enum>

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

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

            <struct name="MyStruct1">
                <member name="first" type="int32"/>
                <member name="second" type="int64"/>
            </struct>

            <union name="MyUnion1">
                <discriminator type="byte"/>
                <case>
```

(continues on next page)

(continued from previous page)

```
                    <caseDiscriminator value="0"/>
                    <caseDiscriminator value="1"/>
                    <member name="first" type="int32"/>
                </case>
                <case>
                    <caseDiscriminator value="2"/>
                    <member name="second" type="nonBasic" nonBasicTypeName="MyStruct"/
↪>
                </case>
                <case>
                    <caseDiscriminator value="default"/>
                    <member name="third" type="int64"/>
                </case>
            </union>

            <!-- All possible members struct type -->
            <struct name="MyFullStruct">
                <!-- Primitives & basic -->
                <member name="my_bool" type="boolean"/>
                <member name="my_byte" type="byte"/>
                <member name="my_char" type="char8"/>
                <member name="my_wchar" type="char16"/>
                <member name="my_short" type="int16"/>
                <member name="my_long" type="int32"/>
                <member name="my_longlong" type="int64"/>
                <member name="my_unsignedshort" type="uint16"/>
                <member name="my_unsignedlong" type="uint32"/>
                <member name="my_unsignedlonglong" type="uint64"/>
                <member name="my_float" type="float32"/>
                <member name="my_double" type="float64"/>
                <member name="my_longdouble" type="float128"/>
                <member name="my_string" type="string"/>
                <member name="my_wstring" type="wstring"/>
                <member name="my_boundedString" type="string" stringMaxLength="41925"/
↪>
                <member name="my_boundedWString" type="wstring" stringMaxLength="41925
↪"/>

                <!-- long long_array[2][3][4]; -->
                <member name="long_array" arrayDimensions="2,3,4" type="int32"/>

                <!-- map<long,map<long,long,2>,2> my_map_map; -->
                <!-->
                <typdefe name="my_map_inner" type="int32" key_type="int32"␣
↪mapMaxLength="2"/>
                <-->
                <member name="my_map_map" type="nonBasic" nonBasicTypeName="my_map_
↪inner" key_type="int32" mapMaxLength="2"/>

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

    <participant profile_name="part_profile_example">
```

(continues on next page)

(continued from previous page)

```
        <domainId>4</domainId>
        <rtps>
            <name>Participant Name</name> <!-- String -->

            <defaultUnicastLocatorList>
                <locator>
                    <udpv4>
                        <!-- Access as physical, like UDP -->
                        <port>7400</port>
                        <address>192.168.1.41</address>
                    </udpv4>
                </locator>
                <locator>
                    <tcpv4>
                        <!-- Both physical and logical (port), like TCP -->
                        <physical_port>5100</physical_port>
                        <port>7400</port>
                        <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.55</address>
                    </tcpv4>
                </locator>
                <locator>
                    <udpv6>
                        <port>8844</port>
                        <address>::1</address>
                    </udpv6>
                </locator>
            </defaultUnicastLocatorList>

            <defaultMulticastLocatorList>
                <locator>
                    <udpv4>
                        <!-- Access as physical, like UDP -->
                        <port>7400</port>
                        <address>192.168.1.41</address>
                    </udpv4>
                </locator>
                <locator>
                    <tcpv4>
                        <!-- Both physical and logical (port), like TCP -->
                        <physical_port>5100</physical_port>
                        <port>7400</port>
                        <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                        <wan_address>80.80.99.45</wan_address>
                        <address>192.168.1.55</address>
                    </tcpv4>
                </locator>
                <locator>
                    <udpv6>
                        <port>8844</port>
                        <address>::1</address>
                    </udpv6>
                </locator>
            </defaultMulticastLocatorList>

            <sendSocketBufferSize>8192</sendSocketBufferSize>
```

(continues on next page)

(continued from previous page)

```
            <listenSocketBufferSize>8192</listenSocketBufferSize>

            <builtin>
                <discovery_config>

                    <discoveryProtocol>NONE</discoveryProtocol>

                    <EDP>SIMPLE</EDP>

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

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

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

                    <staticEndpointXMLFilename>filename.xml</
↪staticEndpointXMLFilename>

                </discovery_config>

                <use_WriterLivelinessProtocol>false</use_WriterLivelinessProtocol>

                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <!-- Access as physical, like UDP -->
                            <port>7400</port>
                            <address>192.168.1.41</address>
                        </udpv4>
                    </locator>
                    <locator>
                        <tcpv4>
                            <!-- Both physical and logical (port), like TCP -->
                            <physical_port>5100</physical_port>
                            <port>7400</port>
                            <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                            <wan_address>80.80.99.45</wan_address>
                            <address>192.168.1.55</address>
                        </tcpv4>
                    </locator>
                    <locator>
                        <udpv6>
                            <port>8844</port>
                            <address>::1</address>
                        </udpv6>
                    </locator>
                </metatrafficUnicastLocatorList>
```

(continues on next page)

(continued from previous page)

```
                <metatrafficMulticastLocatorList>
                    <locator>
                        <udpv4>
                            <!-- Access as physical, like UDP -->
                            <port>7400</port>
                            <address>192.168.1.41</address>
                        </udpv4>
                    </locator>
                    <locator>
                        <tcpv4>
                            <!-- Both physical and logical (port), like TCP -->
                            <physical_port>5100</physical_port>
                            <port>7400</port>
                            <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                            <wan_address>80.80.99.45</wan_address>
                            <address>192.168.1.55</address>
                        </tcpv4>
                    </locator>
                    <locator>
                        <udpv6>
                            <port>8844</port>
                            <address>::1</address>
                        </udpv6>
                    </locator>
                </metatrafficMulticastLocatorList>

                <initialPeersList>
                    <locator>
                        <udpv4>
                            <!-- Access as physical, like UDP -->
                            <port>7400</port>
                            <address>192.168.1.41</address>
                        </udpv4>
                    </locator>
                    <locator>
                        <tcpv4>
                            <!-- Both physical and logical (port), like TCP -->
                            <physical_port>5100</physical_port>
                            <port>7400</port>
                            <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                            <wan_address>80.80.99.45</wan_address>
                            <address>192.168.1.55</address>
                        </tcpv4>
                    </locator>
                    <locator>
                        <udpv6>
                            <port>8844</port>
                            <address>::1</address>
                        </udpv6>
                    </locator>
                </initialPeersList>

                <readerHistoryMemoryPolicy>PREALLOCATED_WITH_REALLOC</
↪readerHistoryMemoryPolicy>

                <writerHistoryMemoryPolicy>PREALLOCATED</writerHistoryMemoryPolicy>
            </builtin>
```

(continues on next page)

(continued from previous page)

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

            <participantID>99</participantID>

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

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

            <useBuiltinTransports>false</useBuiltinTransports>

            <propertiesPolicy>
                <properties>
                    <property>
                        <name>Property1Name</name>
                        <value>Property1Value</value>
                        <propagate>false</propagate>
                    </property>
                    <property>
                        <name>Property2Name</name>
                        <value>Property2Value</value>
                        <propagate>false</propagate>
                    </property>
                </properties>
            </propertiesPolicy>
        </rtps>
    </participant>

    <publisher profile_name="pub_profile_example">
        <topic>
            <kind>WITH_KEY</kind>
            <name>TopicName</name>
            <dataType>TopicDataTypeName</dataType>
            <historyQos>
                <kind>KEEP_LAST</kind>
                <depth>20</depth>
            </historyQos>
            <resourceLimitsQos>
                <max_samples>5</max_samples>
                <max_instances>2</max_instances>
                <max_samples_per_instance>1</max_samples_per_instance>
                <allocated_samples>20</allocated_samples>
            </resourceLimitsQos>
```

(continues on next page)

(continued from previous page)

```
        </topic>

        <qos> <!-- writerQosPoliciesType -->
            <durability>
                <kind>VOLATILE</kind>
            </durability>
            <liveliness>
                <kind>AUTOMATIC</kind>
                <lease_duration>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </lease_duration>
                <announcement_period>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </announcement_period>
            </liveliness>
            <reliability>
                <kind>BEST_EFFORT</kind>
                <max_blocking_time>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </max_blocking_time>
            </reliability>
            <partition>
                <names>
                    <name>part1</name>
                    <name>part2</name>
                </names>
            </partition>
            <publishMode>
                <kind>ASYNCHRONOUS</kind>
            </publishMode>
            <disablePositiveAcks>
                <enabled>true</enabled>
                    <duration>
                    <sec>1</sec>
                </duration>
            </disablePositiveAcks>
        </qos>

        <times>
            <initialHeartbeatDelay>
                <sec>1</sec>
                <nanosec>856000</nanosec>
            </initialHeartbeatDelay>
            <heartbeatPeriod>
                <sec>1</sec>
                <nanosec>856000</nanosec>
            </heartbeatPeriod>
            <nackResponseDelay>
                <sec>1</sec>
                <nanosec>856000</nanosec>
            </nackResponseDelay>
            <nackSupressionDuration>
                <sec>1</sec>
                <nanosec>856000</nanosec>
```

(continues on next page)

(continued from previous page)

```
            </nackSupressionDuration>
        </times>

        <unicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </unicastLocatorList>

        <multicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </multicastLocatorList>

        <throughputController>
            <bytesPerPeriod>8192</bytesPerPeriod>
```

(continues on next page)

(continued from previous page)

```
            <periodMillisecs>1000</periodMillisecs>
        </throughputController>

        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

        <propertiesPolicy>
            <properties>
                <property>
                    <name>Property1Name</name>
                    <value>Property1Value</value>
                    <propagate>false</propagate>
                </property>
                <property>
                    <name>Property2Name</name>
                    <value>Property2Value</value>
                    <propagate>false</propagate>
                </property>
            </properties>
        </propertiesPolicy>

        <userDefinedID>45</userDefinedID>

        <entityID>76</entityID>
    </publisher>

    <subscriber profile_name="sub_profile_example">
        <topic>
            <kind>WITH_KEY</kind>
            <name>TopicName</name>
            <dataType>TopicDataTypeName</dataType>
            <historyQos>
                <kind>KEEP_LAST</kind>
                <depth>20</depth>
            </historyQos>
            <resourceLimitsQos>
                <max_samples>5</max_samples>
                <max_instances>2</max_instances>
                <max_samples_per_instance>1</max_samples_per_instance>
                <allocated_samples>20</allocated_samples>
            </resourceLimitsQos>
        </topic>

        <qos>
            <durability>
                <kind>PERSISTENT</kind>
            </durability>
            <liveliness>
                <kind>MANUAL_BY_PARTICIPANT</kind>
                <lease_duration>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </lease_duration>
                <announcement_period>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </announcement_period>
            </liveliness>
```

(continues on next page)

(continued from previous page)

```
            <reliability>
                <kind>BEST_EFFORT</kind>
                <max_blocking_time>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
                </max_blocking_time>
            </reliability>
            <partition>
                <names>
                    <name>part1</name>
                    <name>part2</name>
                </names>
            </partition>
        </qos>

        <times>
            <initialAcknackDelay>
                <sec>1</sec>
                <nanosec>856000</nanosec>
            </initialAcknackDelay>
            <heartbeatResponseDelay>
                <sec>1</sec>
                <nanosec>856000</nanosec>
            </heartbeatResponseDelay>
        </times>

        <unicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </unicastLocatorList>

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

(continues on next page)

(continued from previous page)

```
                <address>192.168.1.41</address>
            </udpv4>
        </locator>
        <locator>
            <tcpv4>
                <!-- Both physical and logical (port), like TCP -->
                <physical_port>5100</physical_port>
                <port>7400</port>
                <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                <wan_address>80.80.99.45</wan_address>
                <address>192.168.1.55</address>
            </tcpv4>
        </locator>
        <locator>
            <udpv6>
                <port>8844</port>
                <address>::1</address>
            </udpv6>
        </locator>
    </multicastLocatorList>

    <expectsInlineQos>true</expectsInlineQos>

    <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>

    <propertiesPolicy>
        <properties>
            <property>
                <name>Property1Name</name>
                <value>Property1Value</value>
                <propagate>false</propagate>
            </property>
            <property>
                <name>Property2Name</name>
                <value>Property2Value</value>
                <propagate>false</propagate>
            </property>
        </properties>
    </propertiesPolicy>

    <userDefinedID>55</userDefinedID>

    <entityID>66</entityID>
</subscriber>
```

6.13 Environment variables

6.14 Dynamic Topic Types

eProsima Fast DDS provides a dynamic way to define and use topic types and topic data. Our implementation follows the *OMG Extensible and Dynamic Topic Types for DDS interface*. For more information, you can read the specification for DDS-XTypes V1.2.

The dynamic topic types offer the possibility to work over RTPS without the restrictions related to the IDLs. Us-

ing them, the users can declare the different types that they need and manage the information directly, avoiding the additional step of updating the IDL file and the generation of *C++* classes.

6.14.1 Overview of Dynamic Types

This section describes the classes related to dynamic types that are used through the rest of the documentation. At the bottom of the section you can also find a short example using the functionality.

Involved classes

The following class diagram describes the relationship among the classes related to dynamic types. Please, refer to the description of each class to find its purpose and the nature of the relationship with the rest of the classes.

Fig. 8: Dynamic types class diagram

- *Dynamic Type*
- *Dynamic Type Builder Factory*
- *Dynamic Type Builder*
- *Type Descriptor*
- *Dynamic Type Member*
- *Member Descriptor*
- *Dynamic Data*
- *Dynamic Data Factory*
- *Dynamic PubSubType*

Dynamic Type

Base class of all types declared dynamically. It represents a dynamic data type that can be used to create *Dynamic Data* values. By design, the structure of a dynamic type (its member fields) cannot be modified once the type is created.

Dynamic Type Builder Factory

Singleton class that is in charge of the creation and the management of every *Dynamic Type* and *Dynamic Type Builder*. It declares functions to create builders for each kind of supported types. Given a builder for a specific type, it can also create the corresponding *Dynamic Type*. Some simpler types can be created directly, avoiding the step of creating a *Dynamic Type Builder*. Please, refer to the *Supported Types* documentation for details about which ones support this option.

Every object created by the factory must be deleted to avoid memory leaking. Refer to the *Memory management* section for details.

Dynamic Type Builder

Intermediate class used to configure a *Dynamic Type* before it is created. By design, the structure of a *Dynamic Type* (its member fields) cannot be modified once the object is created. Therefore, all its structure must be defined prior to its creation. The builder is the object used to set up this structure.

Once defined, the *Dynamic Type Builder Factory* is used to create the *Dynamic Type* from the information contained in the builder. As a shortcut, the builder exposes a function `build` that internally uses the *Dynamic Type Builder Factory* to return a fully constructed *Dynamic Type*. The types created with `build` are still subject to the *Memory management* restrictions, and must be deleted by the *Dynamic Type Builder Factory*.

Builders can be reused after the creation of a *Dynamic Type*, as the changes applied to the builder do not affect to types created previously.

Type Descriptor

Stores the information about one type with its relationships and restrictions. This is the class that describes the inner structure of a *Dynamic Type*. The *Dynamic Type Builder* has an internal instance of `TypeDescriptor` that modifies during the type building process. When the *Dynamic Type* is created, the *Dynamic Type Builder Factory* uses the information of the `TypeDescriptor` in the builder to create the *Dynamic Type*. During the creation, the `TypeDescriptor` is copied to the *Dynamic Type*, so that it becomes independent from the *Dynamic Type Builder*, and the builder can be reused for another type.

Dynamic Type Member

Represents a data member of a *Dynamic Type* that is also a *Dynamic Type*. Compound types (dynamic types that are composed of other dynamic types) have a `DynamicTypeMember` for every child *Dynamic Type* added to it.

Member Descriptor

Just as a *Type Descriptor* describes the inner structure of a *Dynamic Type*, a `MemberDescriptor` stores all the information needed to manage a *Dynamic Type Member*, like their name, their unique ID, or the default value after the creation. This information is copied to the *Dynamic Data* on its creation.

Dynamic Data

While a *Dynamic Type* *describes* a type, `DynamicData` represents a data instance of a *Dynamic Type*. It provides functions to access and modify the data values in the instance.

There are two ways to work with `DynamicData`:

- Activating the macro `DYNAMIC_TYPES_CHECKING`, which creates a variable for each primitive kind to help the debug process.
- Without this macro, the size of the `DynamicData` is reduced, using only the minimum needed internal values, but it makes the code harder to debug.

Dynamic Data Factory

Singleton class that is in charge of the creation and the management of every *Dynamic Data*. It can take a *Dynamic Type* and create an instance of a corresponding *Dynamic Data*. Every data object created by the factory must be deleted to avoid memory leaking. Refer to the *Memory management* section for details.

It also allows to create a `TypeIdentifier` and a (Minimal and Complete) `TypeObject` from a `TypeDescriptor`.

Dynamic PubSubType

This class is an adapter that allows using *Dynamic Data* on Fast DDS. It inherits from `TopicDataType` and implements the functions needed to communicate the *Dynamic Data* between Publishers and Subscribers.

Minimum example

This is a short example to illustrate the use of the dynamic types and how the classes describe above interact with each other. While the code snippet can be used as a quick reference for code building, the sequence diagram below provides a visual interpretation of the actions.

```
// Create a builder for a specific type
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪enum_builder();

// Use the builder to configure the type
builder->add_empty_member(0, "DEFAULT");
builder->add_empty_member(1, "FIRST");
builder->add_empty_member(2, "SECOND");

// Create the data type using the builder
// The builder will internally use the DynamicTypeBuilderFactory to create the type
DynamicType_ptr type = builder->build();

// Create a new data instance of the create data type
DynamicData_ptr data (DynamicDataFactory::get_instance()->create_data(type));

// Now we can set or read data values
data->set_int32_value(1);

// No need of deleting the objects, since we used the
// automanaged smart pointers
```

Fig. 9: Sequence diagram of the code above

6.14.2 Supported Types

In order to provide maximum flexibility and capability to the defined dynamic types, eProsima Fast DDS supports several member types, ranging from simple primitives to nested structures.

This section describes the basic (not nested) supported types. For more complex structures and examples, please, refer to *Complex Types*.

- *Primitive Types*
- *String and WString*
- *Alias*
- *Enumeration*
- *Bitmask*
- *Structure*
- *Bitset*
- *Union*
- *Sequence*
- *Array*
- *Map*

Primitive Types

This section includes every simple kind:

BOOLEAN	INT64
BYTE	UINT16
CHAR8	UINT32
CHAR16	UINT64
INT16	FLOAT32
INT32	FLOAT64
FLOAT128	

By definition, primitive types are self-described and can be created without configuration parameters. Therefore, `DynamicTypeBuilderFactory` exposes several functions to allow users create the dynamic type avoiding the `DynamicTypeBuilder` step. The `DynamicTypeBuilder` can still be used to create dynamic data of primitive types, as shown on the example below. The `DynamicData` class has a specific `get` and `set` functions for each primitive type of the list.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_int32_builder();
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicData* data = DynamicDataFactory::get_instance()->create_data(created_type);
data->set_int32_value(1);

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
```

(continues on next page)

(continued from previous page)

```
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pType);
data2->set_int32_value(1);
```

String and WString

Strings are pretty similar to primitive types, the main difference being that they need to set the size of the `buffer` that they can manage. By default this size is set to 255 characters.

`DynamicTypeBuilderFactory` exposes the functions `create_string_type` and `create_wstring_type` to allow users create the Dynamic Types avoiding the `DynamicTypeBuilder` step. The `DynamicTypeBuilder` can still be used to create String type dynamic data, as shown on the example below.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_string_builder(100);
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicData* data = DynamicDataFactory::get_instance()->create_data(created_type);
data->set_string_value("Dynamic String");

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_string_
↪type(100);
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pType);
data2->set_string_value("Dynamic String");
```

Alias

Alias types provide an alternative name to an already existing type. Once the `DynamicData` is created, users can access its information as if they were working with the base type.

`DynamicTypeBuilderFactory` exposes the function `create_alias_type` to allow users create the Alias types avoiding the `DynamicTypeBuilder` step. The `DynamicTypeBuilder` can still be used to create Alias, as shown on the example below.

```
// Create the base type
DynamicTypeBuilder_ptr base_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_string_builder(100);
DynamicType_ptr base_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(base_builder.get());

// Create alias using Builders
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪alias_builder(base_type,
                "alias");
DynamicData* data = DynamicDataFactory::get_instance()->create_data(builder.get());
data->set_string_value("Dynamic Alias String");

// Create alias type directly
DynamicType_ptr pAliasType = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪type(base_type, "alias");
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pAliasType);
data2->set_string_value("Dynamic Alias String");
```

Enumeration

An enumeration contains a set of supported values and a selected value among those supported. The supported values must be configured using the `DynamicTypeBuilder`, using the `add_member` function for each supported value. The input to this function is the index and the name of the value we want to add.

The `DynamicData` class has functions `get_enum_value` and `set_enum_value` to work with value index or value name name strings.

```
// Add enumeration values using the DynamicTypeBuilder
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪enum_builder();
builder->add_empty_member(0, "DEFAULT");
builder->add_empty_member(1, "FIRST");
builder->add_empty_member(2, "SECOND");

// Create the data instance
DynamicData* data = DynamicDataFactory::get_instance()->create_data(builder.get());

// Access value using the name
std::string sValue = "SECOND";
data->set_enum_value(sValue);
std::string sStoredValue;
data->get_enum_value(sStoredValue, MEMBER_ID_INVALID);

// Access value using the index
uint32_t uValue = 2;
data->set_enum_value(uValue);
uint32_t uStoredValue;
data->get_enum_value(uStoredValue, MEMBER_ID_INVALID);
```

Bitmask

Bitmasks are similar to *enumeration* types, but their members work as bit flags that can be individually turned on and off. Bit operations can be applied when testing or setting a bitmask value. `DynamicData` has the special functions `get_bitmask_value` and `set_bitmask_value` which allow to retrieve or modify the full value instead of accessing each bit.

Bitmasks can be bound to any number of bits up to 64.

```
uint32_t limit = 5; // Stores as "octet"

// Add bitmask flags using the DynamicTypeBuilder
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪bitmask_builder(limit);
builder->add_empty_member(0, "FIRST");
builder->add_empty_member(1, "SECOND");

// Create the data instance
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(builder.get()));

// Access the mask values using the name
data->set_bool_value(true, "FIRST");                // Set the "FIRST" bit
bool bSecondValue = data->get_bool_value("SECOND"); // Get the "SECOND" bit

// Access the mask values using the index
```

(continues on next page)

(continued from previous page)

```
data->set_bool_value(true, 1);                        // Set the "SECOND" bit
bool bFirstValue = data->get_bool_value(0);           // Get the "FIRST" bit

// Get the complete bitmask as integer
uint64_t fullValue;
data->get_bitmask_value(fullValue);
```

Structure

Structures are the common complex types, they allow to add any kind of members inside them. They don't have any value, they are only used to contain other types.

To manage the types inside the structure, users can call the `get` and `set` functions according to the kind of the type inside the structure using their `ids`. If the structure contains a complex value, it should be used with `loan_value` to access to it and `return_loaned_value` to release that pointer. `DynamicData` manages the counter of loaned values and users can't loan a value that has been loaned previously without calling `return_loaned_value` before.

The `Ids` must be consecutive starting by zero, and the `DynamicType` will change that Id if it doesn't match with the next value. If two members have the same Id, after adding the second one, the previous will change its Id to the next value. To get the Id of a member by name, `DynamicData` exposes the function `get_member_id_by_name`.

```
// Build a structure with two fields ("first" as int32, "other" as uint64) using␣
↪DynamicTypeBuilder
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type());
builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->create_
↪uint64_type());
DynamicType_ptr struct_type(builder->build());

// Create the data instance
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(struct_type));

// Access struct members
data->set_int32_value(5, 0);
data->set_uint64_value(13, 1);
```

Structures allow inheritance, exactly with the same OOP meaning. To inherit from another structure, we must create the structure calling the `create_child_struct_builder` of the factory. This function is shared with bitsets and will deduce our type depending on the parent's type.

```
DynamicTypeBuilder_ptr child_builder =
        DynamicTypeBuilderFactory::get_instance()->create_child_struct_
↪builder(builder.get());
```

Bitset

Bitset types are similar to *structure* types, but their members are merely *bitfields*, which are stored optimally. In the static version of bitsets, each bit uses just one bit in memory (with platform limitations) without alignment considerations. A bitfield can be anonymous (cannot be addressed) to skip unused bits within a bitset.

Each bitfield in a bitset can be modified through their minimal needed primitive representation.

Number of bits	Primitive
1	BOOLEAN
2-8	UINT8
9-16	UINT16
17-32	UINT32
33-64	UINT64

Each bitfield (or member) works like its primitive type with the only difference that the internal storage only modifies the involved bits instead of the full primitive value.

Bit_bound and position of the bitfield can be set using annotations (useful when converting between static and dynamic bitsets).

```
// Create bitfields with the appropriate type for their size
DynamicTypeBuilder_ptr base_type_byte_builder = DynamicTypeBuilderFactory::get_
↪instance()->create_byte_builder();
auto base_type_byte = base_type_byte_builder->build();

DynamicTypeBuilder_ptr base_type_uint32_builder = DynamicTypeBuilderFactory::get_
↪instance()->create_uint32_builder();
auto base_type_uint32 = base_type_uint32_builder->build();

// Create the bitset with two bitfields
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪bitset_builder();
builder->add_member(0, "byte", base_type_byte);
builder->add_member(1, "uint32", base_type_uint32);

// Apply members' annotations
builder->apply_annotation_to_member(0, ANNOTATION_POSITION_ID, "value", "0");   //
↪"byte" starts at position 0
builder->apply_annotation_to_member(0, ANNOTATION_BIT_BOUND_ID, "value", "2");  //
↪"byte" is 2 bit length
builder->apply_annotation_to_member(1, ANNOTATION_POSITION_ID, "value", "10");  //
↪"uint32" starts at position 10 (8 bits empty)
builder->apply_annotation_to_member(1, ANNOTATION_BIT_BOUND_ID, "value", "20"); //
↪"uint32" is 20 bits length

// Create the data instance
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(builder.get()));

// Access values
data->set_byte_value(234, 0);
data->set_uint32_value(2340, 1);
octet bValue;
uint32_t uValue;
data->get_byte_value(bValue, 0);
data->get_uint32_value(uValue, 1);
```

Bitsets allows inheritance, exactly with the same OOP meaning. To inherit from another bitset, we must create the bitset calling the `create_child_struct_builder` of the factory. This function is shared with structures and will deduce our type depending on the parent's type.

```
DynamicTypeBuilder_ptr child_builder =
        DynamicTypeBuilderFactory::get_instance()->create_child_struct_
↪builder(builder.get());
```

Union

Unions are a special kind of structures where only one of the members is active at the same time. To control these members, users must set the `discriminator` type that is going to be used to select the current member calling the `create_union_builder` function. The `discriminator` itself is a Dynamic Type of any primitive type, string type or union type.

Every member that is going to be added needs at least one `union_case_index` to set how it is going to be selected and, optionally, if it is the default value of the union.

```
// Create the union DynamicTypeBuilder with an int32 discriminator
DynamicType_ptr discriminator = DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type();
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪union_builder(discriminator);

// Add the union members. "firts" will be the default value
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type(), "", { 0 },
        true);
builder->add_member(0, "second", DynamicTypeBuilderFactory::get_instance()->create_
↪int64_type(), "", { 1 },
        false);

// Create the data instance
DynamicType_ptr union_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(union_type));

// Access the values using the member index
data->set_int32_value(9, 0);
data->set_int64_value(13, 1);

// Get the label of the currently selected member
uint64_t unionLabel;
data->get_union_label(unionLabel);
```

Sequence

A complex type that manages its members as a list of items allowing users to insert, remove or access to a member of the list. To create this type users need to specify the type that it is going to store and optionally the size limit of the list.

To ease the memory management of this type, `DynamicData` has these functions:

- `insert_sequence_data`: Creates a new element at the end of the list and returns the `id` of the new element.

- `remove_sequence_data`: Removes the element of the given index and refreshes the ids to keep the consistency of the list.
- `clear_data`: Removes all the elements of the list.

```
// Create a DynamicTypeBuilder for a sequence of two elements of type inte32
uint32_t length = 2;
DynamicType_ptr base_type = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicTypeBuilder_ptr builder =
        DynamicTypeBuilderFactory::get_instance()->create_sequence_builder(base_type,␣
↪length);

// Create the data instance
DynamicType_ptr sequence_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(sequence_type));

// Insert and remove elements
MemberId newId, newId2;
data->insert_int32_value(10, newId);
data->insert_int32_value(12, newId2);
data->remove_sequence_data(newId);
```

Array

Arrays are pretty similar to sequences with two main differences: they can have multiple dimensions and they do not need their elements to be stored consecutively.

An array needs to know the number of dimensions it is managing. For that, users must provide a vector with as many elements as dimensions in the array. Each element in the vector represents the size of the given dimension. If the value of an element is set to zero, the default value applies (`100`).

Id values on the `set` and `get` functions of `DynamicData` correspond to the array index. To ease the management of array elements, every `set` function in `DynamicData` class creates the item if the given index is empty.

To ease the memory management of this type, `DynamicData` has these functions:

- `insert_array_data`: Creates a new element at the end of the array and returns the `id` of the new element.
- `remove_array_data`: Clears the element of the given index.
- `clear_data`: Removes all the elements of the array.
- `get_array_index`: Returns the position id giving a vector of indexes on every dimension that the arrays support, which is useful in multidimensional arrays.

```
// Create an array DynamicTypeBuilder for a 2x2 elements of type int32
std::vector<uint32_t> lengths = { 2, 2 };
DynamicType_ptr base_type = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicTypeBuilder_ptr builder =
        DynamicTypeBuilderFactory::get_instance()->create_array_builder(base_type,␣
↪lengths);

// Create the data instance
DynamicType_ptr array_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(array_type));

// Access elements in the multidimensional array
```

(continues on next page)

(continued from previous page)

```
MemberId pos = data->get_array_index({1, 0});
data->set_int32_value(11, pos);
data->set_int32_value(27, pos + 1);
data->clear_array_data(pos);
```

Map

Maps contain a list of 'key-value' pair types, allowing users to insert, remove or modify the element types of the map. The main difference with sequences is that the map works with pairs of elements and creates copies of the key element to block the access to these elements.

To create a map, users must set the types of the key and the value elements, and, optionally, the size limit of the map.

To ease the memory management of this type, `DynamicData` has these functions:

- `insert_map_data`: Inserts a new key value pair and returns the ids of the newly created key and value elements.
- `remove_map_data`: Uses the given id to find the key element and removes the key and the value elements from the map.
- `clear_data`: Removes all the elements from the map.

```
// Create DynamicTypeBuilder for a map of two pairs of {key:int32, value:int32}
uint32_t length = 2;
DynamicType_ptr base = DynamicTypeBuilderFactory::get_instance()->create_int32_type();
DynamicTypeBuilder_ptr builder =
        DynamicTypeBuilderFactory::get_instance()->create_map_builder(base, base,␣
↪length);

// Create the data instance
DynamicType_ptr map_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(map_type));

// Add a new element to the map with key 1
DynamicData_ptr key(DynamicDataFactory::get_instance()->create_data(base));
MemberId keyId;
MemberId valueId;
key->set_int32_value(1);
data->insert_map_data(key.get(), keyId, valueId);

// Add a new element to the map with key 2
// insert_map_data creates a copy of the key, so the same instance can be reused
MemberId keyId2;
MemberId valueId2;
key->set_int32_value(2);
data->insert_map_data(key.get(), keyId2, valueId2);

// Set the value to the element with key 2, using the returned value Id
data->set_int32_value(53, valueId2);

// Remove elements from the map
data->remove_map_data(keyId);
data->remove_map_data(keyId2);
```

6.14.3 Complex Types

If the application's data model is complex, it is possible to combine the *basic types* to create complex types, including nested composed types (structures within structures within unions). Types can also be extended using inheritance, improving the flexibility of the definition of the data types to fit the model.

The following subsections describe these *complex types* and their use.

- *Nested structures*
- *Structure inheritance*
- *Alias of an alias*
- *Unions with complex types*

Nested structures

Structures can contain other structures as members. The access to these compound members is restricted and managed by the `DynamicData` instance. Users must request access calling `loan_value` before using them, and release them with `return_loaned_value` once they finished. The loan operation will fail if the member is already loaned and has not been released yet.

```
// Create a struct type
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type());
builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->create_
↪uint64_type());
DynamicType_ptr struct_type = builder->build();

// Create a struct type with the previous struct as member
DynamicTypeBuilder_ptr parent_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_struct_builder();
parent_builder->add_member(0, "child_struct", struct_type);
parent_builder->add_member(1, "second", DynamicTypeBuilderFactory::get_instance()->
↪create_int32_type());
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(parent_builder.
↪get()));

// Access the child struct with the loan operations
DynamicData* child_data = data->loan_value(0);
child_data->set_int32_value(5, 0);
child_data->set_uint64_value(13, 1);
data->return_loaned_value(child_data);
```

Structure inheritance

To inherit a structure from another one, use the `create_child_struct_type` function from `DynamicTypeBuilderFactory`. The resultant type contains all members from the base class and the new ones added to the child.

Structures support several levels of inheritance, so the base class can be another derived type itself.

```
// Create a base struct type
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type());
builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->create_
↪uint64_type());

// Create a struct type derived from the previous struct
DynamicTypeBuilder_ptr child_builder =
        DynamicTypeBuilderFactory::get_instance()->create_child_struct_
↪builder(builder.get());

// Add new members to the derived type
builder->add_member(2, "third", DynamicTypeBuilderFactory::get_instance()->create_
↪uint64_type());

// Create the data instance
DynamicType_ptr struct_type = child_builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(struct_type));

// The derived type includes the members defined on the base type
data->set_int32_value(5, 0);
data->set_uint64_value(13, 1);
data->set_uint64_value(47, 2);
```

Alias of an alias

Alias types support recursion, simply use an alias name as base type for `create_alias_type`.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_string_builder(100);
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪alias_builder(
    created_builder.get(), "alias");
DynamicTypeBuilder_ptr builder2 = DynamicTypeBuilderFactory::get_instance()->create_
↪alias_builder(
    builder.get(), "alias2");
DynamicData* data(DynamicDataFactory::get_instance()->create_data(builder2->build()));
data->set_string_value("Dynamic Alias 2 String");

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_string_
↪type(100);
DynamicType_ptr pAliasType = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪type(pType, "alias");
```

(continues on next page)

(continued from previous page)

```
DynamicType_ptr pAliasType2 =
        DynamicTypeBuilderFactory::get_instance()->create_alias_type(pAliasType,
↪"alias2");
DynamicData* data2(DynamicDataFactory::get_instance()->create_data(pAliasType));
data2->set_string_value("Dynamic Alias 2 String");
```

Unions with complex types

Unions support complex type fields. The access to these complex type fields is restricted and managed by the `DynamicData` instance. Users must request access calling `loan_value` before using them, and release them with `return_loaned_value` once they finished. The loan operation will fail if the fields is already loaned and has not been released yet.

```
// Create a union DynamicTypeBuilder
DynamicType_ptr discriminator = DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type();
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪union_builder(discriminator);

// Add a int32 to the union
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type(), "", { 0 },
        true);

// Create a struct type and add it to the union
DynamicTypeBuilder_ptr struct_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_struct_builder();
struct_builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->
↪create_int32_type());
struct_builder->add_member(1, "other", DynamicTypeBuilderFactory::get_instance()->
↪create_uint64_type());
builder->add_member(1, "first", struct_builder.get(), "", { 1 }, false);

// Create the union data instance
DynamicType_ptr union_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(union_type));

// Access the struct member using the loan operations
DynamicData* child_data = data->loan_value(1);
child_data->set_int32_value(9, 0);
child_data->set_int64_value(13, 1);
data->return_loaned_value(child_data);
```

6.14.4 Annotations

Dynamic Type Builder allows applying an annotation to both current type and inner members with the functions:

- `apply_annotation`
- `apply_annotation_to_member`

Both functions take the name, the key and the value of the annotation. `apply_annotation_to_member` additionally receives the `MemberId` of the inner member.

For example, if we define an annotation like:

```
@annotation MyAnnotation
{
    long value;
    string name;
};
```

And then we apply it through IDL to a struct:

```
@MyAnnotation(5, "length")
struct MyStruct
{
...
```

The equivalent code using DynamicTypes will be:

```
// Apply the annotation
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪struct_builder();
//...
builder->apply_annotation("MyAnnotation", "value", "5");
builder->apply_annotation("MyAnnotation", "name", "length");
```

Builtin annotations

The following annotations modifies the behavior of DynamicTypes:

- `@position`: When applied to *Bitmask*, sets the position of the flag,
  as expected in the IDL annotation.
  If applied to *Bitset*, sets the base position of the bitfield,
  useful to identify unassigned bits.
- `@bit_bound`: Applies to *Bitset*. Sets the size in bits of the bitfield.
- `@key`: Alias for `@Key`. See *Topics and Keys* section for more details.
- `@default`: Sets a default value for the member.
- `@non_serialized`: Excludes a member from being serialized.

6.14.5 Dynamic Types Discovery and Endpoint Matching

When using Dynamic Types support, Fast DDS checks the optional *TypeObject* and *TypeIdentifier* values during endpoint matching. Currently, the matching only verifies that both endpoints are using the same topic data type, but will not negotiate about it.

The process of checking the types is as follows:

- It checks `CompleteTypeObject` on *TypeObject* first.
- If one or both endpoints do not define the `CompleteTypeObject`, it tries with `MinimalTypeObject`.
- If one or both endpoints do not define `MinimalTypeObject` either, it compares the *TypeIdentifier*.
- If none is defined, then just the type name is checked.

If one of the endpoints transmits a `CompleteTypeObject`, *Discovery-Time Data Typing* can be performed.

TypeObject

TypeObject fully describes a data type, the same way as the IDL representation does. There are two kinds of *TypeObject*: `CompleteTypeObject` and `MinimalTypeObject` .

- `CompleteTypeObject` fully describes the type, the same way as the IDL representation does.
- `MinimalTypeObject` is a compact representation of the data type, that contains only the information relevant for the remote Endpoint to be able to interpret the data.

TypeObject is an IDL union with both *Minimal* and *Complete* representation. Both are described in the annexes of DDS-XTypes V1.2 document, please refer to this document for details.

TypeInformation

TypeInformation is an extension of *XTypes 1.2* that allow Endpoints to share information about data types without sending the *TypeObject*. Endpoints instead share a *TypeInformation* containing the *TypeIdentifier* of the data type. Then each Endpoint can request the complete *TypeObject* for the data types it is interested in. This avoids sending the complete data type to Endpoints that may not be interested.

TypeIdentifier is described in the annexes of DDS-XTypes V1.2 document, please refer to this document for details.

TypeIdentifier

TypeIdentifier provides a unique way to identify each type. For basic types, the information contained in the *TypeIdentifier* completely describes the type, while for complex ones, it serves as a search key to retrieve the complete *TypeObject*.

TypeIdentifier is described in the annexes of DDS-XTypes V1.2 document, please refer to this document for details.

TypeObjectFactory

Singleton class that manages the creation and access for every registered *TypeObject* and *TypeIdentifier*. It can generate a full *Dynamic Type* from a basic *TypeIdentifier* (i.e., one whose discriminator isn't `EK_MINIMAL` or `EK_COMPLETE`).

Fast DDS-Gen

Fast DDS-Gen supports the generation of `XXXTypeObject.h` and `XXXTypeObject.cxx` files, taking `XXX` as our IDL type. These files provide a small Type Factory for the type `XXX`. Generally, these files are not used directly, as now the type `XXX` will register itself through its factory to *TypeObjectFactory* in its constructor, making it very easy to use static types with dynamic types.

Discovery-Time Data Typing

Using the Fast DDS API, when a participant discovers a remote endpoint that sends a complete *TypeObject* or a simple *TypeIdentifier* describing a type that the participant does not know, the participant listener's function *`on_type_discovery`* is called with the received *TypeObject* or *TypeIdentifier*, and, when possible, a pointer to a *Dynamic Type* ready to be used.

Discovery-Time Data Typing allows the discovering of simple *DynamicTypes*. A *TypeObject* that depends on other TypeObjects, cannot be built locally using Discovery-Time Data Typing and should use *TypeLookup Service* instead.

To ease the sharing of the *TypeObject* and *TypeIdentifier* used by Discovery-Time Data Typing, *TopicDataType* contains a data member named `auto_fill_type_object`. If set to true, the local participant will send the *TypeObject* and *TypeIdentifier* to the remote endpoint during discovery.

TypeLookup Service

Using the Fast DDS API, when a participant discovers an endpoint that sends a type information describing a type that the participant doesn't know, the participant listener's function `on_type_information_received` is called with the received TypeInformation. The user can then try to retrieve the full TypeObject hierarchy to build the remote type locally, using the TypeLookup Service.

To enable this builtin TypeLookup Service, the user must enable it in the *QoS* of the *DomainParticipant*:

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

A participant can be enabled to act as a TypeLookup server, client, or both.

The process of retrieving the remote type from its TypeInformation, and then registering it, can be simplified using the `register_remote_type` function on the *DomainParticipant*. This function takes the name of the type, the type information, and a callback function. Internally it uses the TypeLookup Service to retrieve the full *TypeObject*, and, if successful, it will call the callback.

This callback has the following signature:

```
void(std::string& type_name, const DynamicType_ptr type)
```

- **type_name**: Is the name given to the type when calling `register_remote_type`, to allow the same callback to be used across different calls.
- **type**: If the `register_remote_type` was able to build and register a *Dynamic Type*, this parameter contains a pointer to the type. Otherwise it contains `nullptr`. In the latter case, the user can still try to build the type manually using the factories, but it is very likely that the build process will fail.

TopicDataType contains a data member named `auto_fill_type_information`. If set to true, the local participant will send the type information to the remote endpoint during discovery.

6.14.6 Serialization

Dynamic Types have their own `pubsub` type like any class generated with an IDL, and their management is pretty similar to them.

```
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicPubSubType pubsubType(pType);

// SERIALIZATION EXAMPLE
DynamicData* pData = DynamicDataFactory::get_instance()->create_data(pType);
uint32_t payloadSize = static_cast<uint32_t>(pubsubType.
↪getSerializedSizeProvider(pData)());
SerializedPayload_t payload(payloadSize);
pubsubType.serialize(pData, &payload);

// DESERIALIZATION EXAMPLE
types::DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pType);
pubsubType.deserialize(&payload, data2);
```

A member can be marked to be ignored by serialization with the annotation `@non_serialized`.

6.14.7 XML profiles

XML Dynamic Types allows eProsima Fast DDS to create Dynamic Types directly defining them through XML. This allows any application to change `TopicDataTypes` without the need to change its source code.

Please, refer to *XML Dynamic Types* for further information about how to use this feature.

6.14.8 Memory management

Memory management is critical for Dynamic Types, because every dynamic type and dynamic data is managed with pointers. Every object stored inside of a dynamic object is managed by its owner, and users must delete every object they create using the factories.

```
DynamicTypeBuilder* pBuilder = DynamicTypeBuilderFactory::get_instance()->create_
↪uint32_builder();
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicData* pData = DynamicDataFactory::get_instance()->create_data(pType);

DynamicTypeBuilderFactory::get_instance()->delete_builder(pBuilder);
DynamicDataFactory::get_instance()->delete_data(pData);
```

To ease this management, the library defines smart pointers (`DynamicTypeBuilder_ptr`, `DynamicType` and `DynamicData_ptr`) that will delete the objects automatically when they are not needed anymore. `DynamicType` will always be returned as `DynamicType_ptr` because there is no internal management of its memory.

```
DynamicTypeBuilder_ptr pBuilder = DynamicTypeBuilderFactory::get_instance()->create_
↪uint32_builder();
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicData_ptr pData(DynamicDataFactory::get_instance()->create_data(pType));
```

The only case where these smart pointers cannot be used is with functions `loan_value` and `return_loaned_value`. Raw pointers should be used with these functions, because the returned value should not be deleted, and using a smart pointer with them will cause a crash.

6.14.9 Dynamic HelloWorld Examples

These are complete working examples that make use of dynamic types. You can explore them to find how this feature connects to the rest of Fast DDS, and learn how to integrate it in your own application.

DynamicHelloWorldExample

This example is located in folder `examples/C++/DynamicHelloWorldExample`. It shows the use of DynamicType generation to provide the TopicDataType. This example is compatible with the classic HelloWorldExample.

As a quick reference, the following piece of code shows how the HelloWorld type is created using DynamicTypes:

```
// In HelloWorldPublisher.h
// Dynamic Types
eprosima::fastrtps::types::DynamicData* m_DynHello;
eprosima::fastrtps::types::DynamicPubSubType m_DynType;

// In HelloWorldPublisher.cpp
// Create basic builders
DynamicTypeBuilder_ptr struct_type_builder(DynamicTypeBuilderFactory::get_instance()->
↪create_struct_builder());

// Add members to the struct.
struct_type_builder->add_member(0, "index", DynamicTypeBuilderFactory::get_instance()-
↪>create_uint32_type());
struct_type_builder->add_member(1, "message", DynamicTypeBuilderFactory::get_
↪instance()->create_string_type());
struct_type_builder->set_name("HelloWorld");

DynamicType_ptr dynType = struct_type_builder->build();
m_DynType.SetDynamicType(dynType);
m_DynHello = DynamicDataFactory::get_instance()->create_data(dynType);
m_DynHello->set_uint32_value(0, 0);
m_DynHello->set_string_value("HelloWorld", 1);
```

DDSDynamicHelloWorldExample

This example uses the DDS API, and it is located in the `examples/C++/DDS/DynamicHelloWorldExample` folder. It shows a publisher that loads a type from an XML file, and shares it during discovery. The subscriber discovers the type using *Discovery-Time Data Typing*, and registers the discovered type on the `on_type_discovery` listener function.

TypeLookupService

This example uses the DDS API, and it is located in the `examples/C++/DDS/TypeLookupService` folder. It's very similar to DDSDynamicHelloWorldExample, but the shared type is complex enough to require the TypeLookup Service due to the dependency of inner struct types. Specifically, it uses the `register_remote_type` approach with a callback.

6.15 Typical Use-Cases

The use of Fast-RTPS is highly varied, allowing a large number of configurations depending on the scenario in which the library is applied. This section provides configuration examples for the typical use cases arousing when dealing with distributed systems. It is organized as follows:

- *Fast-RTPS over WIFI*. Presents the case of using Fast-RTPS in scenarios where discovery through multicast communication is a challenge. To address this problem, the use of an initial peers list by which the address-port pairs of the remote participants are defined is presented (See *Initial Peers*). Furthermore, it specifies how to disable the multicast discovery mechanism (See *Disabling multicast discovery*).

- *Wide Deployments*. Describes the recommended configurations for using Fast-RTPS in environments with a high number of deployed communicating agents. These are the use of a centralized server for the discovery phases (See *Server-Client Discovery*), and the Fast-RTPS' STATIC discovery mechanism for well known network topologies (See *Well Known Network Topologies*).
- *Fast-RTPS in ROS 2*. Since Fast-RTPS is the default middleware implementation in the OSRF Robot Operation System 2 (ROS 2), this tutorial is an explanation of how to take full advantage of Fast-RTPS wide set of capabilities in a ROS 2 project.

6.15.1 Fast-RTPS over WIFI

The RTPS 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 could be a problem in case the communication is not wired, i.e. WiFi communication, since multicast is not as reliable over WiFi as it is over ethernet. Fast-RTPS' solution to this challenge is to define the participants with which a unicast communication is to be set, i.e an initial list of remote peers.

Initial Peers

According to the RTPS standard (Section 9.6.1.1), each participant must listen for incoming PDP discovery metatraffic in two different ports, one linked with a multicast address, and another one linked to a unicast address. Fast-RTPS allows for the configuration of an initial peers list which contains one or more such address-port pairs corresponding to remote participants PDP discovery listening resources, so that the local participant will not only send its PDP traffic to the default multicast address-port specified by its domain, but also to all the address-port pairs specified in the *Initial peers* list.

A participant's initial peers list contains the list of address-port pairs of all other participants with which it will communicate. It is a list of addresses that a participant 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 participants' discovery traffic unicast listening ports are calculated using the following equation: 7400 + 250 * *domainID* + 10 + 2 * *participantID*. Thus, if for example a participant 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.

The following constitutes an example configuring an Initial Peers list with one peer on host 192.168.10.13 with participant ID 1 in domain 0.

C++

```
Locator_t initial_peers_locator;
IPLocator::setIPv4(initial_peers_locator, "192.168.10.13");
initial_peers_locator.port = 7412;
participant_attr.rtps.builtin.initialPeersList.push_back(initial_peers_locator);
```

XML

```
<participant profile_name="initial_peers_example_profile" is_default_profile="true">
    <rtps>
        <builtin>
            <initialPeersList>
                <locator>
                    <udpv4>
                        <address>192.168.10.13</address>
                        <port>7412</port>
                    </udpv4>
                </locator>
            </initialPeersList>
        </builtin>
    </rtps>
</participant>
```

Disabling multicast discovery

If all the peers are known beforehand, it is possible to disable the multicast meta traffic completely. This is done using the configuration attribute `metatrafficUnicastLocatorList`. By defining a custom `metatrafficUnicastLocatorList`, the default metatraffic multicast and unicast locators to be employed by the participant are avoided, which prevents the participant from listening to any discovery data from multicast sources. The local participant creates a meta traffic receiving resource per address-port pair specified in the `metatrafficUnicastLocatorList`.

Consideration should be given to the assignment of the address-port pair 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.

C++

```
    Locator_t meta_unicast_locator;
    IPLocator::setIPv4(meta_unicast_locator, "192.168.10.13");
    meta_unicast_locator.port = 7412;
    participant_attr.rtps.builtin.metatrafficUnicastLocatorList.push_back(meta_
↪unicast_locator);
```

XML

```
<participant profile_name="initial_peers_multicast_avoidance" is_default_profile=
↪"true" >
    <rtps>
        <builtin>
            <!-- Choosing a specific unicast address -->
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.10.13</address>
                        <port>7412</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>
```

6.15.2 Wide Deployments

Systems with large amounts of communication nodes might pose a challenge to Data Distribution Service (DDS) based middleware implementations in terms of setup times, memory consumption, and network load. This is because, as explained in *Discovery*, the Participant Discovery Phase (PDP) relies on meta traffic announcements sent to multicast addresses so that all the participants in the network can acknowledge each other. This phase is followed by a Endpoint Discovery Phase (EDP) where all the participants exchange information (using unicast addresses) about their publisher and subscriber entities with the rest of the participants, so that matching between publishers and subscribers using the same topic can occur. As the number of participants, publishers, and subscribers increases, the meta-traffic, as well as the number of connections, increases exponentially, severely affecting the setup time and memory consumption. Fast-RTPS provides extra features that expand the DDS standard to adapt it to wide deployment scenarios.

Feature	Purpose
Server-Client Discovery Mechanism	This feature is intended to substitute the standard SPDP and SEDP protocols with a discovery based on a server-client architecture, where all the meta-traffic goes through a hub (server) to be distributed throughout the network communication nodes.
Static Discovery	With this feature, the user can manually specify which participant should communicate with which one and through which address and port. Furthermore, the the user can specify which publisher/subscriber matches with which one, thus eliminating all EDP meta traffic.

Server-Client Discovery

Considering a scenario in which a large number of communication agents, called participants in this case, are deployed, an alternative to the default RTPS standard SIMPLE discovery mechanism may be used. For this purpose, FastRTPS provides a client-server discovery mechanism, in which a server participant operates as the central point of communication, that is the server collects and processes the metatraffic sent by the client participants, and distributes the appropriate information among the rest of the clients.

Various discovery server use cases are presented below.

UDPv4 example setup

To configure the client-server discovery 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:

- **Prefix**: This is the unique identifier of the server.
- **Address-port pair**: Specifies the IP address and port of the machine that implements the server. The port is a random number that can be replaced with any other value. Consideration should be given to the assignment of the address-port pair in the `metatrafficUnicastLocatorList`, avoiding the assignment of ports that are not available. Thus using RTPS standard ports is discouraged.

SERVER

C++

```
    Locator_t server_locator;
    IPLocator::setIPv4(server_locator,
↪"192.168.10.57");
    server_locator.port = 56542;

    participant_attr.rtps.builtin.
↪discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::SERVER;
    participant_attr.rtps.ReadguidPrefix(
↪"72.61.73.70.66.61.72.6d.74.65.73.74");
    participant_attr.rtps.builtin.
↪metatrafficUnicastLocatorList.push_
↪back(server_locator);
    participant_attr.rtps.builtin.
↪discovery_config.discoveryServer_
↪client_syncperiod = Duration_t(0,␣
↪250000000);
```

XML

```
<participant profile_name="UDP SERVER"␣
↪is_default_profile="true">
    <rtps>
        <prefix>72.61.73.70.66.61.72.6d.
↪74.65.73.74</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER
↪</discoveryProtocol>
            </discovery_config>

↪<metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.
↪10.57</address>
                        <port>56542</
↪port>
                    </udpv4>
                </locator>
            </
↪metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
</participant>
```

CLIENT

C++

```
    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(
↪"72.61.73.70.66.61.72.6d.74.65.73.74");
    remote_server_attr.
↪metatrafficUnicastLocatorList.push_
↪back(remote_server_locator);

    participant_attr.rtps.builtin.
↪discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::CLIENT;
    participant_attr.rtps.builtin.
↪discovery_config.m_DiscoveryServers.
↪push_back(remote_server_attr);
```

XML

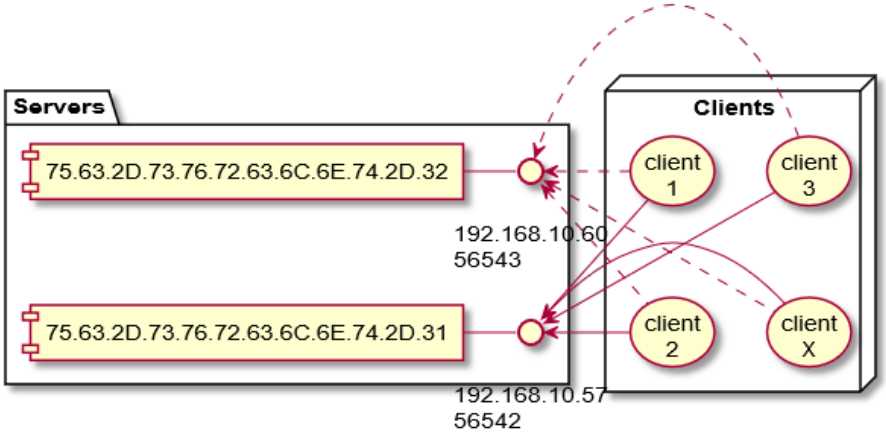
```
<participant profile_name="UDP CLIENT"␣
↪is_default_profile="true">
    <rtps>
        <builtin>
            <discovery_config>
                <discoveryProtocol>CLIENT
↪</discoveryProtocol>
                <discoveryServersList>
                    <RemoteServer prefix=
↪"72.61.73.70.66.61.72.6d.74.65.73.74">

↪<metatrafficUnicastLocatorList>
                            <locator>
                                <udpv4>

↪<address>192.168.10.57</address>

↪<port>56542</port>
                                </udpv4>
                            </locator>
                        </
↪metatrafficUnicastLocatorList>
                    </RemoteServer>
                </discoveryServersList>
            </discovery_config>
        </builtin>
    </rtps>
</participant>
```

UDPv4 redundancy example

The *above example* presents a *single point of failure*, that is, if the *server* fails there is no discovery. In order to prevent this, several servers could be linked to a *client*. By doing this, a discovery failure only takes place if *all servers* fail, which is a more unlikely event.

The following values have been chosen in order to assure each server has a unique **Prefix** and *unicast address*:

Prefix	UDPv4 address
75.63.2D.73.76.72.63.6C.6E.74.2D.32	192.168.10.60:56543
75.63.2D.73.76.72.63.6C.6E.74.2D.31	192.168.10.57:56542



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.

SERVER

C++

```
        Locator_t server_locator_1,␣
↪server_locator_2;

        IPLocator::setIPv4(server_
↪locator_1, "192.168.10.57");
        server_locator_1.port = 56542;
        IPLocator::setIPv4(server_
↪locator_2, "192.168.10.60");
        server_locator_2.port = 56543;

        ParticipantAttributes␣
↪participant_attr_1, participant_attr_2;

        participant_attr_1.rtps.builtin.
↪discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::SERVER;
        participant_attr_1.rtps.
↪ReadguidPrefix("75.63.2D.73.76.72.63.
↪6C.6E.74.2D.31");
        participant_attr_1.rtps.builtin.
↪metatrafficUnicastLocatorList.push_
↪back(server_locator_1);

        participant_attr_2.rtps.builtin.
↪discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::SERVER;
        participant_attr_2.rtps.
↪ReadguidPrefix("75.63.2D.73.76.72.63.
↪6C.6E.74.2D.32");
        participant_attr_2.rtps.builtin.
↪metatrafficUnicastLocatorList.push_
↪back(server_locator_2);
```

XML

```
<participant profile_name="UDP SERVER 1">
    <rtps>
        <prefix>75.63.2D.73.76.72.63.6C.
↪6E.74.2D.31</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER
↪</discoveryProtocol>
            </discovery_config>

↪<metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.
↪10.57</address>
                        <port>56542</
↪port>
                    </udpv4>
                </locator>
            </
↪metatrafficUnicastLocatorList>
        </builtin>
```

CLIENT

C++

```
        Locator_t remote_server_locator_
↪1, remote_server_locator_2;

        IPLocator::setIPv4(remote_server_
↪locator_1, "192.168.10.57");
        remote_server_locator.port =␣
↪56542;
        IPLocator::setIPv4(remote_server_
↪locator_2, "192.168.10.60");
        server_locator.port = 56543;

        RemoteServerAttributes remote_
↪server_attr_1, remote_server_attr_2;

        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);
        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);

        participant_attr.rtps.builtin.
↪discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::CLIENT;
        participant_attr.rtps.builtin.
↪discovery_config.m_DiscoveryServers.
↪push_back(remote_server_attr_1);
        participant_attr.rtps.builtin.
↪discovery_config.m_DiscoveryServers.
↪push_back(remote_server_attr_2);
```

XML

```
<participant profile_name="UDP CLIENT␣
↪REDUNDANCY">
    <rtps>
        <builtin>
            <discovery_config>
                <discoveryProtocol>CLIENT
↪</discoveryProtocol>
                <discoveryServersList>
                    <RemoteServer prefix=
↪"75.63.2D.73.76.72.63.6C.6E.74.2D.31">

↪<metatrafficUnicastLocatorList>
                            <locator>
                                <udpv4>

↪<address>192.168.10.57</address>

↪<port>56542</port>
                                </udpv4>
                            </locator>
                        </
↪metatrafficUnicastLocatorList>
```

UDPv4 persistency example

All participants keeps record of all endpoints discovered (other participants, subscribers or publishers). Different kind of participants populate this record with different procedures:

- *clients* receive this information from its *servers*.
- *servers* receive this information from its *clients*.

Given that *servers* used to have many *clients* associated, this is a lengthy process. In case of *server* failure we may be interested in speed up this process when the *server* restarts.

Keep the discovery information in a file synchronize with the *server*'s record fulfills the goal. In order to enable this we must just specify the *discovery protocol* as **BACKUP**.

Once the *server* is created it generates a *server-<GUIDPREFIX>.db* (*exempli gratia server-73-65-72-76-65-72-63-6C-69-65-6E-74.db*) on its process working directory.

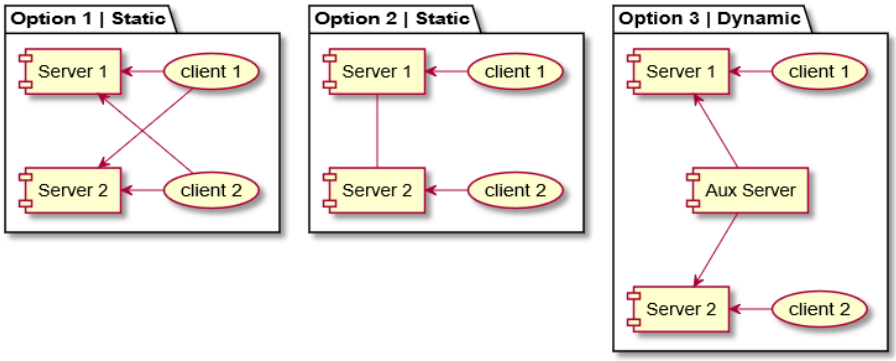
In order to start afresh, that is without deserialize any discovery info, the old backup file must be removed or renamed before launching the server.

UDPv4 partitioning using servers

Server association can be seen as another isolation mechanism besides *domains* and *partitions*. *Clients* that do not share a *server* cannot see each other and belong to isolated server networks. In order to connect server isolated networks we can:

1. Connect each *client* to both *servers*.
2. Connect one *server* to the other.
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 attributes or XML configuration files beforehand. In this regard they match the domain and partition strategy. Option 3 can be implemented at runtime, that is, 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. In this case we consider two servers, each one managing an isolated network:

Network	Prefix	UDPv4 address
A	75.63.2D.73.76.72.69.6C.6E.74.2D.6543	192.168.1.74.2D.6543
B	75.63.2D.73.76.72.69.6C.6E.74.2D.6542	192.168.1.74.2D.6542

In order to communicate both networks we can setup server A to act as client of server B as follows:

C++

```
        Locator_t server_locator, remote_server_locator;

        IPLocator::setIPv4(server_locator, "192.168.10.60");
        server_locator.port = 56543;
        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);

        participant_attr.rtps.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.31");
        participant_attr.rtps.builtin.metatrafficUnicastLocatorList.push_
↪back(server_locator);

        participant_attr.rtps.builtin.discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::SERVER;
        participant_attr.rtps.builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_attr);
```

XML

```
<participant profile_name="UDP SERVER A">
    <rtps>
        <prefix>75.63.2D.73.76.72.63.6C.6E.74.2D.31</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
                <discoveryServersList>
                    <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.32">
                        <metatrafficUnicastLocatorList>
                            <locator>
                                <udpv4>
                                    <address>192.168.10.57</address>
                                    <port>56542</port>
                                </udpv4>
                            </locator>
                        </metatrafficUnicastLocatorList>
                    </RemoteServer>
                </discoveryServersList>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>192.168.10.60</address>
                        <port>56543</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
    </builtin>
    </rtps>
</participant>
```

Option 3

Create a new *server* linked to the *servers* to which the clients are connected. In this case we have two isolated networks A and B, which may be up and running, and join them with a server C.

Server	Prefix	UDPv4 address
A	75.63.2D.73.76.72	192.168.1.74:2653
B	75.63.2D.73.76.72	192.168.1.75:2652
C	75.63.2D.73.76.72	192.168.1.54:2651

In order to communicate both networks we can setup server C to act as client of servers A and B as follows:

C++

```
        Locator_t server_locator, remote_server_locator_A, remote_server_locator_B;

        IPLocator::setIPv4(server_locator, "192.168.10.54");
        server_locator.port = 56541;
        IPLocator::setIPv4(remote_server_locator_A, "192.168.10.60");
        remote_server_locator_A.port = 56543;
        IPLocator::setIPv4(remote_server_locator_B, "192.168.10.57");
        remote_server_locator_B.port = 56542;

        RemoteServerAttributes remote_server_attr_A, remote_server_attr_B;
        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);
        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);

        participant_attr.rtps.ReadguidPrefix("75.63.2D.73.76.72.63.6C.6E.74.2D.33");
        participant_attr.rtps.builtin.metatrafficUnicastLocatorList.push_
↪back(server_locator);

        participant_attr.rtps.builtin.discovery_config.discoveryProtocol =␣
↪DiscoveryProtocol_t::SERVER;
        participant_attr.rtps.builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_attr_A);
        participant_attr.rtps.builtin.discovery_config.m_DiscoveryServers.push_
↪back(remote_server_attr_B);
```

XML

```
<participant profile_name="UDP SERVER C">
    <rtps>
        <prefix>75.63.2D.73.76.72.63.6C.6E.74.2D.33</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
                <discoveryServersList>
                    <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.32">
                        <metatrafficUnicastLocatorList>
                            <locator>
                                <udpv4>
                                    <address>192.168.10.57</address>
                                    <port>56542</port>
                                </udpv4>
                            </locator>
                        </metatrafficUnicastLocatorList>
                    </RemoteServer>
                    <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.31">
                        <metatrafficUnicastLocatorList>
                            <locator>
                                <udpv4>
                                    <address>192.168.10.60</address>
                                    <port>56543</port>
                                </udpv4>
                            </locator>
                        </metatrafficUnicastLocatorList>
                    </RemoteServer>
                </discoveryServersList>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
```

Well Known Network Topologies

It is often the case in industrial deployments, such as productions lines, that the entire network topology (hosts, IP addresses, etc.) is known beforehand. Such scenarios are perfect candidates for Fast-RTPS 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. As explained in the *Discovery* section, all Fast-RTPS discovery mechanisms consist of two steps: PDP and EDP.

Peer-to-Peer Participant Discovery Phase

By default, Fast-RTPS uses SPDP protocol for the PDP phase. This entails the participants sending periodic PDP announcements over a well known multicast addresses, using IP ports calculated from the domain. For large deployments, this can result in quite some meta traffic, since whenever a participant receives a PDP message via multicast, it replies to the remote participant using an address and port specified in the message. In this scenario the number of PDP connections is *N * (N - 1)*, with *N* being the number of participants in the network.

However, it is often the case that not all the participants need to be aware of all the rest of the remote participants present in the network. For limiting all this PDP meta traffic, Fast-RTPS participants can be configured to send their PDP announcements only to the remote participants to which they are required to connect. This is done by specifying a list of peers as a set of IP address-port pairs, and by disabling the participant multicast announcements. Use-case *Fast-RTPS over WIFI* provides a detailed explanation on how to configure Fast-RTPS for such case.

STATIC Endpoint Discovery Phase

As explained in *STATIC Discovery Settings*, the EDP meta traffic can be completely avoided by specifying the EDP discovery using XML files. This way, the user can manually configure which publisher/subscriber matches with which one, so they can start sharing user data right away. To do that, a STATIC discovery XML file must be supplied to the local entity describing the configuration of the remote entity. In this example, a publisher in topic `HelloWorldTopic` from participant `HelloWorldPublisher` is matched with a subscriber from participant `HelloWorldSubscriber`. A fully functional example implementing STATIC EDP is STATIC EDP example.

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 publisher/subscriber attributes must match those defined in the STATIC discovery XML file loaded by the remote entity.

PUBLISHER

C++

```
        // Participant attributes
        participant_attr.rtps.
↪setName("HelloWorldPublisher");
        participant_attr.rtps.
↪builtin.discovery_config.use_
↪SIMPLE_EndpointDiscoveryProtocol␣
↪= false;
        participant_attr.rtps.
↪builtin.discovery_config.use_
↪STATIC_EndpointDiscoveryProtocol␣
↪= true;
        participant_attr.rtps.
↪builtin.discovery_config.
↪setStaticEndpointXMLFilename(
↪"HelloWorldSubscriber.xml");

        // Publisher attributes
        publisher_attr.topic.
↪topicName = "HelloWorldTopic";
        publisher_attr.topic.
↪topicDataType = "HelloWorld";
        publisher_attr.
↪setUserDefinedID(1);
        publisher_attr.
↪setEntityID(2);
```

XML

```
<participant profile_name=
↪"participant_profile_static_pub">
    <rtps>
        <name>HelloWorldPublisher</
↪name>
        <builtin>
            <discovery_config>
                <EDP>STATIC</EDP>

↪<staticEndpointXMLFilename>
↪HelloWorldSubscriber.xml</
↪staticEndpointXMLFilename>
            </discovery_config>
        </builtin>
    </rtps>
</participant>

<publisher profile_name="uc_
↪publisher_xml_conf_static_
↪discovery">
    <topic>
        <name>HelloWorldTopic</name>
        <dataType>HelloWorld</
↪dataType>
    </topic>
    <userDefinedID>1</userDefinedID>
    <entityID>2</entityID>
</publisher>
```

SUBSCRIBER

C++

```
        // Participant attributes
        participant_attr.rtps.
↪setName("HelloWorldSubscriber");
        participant_attr.rtps.
↪builtin.discovery_config.use_
↪SIMPLE_EndpointDiscoveryProtocol␣
↪= false;
        participant_attr.rtps.
↪builtin.discovery_config.use_
↪STATIC_EndpointDiscoveryProtocol␣
↪= true;
        participant_attr.rtps.
↪builtin.discovery_config.
↪setStaticEndpointXMLFilename(
↪"HelloWorldPublisher.xml");

        // Subscriber attributes
        subscriber_attr.topic.
↪topicName = "HelloWorldTopic";
        subscriber_attr.topic.
↪topicDataType = "HelloWorld";
        subscriber_attr.
↪setUserDefinedID(3);
        subscriber_attr.
↪setEntityID(4);
```

XML

```
<participant profile_name=
↪"participant_profile_static_sub">
    <rtps>
        <name>HelloWorldSubscriber</
↪name>
        <builtin>
            <discovery_config>

↪<staticEndpointXMLFilename>
↪HelloWorldPublisher.xml</
↪staticEndpointXMLFilename>
            </discovery_config>
        </builtin>
    </rtps>
</participant>

<subscriber profile_name="uc_
↪subscriber_xml_conf_static_
↪discovery">
    <topic>
        <name>HelloWorldTopic</name>
        <dataType>HelloWorld</
↪dataType>
    </topic>
    <userDefinedID>3</userDefinedID>
    <entityID>4</entityID>
</subscriber>
```

6.15.3 Fast-RTPS in ROS 2

Fast-RTPS is the default middleware implementation in the Open Source Robotic Fundation (OSRF) Robot Operating System ROS 2. This tutorial is an explanation of how to take full advantage of Fast-RTPS wide set of capabilities in a ROS 2 project.

The interface between the ROS2 stack and Fast-RTPS 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-RTPS as middleware layer: `rmw_fastrtps_cpp` and `rmw_fastrtps_dynamic_cpp`. The main difference between the two is that `rmw_fastrtps_dynamic_cpp` uses introspection type support at run time to decide on the serialization/deserialization mechanism, while `rmw_fastrtps_cpp` uses its own type support, which generates the mapping for each message type at build time. The default ROS 2 RMW implementation is `rmw_fastrtps_cpp`. However, it is still possible to select `rmw_fastrtps_dynamic_cpp` using the environment variable `RMW_IMPLEMENTATION`:

1. Exporting `RMW_IMPLEMENTATION` environment variable:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_dynamic_cpp
```

2. When launching your ROS 2 application:

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

Configuring Fast-RTPS with XML files

As described in *XML profiles* section, there are two possibilities for providing Fast-RTPS with XML configuration files:

- **Recommended**: Define the location of the XML configuration file with environment variable `FASTRTPS_DEFAULT_PROFILES_FILE`.

```
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
```

- **Alternative**: Create a *DEFAULT_FASTRTPS_PROFILES.xml* and place it in the same directory as the application executable.

Default profiles

Under ROS 2, the entity creation does not allow for selecting different profiles from the XML. To work around this issue, the profiles can be marked with an attribute `is_default_profile="true"`, so when an entity of that type is created, it will automatically load that profile. The mapping between ROS 2 entities and Fast-RTPS entities is:

ROS entity	Fast-RTPS entity
Node	Participant
Publisher	Publisher
Subscription	Subscriber
Service	Publisher + Subscriber
Client	Publisher + Subscriber

For example, a profile for a ROS 2 `Node` would be specified as:

XML

```
<participant profile_name="participant_profile_ros2" is_default_profile="true">
    <rtps>
        <name>profile_for_ros2_node</name>
    </rtps>
</participant>
```

Configure Publication Mode and History Memory Policy

By default, `rmw_fastrtps` sets some of the Fast-RTPS configurable parameters, ignoring whatever configuration is provided in the XML file. Said parameters, and their default values under ROS 2, are:

Parameter	Description	Default ROS 2 value
History memory policy	Fast-RTPS preallocates memory for the publisher and subscriber histories. When those histories fill up, a reallocation occurs to reserve more memory.	`PREALLOCATED_WITH_REALLOC_MEM`
Publication mode	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`

However, it is possible to fully configure Fast-RTPS (including the history memory policy and the publication mode) using an XML file in combination with environment variable `RMW_FASTRTPS_USE_QOS_FROM_XML`.

```
export FASTRTPS_DEFAULT_PROFILES_FILE=<path_to_xml_file>
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run <package> <application>
```

Example

The following example uses the ROS 2 talker/listener demo, configuring Fast-RTPS to publish synchronously, and to have a dynamically allocated publisher and subscriber histories.

1. Create a XML file *ros_example.xml* and save it in *path/to/xml/*

XML

```
    <publisher profile_name="ros2_publisher_profile" is_default_profile="true">
        <qos>
            <publishMode>
                <kind>SYNCHRONOUS</kind>
            </publishMode>
        </qos>
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </publisher>

    <subscriber profile_name="ros2_subscription_profile" is_default_profile=
↪"true">
        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
    </subscriber>
```

2. Open one terminal and run:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/ros_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp talker
```

3. Open one terminal and run:

```
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export FASTRTPS_DEFAULT_PROFILES_FILE=path/to/xml/ros_example.xml
export RMW_FASTRTPS_USE_QOS_FROM_XML=1
ros2 run demo_nodes_cpp listener
```

6.16 Advanced Functionalities

This section covers slightly more advanced, but useful features that enrich your implementation.

6.16.1 Topics and Keys

The RTPS standard contemplates the use of keys to define multiple data sources/sinks within a single topic.

There are three ways of implementing keys into your topic:

- Defining a *@Key* field in the IDL file when using FastRTPSGen (see the examples that come with the distribution).
- Manually implementing and using a `getKey()` method.
- Adding the attribute *Key* to the member and its parents when using dynamic types (see *Dynamic Topic Types*).

Publishers and Subscribers using topics with keys must be configured to use them, otherwise, they will have no effect:

C++

```
    // Publisher-Subscriber Layer configuration.
    publisher_attr.topic.topicKind = WITH_KEY;
```

XML

```
<publisher profile_name="publisher_profile_qos_key">
    <topic>
        <kind>WITH_KEY</kind>
    </topic>
</publisher>
```

The RTPS Layer requires you to call the `getKey()` method manually within your callbacks.

You can tweak the History to accommodate data from multiple keys based on your current configuration. This consist of defining a maximum number of data sinks and a maximum size for each sink:

C++

```
    // Set the subscriber to remember and store up to 3 different keys.
    subscriber_attr.topic.resourceLimitsQos.max_instances = 3;
    // Hold a maximum of 20 samples per key.
    subscriber_attr.topic.resourceLimitsQos.max_samples_per_instance = 20;
```

XML

```
<subscriber profile_name="subscriber_profile_qos_resourcelimit">
    <topic>
        <resourceLimitsQos>
            <max_instances>3</max_instances>
            <max_samples_per_instance>20</max_samples_per_instance>
        </resourceLimitsQos>
    </topic>
</subscriber>
```

Note that your History must be big enough to accommodate the maximum number of samples for each key. eProsima Fast RTPS will notify you if your History is too small.

6.16.2 Flow Controllers

eProsima Fast RTPS supports user configurable flow controllers on a Publisher and Participant level. These controllers can be used to limit the amount of data to be sent under certain conditions depending on the kind of controller implemented.

The current release implement throughput controllers, which can be used to limit the total message throughput to be sent over the network per time measurement unit. In order to use them, a descriptor must be passed into the Participant or Publisher Attributes.

C++

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

XML

```
<publisher profile_name="publisher_profile_qos_flowcontroller">
    <throughputController>
        <bytesPerPeriod>300000</bytesPerPeriod>
        <periodMillisecs>1000</periodMillisecs>
    </throughputController>
</publisher>
```

In the Writer-Reader layer, the throughput controller is built-in and the descriptor defaults to infinite throughput. To change the values:

```
    WriterAttributes writer_attr;
    writer_attr.throughputController.bytesPerPeriod = 300000; //300kb
    writer_attr.throughputController.periodMillisecs = 1000; //1000ms
```

(continues on next page)

(continued from previous page)

```
//CONF-QOS-PUBLISHMODE
// Allows fragmentation.
publisher_attr.qos.m_publishMode.kind = ASYNCHRONOUS_PUBLISH_MODE;
```

Note that specifying a throughput controller with a size smaller than the socket size can cause messages to never become sent.

6.16.3 Sending large data

The default message size *eProsima Fast RTPS* uses is a conservative value of 65Kb. If your topic data is bigger, it must be fragmented.

Fragmented messages are sent over multiple packets, as understood by the particular transport layer. To make this possible, you must configure the Publisher to work in asynchronous mode.

C++

```
// Allows fragmentation.
publisher_attr.qos.m_publishMode.kind = ASYNCHRONOUS_PUBLISH_MODE;
```

XML

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

In the Writer-Subscriber layer, you have to configure the Writer:

```
WriterAttributes write_attr;
write_attr.mode = ASYNCHRONOUS_WRITER; // Allows fragmentation
```

Note that in best-effort mode messages can be lost if you send big data too fast and the buffer is filled at a faster rate than what the client can process messages. On the other hand, in reliable mode, the existence of a lot of data fragments could decrease the frequency at which messages are received. If this happens, it can be resolved by increasing socket buffers size, as described in *Increasing socket buffers size*. It can also help to set a lower Heartbeat period in reliable mode, as stated in *Tuning Reliable mode*.

When you are sending large data, it is convenient to setup a flow controller to avoid a burst of messages in the network and increase performance. See *Flow Controllers*

Example: Sending a unique large file

This is a proposed example of how should the user configure its application in order to achieve the best performance. To make this example more tangible, it is going to be supposed that the file has a size of 9.9MB and the network in which the publisher and the subscriber are operating has a bandwidth of 100MB/s

First of all, the asynchronous mode has to be activated in the publisher parameters. Then, a suitable reliability mode has to be selected. In this case, it is important to make sure that all fragments of the message are received. The loss of a fragment means the loss of the entire message, so it would be best to choose the reliable mode.

The default message size of this fragments using the UDPv4 transport has a value of 65Kb (which includes the space reserved for the data and the message header). This means that the publisher would have to write at least about 1100 fragments.

This amount of fragment could slow down the transmission, so it could be interesting to decrease the heartbeat period in order to increase the reactivity of the publisher.

Another important consideration is the addition of a flow controller. Without a flow controller, the publisher can occupy the entire bandwidth. A reasonable flow controller for this application could be a limit of 5MB/s, which represents only 5% of the total bandwidth. Anyway, these values are highly dependent on the specific application and its desired behavior.

At last, there is another detail to have in mind: it is critical to check the size of the system UDP buffers. In Linux, buffers can be enlarged with

```
sysctl -w net.ipv4.udp_mem="102400 873800 16777216"
sysctl -w net.core.netdev_max_backlog="30000"
sysctl -w net.core.rmem_max="16777216"
sysctl -w net.core.wmem_max="16777216"
```

Example: Video streaming

In this example, the target application transmits video between a publisher and a subscriber. This video will have a resolution of 640x480 and a frequency of 50fps.

As in the previous example, since the application is sending data that requires fragmentation, the asynchronous mode has to be activated in the publisher parameters.

In audio or video transmissions, sometimes is better to have a stable and high datarate feed than a 100% lossless communication. Working with a frequency of 50Hz makes insignificant the loss of one or two samples each second. Thus, for a higher performance, it can be appropriate to configure the reliability mode to best-effort.

6.16.4 Subscribing to Discovery Topics

As specified in the *Discovery* section, the Participant or RTPS Participant has a series of meta-data endpoints for use during the discovery process. The participant listener interface includes methods which are called each time a Publisher or a Subscriber is discovered. This allows you to create your own network analysis tools.

Implementation of custom listener

```
class CustomParticipantListener : public eprosima::fastrtps::ParticipantListener
{
    /* Custom Listener onSubscriberDiscovery */
    void onSubscriberDiscovery(
            eprosima::fastrtps::Participant* participant,
            eprosima::fastrtps::rtps::ReaderDiscoveryInfo&& info) override
    {
        (void)participant;
        switch (info.status){
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::DISCOVERED_READER:
                /* Process the case when a new subscriber was found in the domain */
                std::cout << "New subscriber for topic '" << info.info.topicName() <
↪< "' of type '" <<
                    info.info.typeName() << "' discovered";
                break;
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::CHANGED_QOS_READER:
                /* Process the case when a subscriber changed its QOS */
                break;
            case eprosima::fastrtps::rtps::ReaderDiscoveryInfo::REMOVED_READER:
                /* Process the case when a subscriber was removed from the domain */
                std::cout << "Subscriber for topic '" << info.info.topicName() << "
↪' of type '" << info.info.typeName() <<
                    "' left the domain.";
                break;
        }
    }

    /* Custom Listener onPublisherDiscovery */
    void onPublisherDiscovery(
            eprosima::fastrtps::Participant* participant,
            eprosima::fastrtps::rtps::WriterDiscoveryInfo&& info) override
    {
        (void)participant;
        switch (info.status){
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo ::DISCOVERED_WRITER:
                /* Process the case when a new publisher was found in the domain */
                std::cout << "New publisher for topic '" << info.info.topicName() <
↪< "' of type '" << info.info.typeName() <<
                    "' discovered";
                break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo ::CHANGED_QOS_WRITER:
                /* Process the case when a publisher changed its QOS */
                break;
            case eprosima::fastrtps::rtps::WriterDiscoveryInfo ::REMOVED_WRITER:
                /* Process the case when a publisher was removed from the domain */
                std::cout << "publisher for topic '" << info.info.topicName() << "'␣
↪of type '" << info.info.typeName() <<
                    "' left the domain.";
                break;
        }
    }

};
```

Setting the custom listener

```
    // Create Custom user ParticipantListener (should inherit from␣
↪eprosima::fastrtps::ParticipantListener.
    CustomParticipantListener* listener = new CustomParticipantListener();
    // Pass the listener on participant creation.
    Participant* participant = Domain::createParticipant(participant_attr,␣
↪listener);
```

The callbacks defined in the ReaderListener you attach to the EDP will execute for each data message after the built-in protocols have processed it.

6.16.5 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 a 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 change, thus reducing network overload.

The Partition membership of an Endpoint can be configured on the `qos.m_partitions` attribute of the `PublisherAttributes` or `SubscriberAttributes` objects. This attribute 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 Partition.

Note: 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++

```
PublisherAttributes pub_11_attr;
pub_11_attr.qos.m_partition.push_back("Partition_1");
pub_11_attr.qos.m_partition.push_back("Partition_2");

PublisherAttributes pub_12_attr;
pub_12_attr.qos.m_partition.push_back("*");

PublisherAttributes pub_21_attr;
//No partitions defined for pub_21

PublisherAttributes pub_22_attr;
pub_22_attr.qos.m_partition.push_back("Partition*");

SubscriberAttributes subs_31_attr;
subs_31_attr.qos.m_partition.push_back("Partition_1");

SubscriberAttributes subs_32_attr;
subs_32_attr.qos.m_partition.push_back("Partition_2");

SubscriberAttributes subs_33_attr;
subs_33_attr.qos.m_partition.push_back("Partition_3");

SubscriberAttributes subs_34_attr;
//No partitions defined for subs_34
```

XML

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

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

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

6.16.6 Intra-process delivery

eProsima Fast RTPS allows to speed up communications between entities within the same process by avoiding any of the copy or send operations involved in the transport layer (either UDP or TCP). This feature is enabled by default, and can be configured using *XML profiles*. Currently the following options are available:

- **INTRAPROCESS_OFF**: The feature is disabled.
- **INTRAPROCESS_USER_DATA_ONLY**: Discovery metadata keeps using ordinary transport.
- **INTRAPROCESS_FULL**: Default value. Both user data and discovery metadata using Intra-process delivery.

XML

```
<library_settings>
    <intraprocess_delivery>FULL</intraprocess_delivery> <!-- OFF | USER_DATA_
↪ONLY | FULL -->
</library_settings>
```

6.16.7 Tuning

Taking advantage of multicast

For topics with several subscribers, it is recommendable to configure them 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. Multicast configuration is explained in *RTPSEndpointQos*.

Increasing socket buffers size

In high rate scenarios or large data scenarios, the bottleneck could be the size of the socket buffers. Network packages could be dropped because there is no space in the socket buffer. Using Reliable *ReliabilityQosPolicy* *Fast RTPS* will try to recover lost samples, but with the penalty of retransmission. Using Best-Effort *ReliabilityQosPolicy* samples will be definitely lost.

By default *eProsima Fast RTPS* creates socket buffers with the system default size, but you can modify it. `sendSocketBufferSize` attribute helps to increase the socket buffer used to send data. `listenSocketBufferSize` attribute helps to increase the socket buffer used to read data.

C++

```
participant_attr.rtps.sendSocketBufferSize = 1048576;
participant_attr.rtps.listenSocketBufferSize = 4194304;
```

XML

```
<participant profile_name="participant_xml_profile_qos_socketbuffers">
    <rtps>
        <sendSocketBufferSize>1048576</sendSocketBufferSize>
        <listenSocketBufferSize>4194304</listenSocketBufferSize>
    </rtps>
</participant>
```

Finding out system maximum values

Linux operating system sets a maximum value for socket buffer sizes. When you set in *Fast RTPS* a socket buffer size, your value cannot exceed the maximum value of the system.

To get these values you can use the command `sysctl`. Maximum buffer size value of socket buffers used to send data could be retrieved using this 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
```

If these default maximum values are not enough for you, you can also increase them.

```
$> echo 'net.core.wmem_max=12582912' >> /etc/sysctl.conf
$> echo 'net.core.rmem_max=12582912' >> /etc/sysctl.conf
```

Tuning Reliable mode

RTPS protocol can maintain reliable communication using special messages (Heartbeat and Ack/Nack messages). RTPS protocol can detect which samples are lost and re-sent them again.

You can modify the frequency these special submessages are exchanged by specifying a custom heartbeat period. The heartbeat period in the Publisher-Subscriber level is configured as part of the `ParticipantAttributes`:

```
    publisher_attr.times.heartbeatPeriod.seconds = 0;
    publisher_attr.times.heartbeatPeriod.nanosec = 500000000; //500 ms
```

In the Writer-Reader layer, this belongs to the `WriterAttributes`:

```
    writer_attr.times.heartbeatPeriod.seconds = 0;
    writer_attr.times.heartbeatPeriod.nanosec = 500000000; //500 ms
```

A smaller heartbeat period increases the number of overhead messages in the network, but speeds up the system response when a piece of data is lost.

Non-strict reliability

Using a strict reliability, configuring *HistoryQosPolicyKind* kind as `KEEP_ALL`, determines all samples have to be received by all subscribers. This implicates a performance decrease in case a lot of samples are dropped. If you don't need this strictness, use a non-strict reliability, i.e. configure *HistoryQosPolicyKind* kind as `KEEP_LAST`.

Slow down sample rate

Sometimes publishers could send data in a too high rate for subscribers. This can end dropping samples. To avoid this you can slow down the rate using *Flow Controllers*.

6.16.8 Additional Quality of Service options

As a user, you can implement your own quality of service (QoS) restrictions in your application. *eProsima Fast RTPS* comes bundled with a set of examples of how to implement common client-wise QoS settings:

- Ownership Strength: When multiple data sources come online, filter duplicates by focusing on the higher priority sources.
- Filtering: Filter incoming messages based on content, time, or both.

These examples come with their own *Readme.txt* that explains how the implementations work.

6.17 Real-time behavior

Fast RTPS can be configured to offer real-time features. These features will guarantee Fast RTPS responses within specified time constrains. To maintain this compromise Fast RTPS is able to have the following behavior:

- Not allocate memory after the initialization of Fast RTPS entities.
- Several methods are blocked for a maximum period of time.

This section explains how to configure Fast RTPS to achieve this behavior. For easier understanding it was divided in two subsections:

- *Tuning allocations*: configuration to avoid memory allocation after initialization.
- *Non-blocking calls*: usage of non-blocking methods for real-time behavior.

6.17.1 Tuning allocations

Some important non-deterministic operating system calls are the ones for allocating and deallocating memory. Most real-time systems have the need to operate in a way that all dynamic memory is allocated on the application startup, and avoid calls to memory management APIs on the main loop.

Fast-RTPS provides some configuration parameters to meet these requirements, allowing the items of internal data collections to be preallocated. In order to choose the correct values for these parameters, the user should be aware of the topology of the whole domain, so the number of participants and endpoints should be known when setting them.

Parameters on the participant

All the allocation related parameters on the participant are grouped into the `rtps.allocation` field of the `ParticipantAttributes` struct.

Limiting the number of discovered participants

Every participant in Fast-RTPS holds an internal collection of `ParticipantProxyData` objects with the information of the local and the remote participants. Field `participants` inside `RTPSParticipantAllocationAttributes` allows the configuration of the allocation behavior of that collection. The user can specify the `initial` number of elements preallocated, the `maximum` number of elements allowed, and the allocation `increment`. By default, a full dynamic behavior is used.

Limiting the number of discovered endpoints

Every `ParticipantProxyData` object holds internal collections with the `ReaderProxyData` and `WriterProxyData` objects with the information of the readers and writers of a participant. In a similar way to the `participants` field, `RTPSParticipantAllocationAttributes` has fields `readers` and `writers` to set the configuration of the allocation behavior of those collections. The user can specify the `initial` number of elements preallocated, the `maximum` number of elements allowed, and the allocation `increment`. By default, a full dynamic behavior is used.

Limiting the size of parameters

Most of the information held for participants and endpoints have a defined size limit, so the amount of memory to allocate for each local and remote peer is known. For the parameters which size is not limited, a maximum size can be configured with `RTPSParticipantAllocationAttributes::data_limits`, which has the following attributes:

- `max_partitions` limits the size of partition data to the given number of octets.
- `max_user_data` limits the size of user data to the given number of octets.
- `max_properties` limits the size of participant properties data to the given number of octets.

A value of zero implies no size limitation. If these sizes are configured to something different than zero, enough memory will be allocated for them for each participant and endpoint. If these sizes are not limited, memory will be dynamically allocated as needed. By default, a full dynamic behavior is used.

Parameters on the publisher

Every publisher holds a collection with some information regarding the subscribers it has matched to. Field `matched_subscriber_allocation` inside `PublisherAttributes` allows the configuration of the allocation behavior of that collection. The user can specify the `initial` number of elements preallocated, the `maximum` number of elements allowed, and the allocation `increment`. By default, a full dynamic behavior is used.

Parameters on the subscriber

Every subscriber holds a collection with some information regarding the publishers it has matched to. Field `matched_publisher_allocation` inside `SubscriberAttributes` allows the configuration of the allocation behavior of that collection. The user can specify the `initial` number of elements preallocated, the `maximum` number of elements allowed, and the allocation `increment`. By default, a full dynamic behavior is used.

Full example

Given a system with the following topology:

Table 2: **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

- All the subscribers match exactly with 1 publisher.
- The publisher for topic 1 matches with 3 subscribers, and the publisher for topic 2 matches with 2 subscribers.
- The maximum number of publishers per participant is 1, and the maximum number of subscribers per participant is 2.
- The total number of participants is 3.

We will also limit the size of the parameters:

- Maximum partition data size: 256
- Maximum user data size: 256
- Maximum properties data size: 512

The following piece of code shows the set of parameters needed for the use case depicted in this example.

C++

```
    // Before creating a participant:
    // We know we have 3 participants on the domain
    participant_attr.rtps.allocation.participants =
            eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);
    // We know we have at most 2 readers on each participant
    participant_attr.rtps.allocation.readers =
            eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(2u);
    // We know we have at most 1 writer on each participant
    participant_attr.rtps.allocation.writers =
            eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(1u);
    // We know the maximum size of partition data
    participant_attr.rtps.allocation.data_limits.max_partitions = 256u;
    // We know the maximum size of user data
    participant_attr.rtps.allocation.data_limits.max_user_data = 256u;
    // We know the maximum size of properties data
    participant_attr.rtps.allocation.data_limits.max_properties = 512u;

    // Before creating the publisher for topic 1:
    // we know we will only have three matching subscribers
    publisher_attr.matched_subscriber_allocation =
            eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(3u);

    // Before creating the publisher for topic 2:
    // we know we will only have two matching subscribers
    publisher_attr.matched_subscriber_allocation =
            eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(2u);

    // Before creating a subscriber:
    // we know we will only have one matching publisher
    subscriber_attr.matched_publisher_allocation =
            eprosima::fastrtps::ResourceLimitedContainerConfig::fixed_size_
↪configuration(1u);
```

XML

```
<participant profile_name="participant_alloc_qos_example">
    <rtps>
        <allocation>
            <!-- We know we have 3 participants on the domain -->
            <total_participants>
                <initial>3</initial>
                <maximum>3</maximum>
                <increment>0</increment>
            </total_participants>
            <!-- We know we have at most 2 readers on each participant -->
            <total_readers>
                <initial>2</initial>
                <maximum>2</maximum>
                <increment>0</increment>
            </total_readers>
            <!-- We know we have at most 1 writer on each participant -->
            <total_writers>
                <initial>1</initial>
                <maximum>1</maximum>
                <increment>0</increment>
            </total_writers>
            <max_partitions>256</max_partitions>
```

6.17.2 Non-blocking calls

Note: This feature is not fully supported on OSX. It doesn't support necessary POSIX Real-time features. The feature is limited by the implementation of *std::timed_mutex* and *std::condition_variable_any*.

It is important that a method isn't blocked for indeterminate time to achieve real-time. A method must only be blocked for a maximum period of time. In Fast-RTPS API there are several methods that permit to set this. But first Fast-RTPS should be configured with the CMake option `-DSTRICT_REALTIME=ON`. The list of these functions is displayed in the table below.

Table 3: **Fast RTPS non-blocking API**

Method	Description
Publisher::write()	These methods are blocked for a period of time. *ReliabilityQosPolicy.max_blocking_time* on *PublisherAttributes* defines this period of time. Default value is 100 milliseconds.
Subscriber::takeNextData()	This methods is blocked for a period of time. *ReliabilityQosPolicy.max_blocking_time* on *SubscriberAttributes* defines this period of time. Default value is 100 milliseconds.
Subscriber::readNextData()	This method is blocked for a period of time. *ReliabilityQosPolicy.max_blocking_time* on *SubscriberAttributes* defines this period of time. Default value is 100 milliseconds.
Subscriber::wait_for_unread_samples()	Accepts an argument specifying how long the method can be blocked.

6.18 Persistence

By default, the writer's history is available for remote readers throughout writer's life. You can configure Fast RTPS to provide persistence between application executions. When a writer is created again, it will maintain the previous history and a new remote reader will receive all samples sent by the writer throughout its life.

A reader keeps information on the latest change notified to the user for each matching writer. Persisting this information, you could save bandwidth, as the reader will not ask the writers for changes already notified.

In summary, enabling this feature you will protect the state of endpoints against unexpected failures, as they will continue communicating after being restarted as if they were just disconnected from the network.

Imagine, for instance, that a writer with a policy to keep its last 100 samples has its history full of changes and the machine where it runs has a power failure. When the writer is started again, if a new reader is created, it will not receive the 100 samples that were on the history of the writer. With persistence enabled, changes in the history of the writer will be written to disk and read again when the writer is restarted.

With readers, the information written to disk is different. Only information about the last change notified to the user is stored on disk. When a persistent reader is restarted, it will load this information, and will only ask the matching writers to resend those changes that were not notified to the upper layers.

persistence_guid

Whenever an endpoint (reader or writer) is created, a unique identifier (GUID) is generated. If the endpoint is restarted, a new GUID will be generated, and other endpoints won't be able to know it was the same one. For this reason, a specific parameter *persistence_guid* should be configured on `eprosima::fastrtps::rtps::EndpointAttributes`. This parameter will be used as the primary key of the data saved on disk, and will also be used to identify the endpoint on the DDS domain.

6.18.1 Configuration

We recommend you to look at the example of how to use this feature the distribution comes with while reading this section. It is located in *examples/RTPSTest_persistent*

In order for the persistence feature to work, some specific `eprosima::fastrtps::rtps::Writer` or `eprosima::fastrtps::rtps::Reader` attributes should be set:

- `durabilityKind` should be set to `TRANSIENT`
- `persistence_guid` should not be all zeros
- A persistence plugin should be configured either on the `eprosima::fastrtps::rtps::Writer`, the `eprosima::fastrtps::rtps::Reader` or the `eprosima::fastrtps::rtps::RTPSParticipant`

You can select and configure the persistence plugin through `eprosima::fastrtps::rtps::RTPSParticipant` attributes using properties. A `eprosima::fastrtps::rtps::Property` is defined by its name (`std::string`) and its value (`std::string`). Throughout this page, there are tables showing you the properties used by each persistence plugin.

6.18.2 Built-in plugins

The current version comes out with one persistence built-in plugin:

- **SQLITE3**: this plugin provides persistence on a local file using SQLite3 API.

PERSISTENCE:SQLITE3

This built-in plugin provides persistence on a local file using SQLite3 API.

You can activate this plugin using RTPSParticipant, Reader or Writer property `dds.persistence.plugin` with the value `builtin.SQLITE3`. Next table shows you the properties used by this persistence plugin.

Table 4: **Properties to configure Persistence::SQLITE3**

Property name (all properties have "dds.persistence.sqlite3." prefix)	Property value
filename	Name of the file used for persistent storage. Default value: *persistence.db*

Note: Currently this plugin doesn't support two processes accessing the same SQLite3 file. It could end up in inconsistency and a failure. Be sure each process uses a different SQLite3 file.

Example

This example shows you how to configure an RTPSParticipant to activate and configure *PERSISTENCE:SQLITE3* plugin. It also configures a Writer to persist its history on local storage, and a Reader to persist the highest notified sequence number on local storage.

RTPSParticipant attributes

```
    eprosima::fastrtps::rtps::RTPSParticipantAttributes part_attr;

    // Activate Persistence:SQLITE3 plugin
    part_attr.properties.properties().emplace_back("dds.persistence.plugin", "builtin.
↪SQLITE3");
```

(continues on next page)

(continued from previous page)

```
    // Configure Persistence:SQLITE3 plugin
    part_attr.properties.properties().emplace_back("dds.persistence.sqlite3.filename",
↪ "example.db");
```

Writer attributes

```
    eprosima::fastrtps::rtps::WriterAttributes writer_attr;

    // Set durability to TRANSIENT
    writer_attr.endpoint.durabilityKind = TRANSIENT;

    // Set persistence_guid
    writer_attr.endpoint.persistence_guid.guidPrefix.value[11] = 1;
    writer_attr.endpoint.persistence_guid.entityId = 0x12345678;
```

Reader attributes

```
    eprosima::fastrtps::rtps::ReaderAttributes reader_attr;

    // Set durability to TRANSIENT
    reader_attr.endpoint.durabilityKind = TRANSIENT;

    // Set persistence_guid
    reader_attr.endpoint.persistence_guid.guidPrefix.value[11] = 1;
    reader_attr.endpoint.persistence_guid.entityId = 0x3456789A;
```

6.19 Code generation using fastrtpsgen

eprosima Fast RTPS comes with a built-in code generation tool, fastrtpsgen, which eases the process of translating an IDL specification of a data type to a working implementation of the methods needed to create topics, used by publishers and subscribers, of that data type. This tool can be instructed to generate a sample application using this data type, providing a *Makefile* to compile it on Linux and a Visual Studio project for Windows.

fastrtpsgen can be invoked by calling fastrtpsgen on Linux or fastrtpsgen.bat on Windows.

```
fastrtpsgen [-d <outputdir>] [-example <platform>] [-replace] [-typeobject] <IDLfile>␣
↪[<IDLfile> ...]
```

The *-replace* argument is needed to replace the currently existing files in case the files for the IDL have been generated previously.

When the *-example* argument is added, the tool will generate an automated example and the files to build it for the platform currently invoked. The *-help* argument provides a list of currently supported Visual Studio versions and platforms.

When *-typeobject* argument is used, the tool will generate additional files for TypeObject generation and management. For more information about TypeObject go to *Dynamic Topic Types*.

6.19.1 Output

fastrtpsgen outputs the several files. Assuming the IDL file had the name *"Mytype"*, these files are:

- MyType.cxx/.h: Type definition.
- MyTypePublisher.cxx/.h: Definition of the Publisher as well as of a PublisherListener. The user must fill the needed methods for his application.
- MyTypeSubscriber.cxx/.h: Definition of the Subscriber as well as of a SubscriberListener. The behavior of the subscriber can be altered changing the methods implemented on these files.
- MyTypePubSubType.cxx/.h: Serialization and Deserialization code for the type. It also defines the *getKey* method in case the topic uses keys.
- MyTypePubSubMain.cxx: The main file of the example application in case it is generated.
- *Makefile* or Visual Studio project files.

If *-typeobject* was used, *MyType.cxx* is modified to register the TypeObject representation in the TypeObjectFactory, and these files will be generated too:

- MyTypeTypeObject.cxx/.h: TypeObject representation for *MyType* IDL.

6.19.2 Where to find *fastrtpsgen*

If you are using the binary distribution of *eProsima Fast RTPS*, *fastrtpsgen* is already provided for you. If you are building from sources, you have to compile *fastrtpsgen*. You can find instructions in section *Installation from Sources*.

6.20 API Reference

Fast RTPS, 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.

6.20.1 DDS DCPS PIM

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

Core

Entity

class `eprosima::fastdds::dds::`**`Entity`**

The *Entity* class is the abstract base class for all the objects that support QoS policies, a listener and a status condition.

Subclassed by *eprosima::fastdds::dds::DomainEntity*, *eprosima::fastdds::dds::DomainParticipant*

Public Functions

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

Parameters

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

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

Return RETCODE_OK

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

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

Return Reference to the *StatusMask* with the relevant statuses set to 1

const *StatusMask* &**get_status_changes**() **const**
Retrieves the set of triggered statuses in the *Entity*.

Triggered statuses are the ones whose value has changed since the last time the application read the status. When the entity is first created or if the entity is not enabled, all communication statuses are in the non-triggered state, so the list returned by the get_status_changes operation will be empty. The list of statuses returned by the get_status_changes operation refers to the status that are triggered on the *Entity* itself and does not include statuses that apply to contained entities.

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

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

Return Reference to the InstanceHandle

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

Return true if enabled, false if not

DomainEntity

class `eprosima::fastdds::dds::`**DomainEntity** : **public** eprosima::fastdds::dds::*Entity*
The *DomainEntity* class is a subclass of *Entity* created in order to differentiate between DomainParticipants and the rest of Entities.

Subclassed by *eprosima::fastdds::dds::DataReader*, *eprosima::fastdds::dds::DataWriter*, *eprosima::fastdds::dds::Publisher*, *eprosima::fastdds::dds::Subscriber*, *eprosima::fastdds::dds::Topic*

Public Functions

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

Parameters

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

Policy

DataRepresentationId

enum `dds::fastdds::eprosima::`**DataRepresentationId**
Enum DataRepresentationId, different kinds of topic data representation

Values:

enumerator XCDR_DATA_REPRESENTATION = 0
Extended CDR Encoding version 1.

enumerator XML_DATA_REPRESENTATION = 1
XML Data Representation (Unsupported)

enumerator XCDR2_DATA_REPRESENTATION = 2
Extended CDR Encoding version 2.

DataRepresentationQosPolicy

class `eprosima::fastdds::dds::`**DataRepresentationQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **p**
With multiple standard data Representations available, and vendor-specific extensions possible, DataWriters and DataReaders must be able to negotiate which data representation(s) to use. This negotiation shall occur based on *DataRepresentationQosPolicy*.

Warning If a writer's offered representation is contained within a reader's sequence, the offer satisfies the request and the policies are compatible. Otherwise, they are incompatible.

Note Immutable Qos Policy

Public Functions

DataRepresentationQosPolicy()
Constructor.

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

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

Return True if the policy is equal.

Parameters

- `b`: QoS Policy.

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

Public Members

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

DeadlineQosPolicy

class eprosima::fastdds::dds::**DeadlineQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosi
DataReader expects a new sample updating the value of each instance at least once every deadline period. *DataWriter* indicates that the application commits to write a new value (using the *DataWriter*) for each instance managed by the *DataWriter* at least once every deadline period.

Note Mutable Qos Policy

Public Functions

DeadlineQosPolicy()
Constructor.

~DeadlineQosPolicy() = default
Destructor.

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

Public Members

fastrtps::Duration_t **period**
Maximum time expected between samples. It is inconsistent for a *DataReader* to have a DEADLINE period less than its *TimeBasedFilterQosPolicy* minimum_separation. By default, c_TimeInifinite.

DestinationOrderQosPolicy

class eprosima::fastdds::dds::**DestinationOrderQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **pub**
Controls the criteria used to determine the logical order among changes made by *Publisher* entities to the same instance of data (i.e., matching *Topic* and key).

Warning This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Note Immutable Qos Policy

Public Functions

DestinationOrderQosPolicy()
Constructor.

~DestinationOrderQosPolicy() = default
Destructor.

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

Public Members

DestinationOrderQosPolicyKind **kind**
DestinationOrderQosPolicyKind. By default, BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS.

DestinationOrderQosPolicyKind

enum dds::fastdds::eprosima::**DestinationOrderQosPolicyKind**
Enum DestinationOrderQosPolicyKind, different kinds of destination order for *DestinationOrderQosPolicy*.

Values:

enumerator BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
Indicates that data is ordered based on the reception time at each *Subscriber*. Since each subscriber may receive the data at different times there is no guaranteed that the changes will be seen in the same order. Consequently, it is possible for each subscriber to end up with a different final value for the data.

enumerator BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
Indicates that data is ordered based on a timestamp placed at the source (by the Service or by the application). In any case this guarantees a consistent final value for the data in all subscribers.

DisablePositiveACKsQosPolicy

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

Note Immutable Qos Policy

Public Functions

DisablePositiveACKsQosPolicy()
Constructor.

~DisablePositiveACKsQosPolicy() = default
Destructor.

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

Public Members

bool **enabled**
True if this QoS is enabled. By default, false.

fastrtps::Duration_t **duration**
The duration to keep samples for (not serialized as not needed by reader). By default, c_TimeInfinite.

DurabilityQosPolicy

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

Note Immutable Qos Policy

Public Functions

DurabilityQosPolicy()
Constructor.

~DurabilityQosPolicy() = default
Destructor.

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

Return fastrtps::rtps::DurabilityKind_t

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

Parameters

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

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

Public Members

DurabilityQosPolicyKind_t **kind**
DurabilityQosPolicyKind. By default the value for DataReaders: VOLATILE_DURABILITY_QOS, for DataWriters TRANSIENT_LOCAL_DURABILITY_QOS.

DurabilityQosPolicyKind

enum `dds::fastdds::eprosima::`**DurabilityQosPolicyKind**
Enum DurabilityQosPolicyKind_t, different kinds of durability for *DurabilityQosPolicy*.

Values:

enumerator VOLATILE_DURABILITY_QOS
The Service does not need to keep any samples of data-instances on behalf of any *DataReader* that is not known by the *DataWriter* at the time the instance is written. In other words the Service will only attempt to provide the data to existing subscribers

enumerator TRANSIENT_LOCAL_DURABILITY_QOS
For TRANSIENT_LOCAL, the service is only required to keep the data in the memory of the *DataWriter* that wrote the data and the data is not required to survive the *DataWriter*.

enumerator TRANSIENT_DURABILITY_QOS
For TRANSIENT, the service is only required to keep the data in memory and not in permanent storage; but the data is not tied to the lifecycle of the *DataWriter* and will, in general, survive it.

enumerator PERSISTENT_DURABILITY_QOS
Data is kept on permanent storage, so that they can outlive a system session.

Warning Not Supported

DurabilityServiceQosPolicy

class eprosima::fastdds::dds::**DurabilityServiceQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **pu**
Specifies the configuration of the durability service. That is, the service that implements the *DurabilityQosPolicy* kind of TRANSIENT and PERSISTENT.

Warning This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Note Immutable Qos Policy

Public Functions

DurabilityServiceQosPolicy()
Constructor.

~DurabilityServiceQosPolicy() = default
Destructor.

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

Public Members

fastrtps::Duration_t **service_cleanup_delay**
Control when the service is able to remove all information regarding a data-instance. By default, c_TimeZero.

HistoryQosPolicyKind **history_kind**
Controls the *HistoryQosPolicy* of the fictitious *DataReader* that stores the data within the durability service. By default, KEEP_LAST_HISTORY_QOS.

int32_t **history_depth**
Number of most recent values that should be maintained on the History. It only have effect if the history_kind is KEEP_LAST_HISTORY_QOS. By default, 1.

int32_t **max_samples**
Control the ResourceLimitsQos of the implied *DataReader* that stores the data within the durability service. Specifies the maximum number of data-samples the *DataWriter* (or *DataReader*) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one *DataWriter* (or *DataReader*). It is inconsistent for this value to be less than max_samples_per_instance. By default, -1 (Length Unlimited).

int32_t **max_instances**
Control the ResourceLimitsQos of the implied *DataReader* that stores the data within the durability service. Represents the maximum number of instances *DataWriter* (or *DataReader*) can manage. By default, -1 (Length Unlimited).

int32_t **max_samples_per_instance**
Control the ResourceLimitsQos of the implied *DataReader* that stores the data within the durability service. Represents the maximum number of samples of any one instance a DataWriter(or DataReader) can manage. It is inconsistent for this value to be greater than max_samples. By default, -1 (Length Unlimited).

EntityFactoryQosPolicy

class `eprosima::fastdds::dds::`**EntityFactoryQosPolicy**
Controls the behavior of the entity when acting as a factory for other entities. In other words, configures the side-effects of the create_* and delete_* operations.

Note Mutable Qos Policy

Public Functions

EntityFactoryQosPolicy()
Constructor without parameters.

EntityFactoryQosPolicy(bool *autoenable*)
Constructor.

Parameters

- `autoenable`: Value for the autoenable_created_entities boolean

~EntityFactoryQosPolicy()
Destructor.

Public Members

bool **autoenable_created_entities**
Specifies whether the entity acting as a factory automatically enables the instances it creates. If True the factory will automatically enable each created *Entity* otherwise it will not. By default, True.

GenericDataQosPolicy

class `eprosima::fastdds::dds::`**GenericDataQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** ep
Class *GenericDataQosPolicy*, base class to transmit user data during the discovery phase.

Subclassed by *eprosima::fastdds::dds::GroupDataQosPolicy*, *eprosima::fastdds::dds::TopicDataQosPolicy*, *eprosima::fastdds::dds::UserDataQosPolicy*

Public Functions

GenericDataQosPolicy(**const** *GenericDataQosPolicy* &*data*)
Construct from another *GenericDataQosPolicy*.

The resulting *GenericDataQosPolicy* will have the same size limits as the input attribute

Parameters

- `data`: data to copy in the newly created object

GenericDataQosPolicy(ParameterId_t *pid*, **const** collection_type &*data*)
Construct from underlying collection type.

Useful to easy integration on old APIs where a traditional container was used. The resulting *GenericDataQosPolicy* will always be unlimited in size

Parameters

- `pid`: Id of the parameter
- `data`: data to copy in the newly created object

GenericDataQosPolicy &**operator=**(**const** collection_type &*b*)
Copies data from underlying collection type.

Useful to easy integration on old APIs where a traditional container was used. The resulting *GenericDataQosPolicy* will keep the current size limit. If the input data is larger than the current limit size, the elements exceeding that maximum will be silently discarded.

Return reference to the current object.

Parameters

- `b`: object to be copied

GenericDataQosPolicy &**operator=**(**const** *GenericDataQosPolicy* &*b*)
Copies another *GenericDataQosPolicy*.

The resulting *GenericDataQosPolicy* will have the same size limit as the input parameter, so all data in the input will be copied.

Return reference to the current object.

Parameters

- `b`: object to be copied

void **set_max_size**(size_t *size*)
Set the maximum size of the user data and reserves memory for that much.

Parameters

- `size`: new maximum size of the user data. Zero for unlimited size

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

Return const reference to the internal raw data.

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

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

Return raw data as vector of octets.

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

Return raw data as vector of octets.

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

Parameters

- `vec`: raw data to set.

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

Return raw data as vector of octets.

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

Parameters

- `vec`: raw data to set.

GroupDataQosPolicy

class GroupDataQosPolicy : **public** eprosima::fastdds::dds::*GenericDataQosPolicy*
Class derived from *GenericDataQosPolicy*.

The purpose of this QoS is to allow the application to attach additional information to the created *Publisher* or *Subscriber*. The value of the GROUP_DATA is available to the application on the *DataReader* and *DataWriter* entities and is propagated by means of the built-in topics.

This QoS can be used by an application combination with the *DataReaderListener* and *DataWriterListener* to implement matching policies similar to those of the PARTITION QoS except the decision can be made based on an application-defined policy.

HistoryQosPolicy

class `eprosima::fastdds::dds::`**HistoryQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosim
Specifies the behavior of the Service in the case where the value of a sample changes (one or more times) before it can be successfully communicated to one or more existing subscribers. This QoS policy controls whether the Service should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between. On the publishing side this policy controls the samples that should be maintained by the *DataWriter* on behalf of existing *DataReader* entities. The behavior with regards to a DataReaderentities discovered after a sample is written is controlled by the DURABILITY QoS policy. On the subscribing side it controls the samples that should be maintained until the application "takes" them from the Service.

Note Immutable Qos Policy

Public Functions

HistoryQosPolicy()
Constructor.

~HistoryQosPolicy() = default
Destructor.

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

Public Members

HistoryQosPolicyKind **kind**
HistoryQosPolicyKind. By default, KEEP_LAST_HISTORY_QOS.

int32_t **depth**
History depth. By default, 1. If a value other than 1 is specified, it should be consistent with the settings of the *ResourceLimitsQosPolicy*.

Warning Only takes effect if the kind is KEEP_LAST_HISTORY_QOS.

HistoryQosPolicyKind

enum dds::fastdds::eprosima::**HistoryQosPolicyKind**
Enum HistoryQosPolicyKind, different kinds of History Qos for *HistoryQosPolicy*.

Values:

enumerator KEEP_LAST_HISTORY_QOS
On the publishing side, the Service will only attempt to keep the most recent "depth" samples of each instance of data (identified by its key) managed by the *DataWriter*. On the subscribing side, the *DataReader* will only attempt to keep the most recent "depth" samples received for each instance (identified by its key) until the application "takes" them via the DataReader's take operation.

enumerator KEEP_ALL_HISTORY_QOS
On the publishing side, the Service will attempt to keep all samples (representing each value written) of each instance of data (identified by its key) managed by the *DataWriter* until they can be delivered to all subscribers. On the subscribing side, the Service will attempt to keep all samples of each instance of data (identified by its key) managed by the *DataReader*. These samples are kept until the application "takes" them from the Service via the take operation.

LatencyBudgetQosPolicy

class eprosima::fastdds::dds::**LatencyBudgetQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public**
Specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver's application-cache and the receiving application is notified of the fact.This policy is a hint to the Service, not something that must be monitored or enforced. The Service is not required to track or alert the user of any violation.

Warning This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Note Mutable Qos Policy

Public Functions

LatencyBudgetQosPolicy()
Constructor.

~LatencyBudgetQosPolicy() = default
Destructor.

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

Public Members

fastrtps::Duration_t **duration**
Maximum acceptable delay from the time data is written until it is received. By default, c_TimeZero.

LifespanQosPolicy

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

Note Mutable Qos Policy

Public Functions

LifespanQosPolicy()
Constructor.

~LifespanQosPolicy() = default
Destructor.

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

Public Members

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

LivelinessQosPolicy

class eprosima::fastdds::dds::**LivelinessQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** epr
Determines the mechanism and parameters used by the application to determine whether an *Entity* is "active" (alive). The "liveliness" status of an *Entity* is used to maintain instance ownership in combination with the setting of the *OwnershipQosPolicy*. The application is also informed via listener when an *Entity* is no longer alive.

The *DataReader* requests that liveliness of the writers is maintained by the requested means and loss of liveliness is detected with delay not to exceed the lease_duration.

The *DataWriter* commits to signaling its liveliness using the stated means at intervals not to exceed the lease_duration. Listeners are used to notify the DataReaderof loss of liveliness and *DataWriter* of violations to the liveliness contract.

Public Functions

LivelinessQosPolicy()
Constructor.

~LivelinessQosPolicy() = default
Destructor.

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

Public Members

LivelinessQosPolicyKind **kind**
Liveliness kind By default, AUTOMATIC_LIVELINESS.

fastrtps::Duration_t **lease_duration**
Period within which liveliness should be asserted. On a *DataWriter* it represents the period it commits to signal its liveliness. On a *DataReader* it represents the period without assertion after which a *DataWriter* is considered inactive. By default, c_TimeInfinite.

fastrtps::Duration_t **announcement_period**
The period for automatic assertion of liveliness. Only used for DataWriters with AUTOMATIC liveliness. By default, c_TimeInfinite.

Warning When not infinite, must be < lease_duration, and it is advisable to be less than 0.7*lease_duration.

LivelinessQosPolicyKind

enum dds::fastdds::eprosima::**LivelinessQosPolicyKind**
Enum LivelinessQosPolicyKind, different kinds of liveliness for *LivelinessQosPolicy*

Values:

enumerator AUTOMATIC_LIVELINESS_QOS
The infrastructure will automatically signal liveliness for the DataWriters at least as often as required by the lease_duration.

enumerator MANUAL_BY_PARTICIPANT_LIVELINESS_QOS
The Service will assume that as long as at least one *Entity* within the *DomainParticipant* has asserted its liveliness the other Entities in that same *DomainParticipant* are also alive.

enumerator MANUAL_BY_TOPIC_LIVELINESS_QOS
The Service will only assume liveliness of the *DataWriter* if the application has asserted liveliness of that *DataWriter* itself.

OwnershipQosPolicy

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

Specifies whether it is allowed for multiple DataWriters to write the same instance of the data and if so, how these modifications should be arbitrated

Note Immutable Qos Policy

Public Functions

OwnershipQosPolicy()

Constructor.

~OwnershipQosPolicy() = default

Destructor.

void **clear**() **override**

Clears the *QosPolicy* object.

Public Members

OwnershipQosPolicyKind **kind**

OwnershipQosPolicyKind.

OwnershipQosPolicyKind

enum dds::fastdds::eprosima::**OwnershipQosPolicyKind**

Enum OwnershipQosPolicyKind, different kinds of ownership for *OwnershipQosPolicy*.

Values:

enumerator SHARED_OWNERSHIP_QOS

Indicates shared ownership for each instance. Multiple writers are allowed to update the same instance and all the updates are made available to the readers. In other words there is no concept of an "owner" for the instances.

enumerator EXCLUSIVE_OWNERSHIP_QOS

Indicates each instance can only be owned by one *DataWriter*, but the owner of an instance can change dynamically. The selection of the owner is controlled by the setting of the *OwnershipStrengthQosPolicy*. The owner is always set to be the highest-strength *DataWriter* object among the ones currently "active" (as determined by the *LivelinessQosPolicy*).

OwnershipStrengthQosPolicy

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

Specifies the value of the "strength" used to arbitrate among multiple *DataWriter* objects that attempt to modify the same instance of a data-object (identified by *Topic* + key).This policy only applies if the OWNERSHIP QoS policy is of kind EXCLUSIVE.

Note Mutable Qos Policy

Public Functions

OwnershipStrengthQosPolicy()
Constructor.

~OwnershipStrengthQosPolicy() = default
Destructor.

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

Public Members

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

Partition_t

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

Public Functions

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

Parameters

- `ptr`: Pointer to be set

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

Return uint32_t with the size

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

Return name

PartitionQosPolicy

class eprosima::fastdds::dds::**PartitionQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** epros
Set of strings that introduces a logical partition among the topics visible by the *Publisher* and *Subscriber*. A *DataWriter* within a *Publisher* only communicates with a *DataReader* in a *Subscriber* if (in addition to matching the *Topic* and having compatible QoS) the *Publisher* and *Subscriber* have a common partition name string.

The empty string ("") is considered a valid partition that is matched with other partition names using the same rules of string matching and regular-expression matching used for any other partition name.

Note Mutable Qos Policy

Public Functions

PartitionQosPolicy()
Constructor without parameters.

PartitionQosPolicy(uint16_t *in_length*)
Constructor using Parameter length.

Parameters

- `in_length`: Length of the parameter

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

Parameters

- `b`: Another *PartitionQosPolicy* instance

~PartitionQosPolicy() = default
Destructor.

const_iterator **begin**() **const**
Getter for the first position of the partition list.

Return *const_iterator*

const_iterator **end**() **const**
Getter for the end of the partition list.

Return *const_iterator*

uint32_t **size**() **const**
Getter for the number of partitions.

Return uint32_t with the size

uint32_t **empty**() **const**
Check if the set is empty.

Return true if it is empty, false otherwise

void **set_max_size**(uint32_t *size*)
Setter for the maximum size.

Parameters

- `size`: Size to be set

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

Return uint32_t with the maximum size

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

Parameters

- `name`: Name to append.

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

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

Return Vector of partition name strings.

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

Parameters

- `nam`: Vector of partition name strings.

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

Return Vector of partition name strings.

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

Parameters

- `nam`: Vector of partition name strings.

class const_iterator

Public Functions

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

Parameters
- `ptr`: Pointer to be set

PresentationQosPolicy

class eprosima::fastdds::dds::**PresentationQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** e
Specifies how the samples representing changes to data instances are presented to the subscribing application. This policy affects the application's ability to specify and receive coherent changes and to see the relative order of changes.access_scope determines the largest scope spanning the entities for which the order and coherency of changes can be preserved. The two booleans control whether coherent access and ordered access are supported within the scope access_scope.

Warning This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Note Immutable Qos Policy

Public Functions

PresentationQosPolicy()
Constructor without parameters.

~PresentationQosPolicy() = default
Destructor.

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

Public Members

PresentationQosPolicyAccessScopeKind **access_scope**
Access Scope Kind By default, INSTANCE_PRESENTATION_QOS.

bool **coherent_access**
Specifies support coherent access. That is, the ability to group a set of changes as a unit on the publishing end such that they are received as a unit at the subscribing end. by default, false.

bool **ordered_access**
Specifies support for ordered access to the samples received at the subscription end. That is, the ability of the subscriber to see changes in the same order as they occurred on the publishing end. By default, false.

PresentationQosPolicyAccessScopeKind

enum dds::fastdds::eprosima::**PresentationQosPolicyAccessScopeKind**
Enum PresentationQosPolicyAccessScopeKind, different kinds of Presentation Policy order for *PresentationQosPolicy*.

Values:

enumerator INSTANCE_PRESENTATION_QOS
Scope spans only a single instance. Indicates that changes to one instance need not be coherent nor ordered with respect to changes to any other instance. In other words, order and coherent changes apply to each instance separately.

enumerator TOPIC_PRESENTATION_QOS
Scope spans to all instances within the same *DataWriter* (or *DataReader*), but not across instances in different *DataWriter* (or *DataReader*).

enumerator GROUP_PRESENTATION_QOS
Scope spans to all instances belonging to *DataWriter* (or *DataReader*) entities within the same *Publisher* (or *Subscriber*).

PropertyPolicyQos

using dds::fastdds::eprosima::**PropertyPolicyQos** = fastrtps::rtps::PropertyPolicy
Property policies.

PublishModeQosPolicy

class eprosima::fastdds::dds::**PublishModeQosPolicy** : **public** eprosima::fastdds::dds::*QosPolicy*

Class *PublishModeQosPolicy*, defines the publication mode for a specific writer.

Public Functions

PublishModeQosPolicy()

Constructor.

~PublishModeQosPolicy() = default

Destructor.

void **clear**() **override**

Clears the *QosPolicy* object.

Public Members

PublishModeQosPolicyKind **kind**

PublishModeQosPolicyKind By default, SYNCHRONOUS_PUBLISH_MODE.

PublishModeQosPolicyKind

enum dds::fastdds::eprosima::**PublishModeQosPolicyKind**

Enum PublishModeQosPolicyKind, different kinds of publication synchronism

Values:

enumerator SYNCHRONOUS_PUBLISH_MODE

Synchronous publication mode (default for writers).

enumerator ASYNCHRONOUS_PUBLISH_MODE

Asynchronous publication mode.

QosPolicy

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

Class *QosPolicy*, base for all QoS policies defined for Writers and Readers.

Subclassed by *eprosima::fastdds::dds::DataRepresentationQosPolicy*, *eprosima::fastdds::dds::DeadlineQosPolicy*, *eprosima::fastdds::dds::DestinationOrderQosPolicy*, *eprosima::fastdds::dds::DisablePositiveACKsQosPolicy*, *eprosima::fastdds::dds::DurabilityQosPolicy*, *eprosima::fastdds::dds::DurabilityServiceQosPolicy*, *eprosima::fastdds::dds::GenericDataQosPolicy*, *eprosima::fastdds::dds::HistoryQosPolicy*, *eprosima::fastdds::dds::LatencyBudgetQosPolicy*, *eprosima::fastdds::dds::LifespanQosPolicy*, *eprosima::fastdds::dds::LivelinessQosPolicy*, *eprosima::fastdds::dds::OwnershipQosPolicy*, *eprosima::fastdds::dds::OwnershipStrengthQosPolicy*, *eprosima::fastdds::dds::PartitionQosPolicy*, *eprosima::fastdds::dds::PresentationQosPolicy*, *eprosima::fastdds::dds::PublishModeQosPolicy*, *eprosima::fastdds::dds::ReliabilityQosPolicy*, *eprosima::fastdds::dds::ResourceLimitsQosPolicy*, *eprosima::fastdds::dds::TimeBasedFilterQosPolicy*, *eprosima::fastdds::dds::TransportConfigQos*, *eprosima::fastdds::dds::TransportPriorityQosPolicy*, *eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy*, *eprosima::fastdds::dds::TypeConsistencyQos*, *eprosima::fastdds::dds::TypeIdV1*, *eprosima::fastdds::dds::TypeObjectV1*, *eprosima::fastdds::dds::WireProtocolConfigQos*, *eprosima::fastdds::dds::xtypes::TypeInformation*

Public Functions

QosPolicy()
Constructor without parameters.

QosPolicy(bool *send_always*)
Constructor.

Parameters

- `send_always`: Boolean that set if the Qos need to be sent even if it is not changed

QosPolicy(**const** *QosPolicy* &*b*) = default
Copy Constructor.

Parameters

- `b`: Another instance of *QosPolicy*

~QosPolicy() = default
Destructor.

bool **send_always**() **const**
Whether it should always be sent.

Return True if it should always be sent.

void **clear**() = 0
Clears the *QosPolicy* object.

Public Members

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

ReaderDataLifecycleQosPolicy

class `eprosima::fastdds::dds::`**ReaderDataLifecycleQosPolicy**
Specifies the behavior of the *DataReader* with regards to the lifecycle of the data-instances it manages.

Warning This Qos Policy will be implemented in future releases.

Note Mutable Qos Policy

Public Functions

ReaderDataLifecycleQosPolicy()
Constructor.

~ReaderDataLifecycleQosPolicy()
Destructor.

Public Members

Duration_t **autopurge_no_writer_samples_delay**
Indicates the duration the *DataReader* must retain information regarding instances that have the instance_state NOT_ALIVE_NO_WRITERS. By default, c_TimeInfinite.

Duration_t **autopurge_disposed_samples_delay**
Indicates the duration the *DataReader* must retain information regarding instances that have the instance_state NOT_ALIVE_DISPOSED. By default, c_TimeInfinite.

ReliabilityQosPolicy

class `eprosima::fastdds::dds::`**ReliabilityQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **public** ep
Indicates the reliability of the endpoint.

Note Immutable Qos Policy

Public Functions

ReliabilityQosPolicy()
Constructor.

~ReliabilityQosPolicy() = default
Destructor.

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

Public Members

ReliabilityQosPolicyKind **kind**
Defines the reliability kind of the endpoint.

By default, BEST_EFFORT_RELIABILITY_QOS for DataReaders and RELIABLE_RELIABILITY_QOS for DataWriters.

fastrtps::Duration_t **max_blocking_time**
Defines the maximum period of time certain methods will be blocked.

Methods affected by this property are:

- *DataWriter::write*
- DataReader::takeNextData
- DataReader::readNextData By default, 100 ms.

ReliabilityQosPolicyKind

enum `dds::fastdds::eprosima::`**`ReliabilityQosPolicyKind`**

Enum ReliabilityQosPolicyKind, different kinds of reliability for *ReliabilityQosPolicy*.

Values:

enumerator BEST_EFFORT_RELIABILITY_QOS = 0x01

Indicates that it is acceptable to not retry propagation of any samples. Presumably new values for the samples are generated often enough that it is not necessary to re-send or acknowledge any samples

enumerator RELIABLE_RELIABILITY_QOS = 0x02

Specifies the Service will attempt to deliver all samples in its history. Missed samples may be retried. In steady-state (no modifications communicated via the *DataWriter*) the middleware guarantees that all samples in the *DataWriter* history will eventually be delivered to all the *DataReader* objects. Outside steady state the *HistoryQosPolicy* and *ResourceLimitsQosPolicy* will determine how samples become part of the history and whether samples can be discarded from it.

ResourceLimitsQosPolicy

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

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

Note Immutable Qos Policy

Public Functions

ResourceLimitsQosPolicy()

Constructor.

~ResourceLimitsQosPolicy() = default

Destructor.

void **clear**() **override**

Clears the *QosPolicy* object.

Public Members

int32_t **max_samples**

Specifies the maximum number of data-samples the *DataWriter* (or *DataReader*) can manage across all the instances associated with it. Represents the maximum samples the middleware can store for any one *DataWriter* (or *DataReader*). By default, 5000.

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

int32_t **max_instances**

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

int32_t **max_samples_per_instance**

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

Warning It is inconsistent for this value to be greater than max_samples.

int32_t **allocated_samples**
Number of samples currently allocated. By default, 100.

RTPSEndpointQos

class eprosima::fastdds::dds::**RTPSEndpointQos**
Qos Policy to configure the endpoint.

Public Members

fastrtps::rtps::LocatorList_t **unicast_locator_list**
Unicast locator list.

fastrtps::rtps::LocatorList_t **multicast_locator_list**
Multicast locator list.

fastrtps::rtps::LocatorList_t **remote_locator_list**
Remote locator list.

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

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

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

TimeBasedFilterQosPolicy

class eprosima::fastdds::dds::**TimeBasedFilterQosPolicy** : **public** eprosima::fastdds::dds::Parameter_t, **publ**
Filter that allows a *DataReader* to specify that it is interested only in (potentially) a subset of the values of the data. The filter states that the *DataReader* does not want to receive more than one value each minimum_separation, regardless of how fast the changes occur. It is inconsistent for a *DataReader* to have a minimum_separation longer than its Deadline period.

Warning This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Note Mutable Qos Policy

Public Functions

TimeBasedFilterQosPolicy()
Constructor.

~TimeBasedFilterQosPolicy() = default
Destructor.

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

Public Members

fastrtps::Duration_t **minimum_separation**
Minimum interval between samples. By default, c_TimeZero (the *DataReader* is interested in all values)

TopicDataQosPolicy

class TopicDataQosPolicy : **public** eprosima::fastdds::dds::*GenericDataQosPolicy*
Class derived from *GenericDataQosPolicy*.

The purpose of this QoS is to allow the application to attach additional information to the created *Topic* such that when a remote application discovers their existence it can examine the information and use it in an application-defined way.

In combination with the listeners on the *DataReader* and *DataWriter* as well as by means of operations such as ignore_topic,these QoS can assist an application to extend the provided QoS.

TransportConfigQos

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

Public Functions

TransportConfigQos()
Constructor.

~TransportConfigQos() = default
Destructor.

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

Public Members

std::vector<std::shared_ptr<fastdds::rtps::TransportDescriptorInterface>> **user_transports**
User defined transports to use alongside or in place of builtins.

bool **use_builtin_transports**
Set as false to disable the default UDPv4 implementation. By default, true.

uint32_t **send_socket_buffer_size**
Send socket buffer size for the send resource.

Zero value indicates to use default system buffer size. By default, 0.

uint32_t **listen_socket_buffer_size**
Listen socket buffer for all listen resources.

Zero value indicates to use default system buffer size. By default, 0.

TransportPriorityQosPolicy

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

This policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data.

Warning This *QosPolicy* can be defined and is transmitted to the rest of the network but is not implemented in this version.

Note Mutable Qos Policy

Public Functions

TransportPriorityQosPolicy()

Constructor.

~TransportPriorityQosPolicy() = default

Destructor.

void **clear**() **override**

Clears the *QosPolicy* object.

Public Members

uint32_t **value**

Priority By default, 0.

TypeConsistencyEnforcementQosPolicy

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

The *TypeConsistencyEnforcementQosPolicy* defines the rules for determining whether the type used to publish a given data stream is consistent with that used to subscribe to it. It applies to DataReaders.

Note Immutable Qos Policy

Public Functions

TypeConsistencyEnforcementQosPolicy()

Constructor.

~TypeConsistencyEnforcementQosPolicy() **override** = default

Destructor.

void **clear**() **override**

Clears the *QosPolicy* object.

Public Members

TypeConsistencyKind **m_kind**
TypeConsistencyKind. By default, ALLOW_TYPE_COERCION.

bool **m_ignore_sequence_bounds**
This option controls whether sequence bounds are taken into consideration for type assignability. If the option is set to TRUE, sequence bounds (maximum lengths) are not considered as part of the type assignability. This means that a T2 sequence type with maximum length L2 would be assignable to a T1 sequence type with maximum length L1, even if L2 is greater than L1. If the option is set to false, then sequence bounds are taken into consideration for type assignability and in order for T1 to be assignable from T2 it is required that L1>= L2. By default, true.

bool **m_ignore_string_bounds**
This option controls whether string bounds are taken into consideration for type assignability. If the option is set to TRUE, string bounds (maximum lengths) are not considered as part of the type assignability. This means that a T2 string type with maximum length L2 would be assignable to a T1 string type with maximum length L1, even if L2 is greater than L1. If the option is set to false, then string bounds are taken into consideration for type assignability and in order for T1 to be assignable from T2 it is required that L1>= L2. By default, true.

bool **m_ignore_member_names**
This option controls whether member names are taken into consideration for type assignability. If the option is set to TRUE, member names are considered as part of assignability in addition to member IDs (so that members with the same ID also have the same name). If the option is set to FALSE, then member names are not ignored. By default, false.

bool **m_prevent_type_widening**
This option controls whether type widening is allowed. If the option is set to FALSE, type widening is permitted. If the option is set to TRUE,it shall cause a wider type to not be assignable to a narrower type. By default, false.

bool **m_force_type_validation**
This option requires type information to be available in order to complete matching between a *DataWriter* and *DataReader* when set to TRUE, otherwise matching can occur without complete type information when set to FALSE. By default, false.

TypeConsistencyKind

enum `dds::fastdds::eprosima::`**TypeConsistencyKind**
Values:

enumerator DISALLOW_TYPE_COERCION
The *DataWriter* and the *DataReader* must support the same data type in order for them to communicate.

enumerator ALLOW_TYPE_COERCION
The *DataWriter* and the *DataReader* need not support the same data type in order for them to communicate as long as the reader's type is assignable from the writer's type.

UserDataQosPolicy

class UserDataQosPolicy : **public** eprosima::fastdds::dds::*GenericDataQosPolicy*

Class derived from *GenericDataQosPolicy*.

The purpose of this QoS is to allow the application to attach additional information to the created *Entity* objects such that when a remote application discovers their existence it can access that information and use it for its own purposes.

One possible use of this QoS is to attach security credentials or some other information that can be used by the remote application to authenticate the source.

WireProtocolConfigQos

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

Qos Policy that configures the wire protocol.

Public Functions

WireProtocolConfigQos()

Constructor.

~WireProtocolConfigQos() = default

Destructor.

void **clear**() **override**

Clears the *QosPolicy* object.

Public Members

fastrtps::rtps::GuidPrefix_t **prefix**

Optionally allows user to define the GuidPrefix_t.

int32_t **participant_id**

Participant ID By default, -1.

fastrtps::rtps::BuiltinAttributes **builtin**

Builtin parameters.

fastrtps::rtps::PortParameters **port**

Port Parameters.

fastrtps::rtps::ThroughputControllerDescriptor **throughput_controller**

Throughput controller parameters. Leave default for uncontrolled flow.

fastrtps::rtps::LocatorList_t **default_unicast_locator_list**

Default list of Unicast Locators to be used for any Endpoint defined inside this RTPSParticipant in the case that it was defined with NO UnicastLocators. At least ONE locator should be included in this list.

fastrtps::rtps::LocatorList_t **default_multicast_locator_list**

Default list of Multicast Locators to be used for any Endpoint defined inside this RTPSParticipant in the case that it was defined with NO UnicastLocators. This is usually left empty.

WriterDataLifecycleQosPolicy

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

Specifies the behavior of the *DataWriter* with regards to the lifecycle of the data-instances it manages.

Warning This Qos Policy will be implemented in future releases.

Note Mutable Qos Policy

Public Functions

WriterDataLifecycleQosPolicy()

Constructor.

~WriterDataLifecycleQosPolicy()

Destructor.

Public Members

bool **autodispose_unregistered_instances**

Controls whether a *DataWriter* will automatically dispose instances each time they are unregistered. The setting autodispose_unregistered_instances = TRUE indicates that unregistered instances will also be considered disposed. By default, true.

WriterResourceLimitsQos

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

Qos Policy to configure the limit of the writer resources.

Public Functions

WriterResourceLimitsQos()

Constructor.

~WriterResourceLimitsQos() = default

Destructor.

Public Members

fastrtps::ResourceLimitedContainerConfig **matched_subscriber_allocation**

Matched subscribers allocation limits.

Status

BaseStatus

struct eprosima::fastdds::dds::**BaseStatus**

A struct storing the base status.

Public Members

int32_t **total_count** = 0

Total cumulative count.

int32_t **total_count_change** = 0

Increment since the last time the status was read.

DeadlineMissedStatus

struct eprosima::fastdds::dds::**DeadlineMissedStatus**

A struct storing the deadline status.

Public Functions

DeadlineMissedStatus()

Constructor.

~DeadlineMissedStatus()

Destructor.

Public Members

uint32_t **total_count**

Total cumulative number of offered deadline periods epased during which a writer failed to provide data.

Missed deadlines accumulate, that is, each deadline period the total_count will be incremented by 1

uint32_t **total_count_change**

The change in total_count since the last time the listener was called or the status was read.

fastrtps::rtps::InstanceHandle_t **last_instance_handle**

Handle to the last instance missing the deadline.

IncompatibleQosStatus

struct eprosima::fastdds::dds::**IncompatibleQosStatus**

A struct storing the requested incompatible QoS status.

Public Members

uint32_t **total_count** = 0
Total cumulative number of times the concerned writer discovered a reader for the same topic.

The requested QoS is incompatible with the one offered by the writer

uint32_t **total_count_change** = 0
The change in total_count since the last time the listener was called or the status was read.

QosPolicyId_t **last_policy_id** = INVALID_QOS_POLICY_ID
The id of the policy that was found to be incompatible the last time an incompatibility is detected.

QosPolicyCountSeq **policies**
A list of *QosPolicyCount*.

InconsistentTopicStatus

using `dds::fastdds::eprosima::`**InconsistentTopicStatus** = BaseStatus
Alias of *BaseStatus*.

LivelinessChangedStatus

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

Public Members

int32_t **alive_count** = 0
The total number of currently active publishers that write the topic read by the subscriber.

This count increases when a newly matched publisher asserts its liveliness for the first time or when a publisher previously considered to be not alive reasserts its liveliness. The count decreases when a publisher considered alive fails to assert its liveliness and becomes not alive, whether because it was deleted normally or for some other reason

int32_t **not_alive_count** = 0
The total count of current publishers that write the topic read by the subscriber that are no longer asserting their liveliness.

This count increases when a publisher considered alive fails to assert its liveliness and becomes not alive for some reason other than the normal deletion of that publisher. It decreases when a previously not alive publisher either reasserts its liveliness or is deleted normally

int32_t **alive_count_change** = 0
The change in the alive_count since the last time the listener was called or the status was read.

int32_t **not_alive_count_change** = 0
The change in the not_alive_count since the last time the listener was called or the status was read.

fastrtps::rtps::InstanceHandle_t **last_publication_handle**
Handle to the last publisher whose change in liveliness caused this status to change.

MatchedStatus

struct `eprosima::fastdds::dds::`**`MatchedStatus`**

A structure storing a matching status.

Subclassed by *eprosima::fastdds::dds::PublicationMatchedStatus*, *eprosima::fastdds::dds::SubscriptionMatchedStatus*

Public Functions

`MatchedStatus``()` = default

Constructor.

`~MatchedStatus``()` = default

Destructor.

Public Members

int32_t **`total_count`** = 0

Total cumulative count the concerned reader discovered a match with a writer.

It found a writer for the same topic with a requested QoS that is compatible with that offered by the reader

int32_t **`total_count_change`** = 0

The change in total_count since the last time the listener was called or the status was read.

int32_t **`current_count`** = 0

The number of writers currently matched to the concerned reader.

int32_t **`current_count_change`** = 0

The change in current_count since the last time the listener was called or the status was read.

OfferedDeadlineMissedStatus

typedef DeadlineMissedStatus `dds::fastdds::eprosima::`**`OfferedDeadlineMissedStatus`**

Typedef of *DeadlineMissedStatus*.

OfferedIncompatibleQosStatus

using `dds::fastdds::eprosima::`**`OfferedIncompatibleQosStatus`** = IncompatibleQosStatus

Alias of *IncompatibleQosStatus*.

PublicationMatchedStatus

struct `eprosima::fastdds::dds::`**`PublicationMatchedStatus`** : **public** eprosima::fastdds::dds::*MatchedStatus*

A structure storing the publication status.

Public Members

eprosima::fastrtps::rtps::InstanceHandle_t **last_subscription_handle**
Handle to the last reader that matched the writer causing the status to change.

QosPolicyCount

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

Public Functions

QosPolicyCount(QosPolicyId_t *id*, int32_t *c*)
Constructor.

Public Members

QosPolicyId_t **policy_id** = INVALID_QOS_POLICY_ID
The id of the policy.

uint32_t **count** = 0
Total number of times that the concerned writer discovered a reader for the same topic.

The requested QoS is incompatible with the one offered by the writer

QosPolicyCountSeq

using dds::fastdds::eprosima::**QosPolicyCountSeq** = std::vector<QosPolicyCount>
Alias of std::vector<QosPolicyCount>

QosPolicyId_t

Warning: doxygentypedef: Cannot find typedef "eprosima::fastdds::dds::QosPolicyId_t" in doxygen xml output for project "FastDDS" from directory: /home/docs/checkouts/readthedocs.org/user_builds/eprosima-fast-rtps/checkouts/v2.0.0/build/code/doxygen/xml

RequestedDeadlineMissedStatus

typedef DeadlineMissedStatus dds::fastdds::eprosima::**RequestedDeadlineMissedStatus**
Typedef of *DeadlineMissedStatus*.

RequestedIncompatibleQosStatus

using dds::fastdds::eprosima::**RequestedIncompatibleQosStatus** = IncompatibleQosStatus

Alias of *IncompatibleQosStatus*.

LivelinessLostStatus

using dds::fastdds::eprosima::**LivelinessLostStatus** = BaseStatus

Alias of *BaseStatus*.

SampleLostStatus

using dds::fastdds::eprosima::**SampleLostStatus** = BaseStatus

Alias of *BaseStatus*.

SampleRejectedStatus

struct eprosima::fastdds::dds::**SampleRejectedStatus**

A struct storing the sample lost status.

Public Members

uint32_t **total_count** = 0

Total cumulative count of samples rejected by the *DataReader*.

uint32_t **total_count_change** = 0

The incremental number of samples rejected since the last time the listener was called or the status was read.

SampleRejectedStatusKind **last_reason** = NOT_REJECTED

Reason for rejecting the last sample rejected. If no samples have been rejected, the reason is the special value NOT_REJECTED.

fastrtps::rtps::InstanceHandle_t **last_instance_handle**

Handle to the instance being updated by the last sample that was rejected.

SampleRejectedStatusKind

enum dds::fastdds::eprosima::**SampleRejectedStatusKind**

An enum with the possible values for the sample rejected reason.

Values:

enumerator NOT_REJECTED

Default value.

enumerator REJECTED_BY_INSTANCES_LIMIT

Exceeds the max_instance limit.

enumerator REJECTED_BY_SAMPLES_LIMIT

Exceeds the max_samples limit.

enumerator REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT
Exceeds the max_samples_per_instance limit.

StatusMask

class eprosima::fastdds::dds::**StatusMask** : **public** std::bitset<FASTDDS_STATUS_COUNT>
StatusMask is a bitmap or bitset field.

This bitset is used to:

- determine which listener functions to call
- set conditions in dds::core::cond::StatusCondition
- indicate status changes when calling dds::core::Entity::status_changes

Public Types

typedef std::bitset<FASTDDS_STATUS_COUNT> **MaskType**
Convenience typedef for std::bitset<FASTDDS_STATUS_COUNT>.

Public Functions

StatusMask()
Construct an *StatusMask* with no flags set.

StatusMask(uint32_t *mask*)
Construct an *StatusMask* with an uint32_t bit mask.

Parameters

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

StatusMask &**operator<<**(**const** *StatusMask* &*mask*)
Add given *StatusMask* bits into this *StatusMask* bitset.

Return *StatusMask* this

StatusMask &**operator>>**(**const** *StatusMask* &*mask*)
Remove given *StatusMask* bits into this *StatusMask* bitset.

Return *StatusMask* this

bool **is_active**(*StatusMask status*) **const**
Checks if the status passed as parameter is 1 in the actual *StatusMask*.

Return true if the status is active and false if not

Parameters

- `status`: Status that need to be checked

Public Static Functions

StatusMask **all**()
Get all StatusMasks

Return *StatusMask* all

StatusMask **none**()
Get no StatusMasks

Return *StatusMask* none

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

Return *StatusMask* inconsistent_topic

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

Return *StatusMask* offered_deadline_missed

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

Return *StatusMask* requested_deadline_missed

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

Return *StatusMask* offered_incompatible_qos

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

Return *StatusMask* requested_incompatible_qos

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

Return *StatusMask* sample_lost

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

Return *StatusMask* sample_rejected

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

Return *StatusMask* data_on_readers

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

Return statusmask data_available

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

Return *StatusMask* liveliness_lost

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

Return *StatusMask* liveliness_changed

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

Return *StatusMask* publication_matched

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

Return *StatusMask* subscription_matched

FASTDDS_STATUS_COUNT
Alias of size_t(16)

SubscriptionMatchedStatus

struct eprosima::fastdds::dds::**SubscriptionMatchedStatus** : **public** eprosima::fastdds::dds::*MatchedStatus*
A structure storing the subscription status.

Public Members

eprosima::fastrtps::rtps::InstanceHandle_t **last_publication_handle**
Handle to the last writer that matched the reader causing the status change.

Domain

DomainParticipant

class eprosima::fastdds::dds::**DomainParticipant** : **public** eprosima::fastdds::dds::*Entity*
Class *DomainParticipant* used to group Publishers and Subscribers into a single working unit.

Public Functions

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

Return RETCODE_OK

Parameters

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

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

Return A reference to the *DomainParticipantQos*

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

Return RETCODE_OK

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

Return RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *DomainParticipantQos* to be set

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

Return RETCODE_OK

Parameters

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

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

Return RETCODE_OK

Parameters

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

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

Return *DomainParticipantListener* pointer

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

Return Pointer to the created *Publisher*.

Parameters

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

Publisher ***create_publisher_with_profile**(**const** std::string *&profile_name*, *PublisherListener* **listener* = nullptr, **const** *StatusMask* *&mask* = *StatusMask*::*all*())

Create a *Publisher* in this Participant.

Return Pointer to the created *Publisher*.

Parameters

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

ReturnCode_t **delete_publisher**(*Publisher* **publisher*)

Deletes an existing *Publisher*.

Return RETCODE_PRECONDITION_NOT_MET if the publisher does not belong to this participant or if it has active DataWriters, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Parameters

- `publisher`: to be deleted.

Subscriber ***create_subscriber**(**const** *SubscriberQos* *&qos*, *SubscriberListener* **listener* = nullptr, **const** *StatusMask* *&mask* = *StatusMask*::*all*())

Create a *Subscriber* in this Participant.

Return Pointer to the created *Subscriber*.

Parameters

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

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

Create a *Subscriber* in this Participant.

Return Pointer to the created *Subscriber*.

Parameters

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

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

Deletes an existing *Subscriber*.

Return RETCODE_PRECONDITION_NOT_MET if the subscriber does not belong to this participant or if it has active DataReaders, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Parameters

- `subscriber`: to be deleted.

ReturnCode_t **`register_type`**(*TypeSupport type*, **const** std::string *&type_name*)

Register a type in this participant.

Return RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there is another *TypeSupport* with the same name and RETCODE_OK if it is correctly registered.

Parameters

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

ReturnCode_t **`register_type`**(*TypeSupport type*)

Register a type in this participant.

Return RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there is another *TypeSupport* with the same name and RETCODE_OK if it is correctly registered.

Parameters

- `type`: *TypeSupport*.

ReturnCode_t **`unregister_type`**(**const** std::string *&typeName*)

Unregister a type in this participant.

Return RETCODE_BAD_PARAMETER if the size of the name is 0, RERCODE_PRECONDITION_NOT_MET if there are entities using that *TypeSupport* and RETCODE_OK if it is correctly unregistered.

Parameters

- `typeName`: Name of the type

Topic ***`create_topic`**(**const** std::string *&topic_name*, **const** std::string *&type_name*, **const** *TopicQos &qos*, *TopicListener *listener* = nullptr, **const** *StatusMask &mask* = *StatusMask*::*all*())

Create a *Topic* in this Participant.

Return Pointer to the created *Topic*.

Parameters

- `topic_name`: Name of the *Topic*.
- `type_name`: Data type of the *Topic*.
- `qos`: QoS of the *Topic*.
- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all)

Topic ***`create_topic_with_profile`**(**const** std::string *&topic_name*, **const** std::string *&type_name*, **const** std::string *&profile_name*, *TopicListener *listener* = nullptr, **const** *StatusMask &mask* = *StatusMask*::*all*())

Create a *Topic* in this Participant.

Return Pointer to the created *Topic*.

Parameters

- `topic_name`: Name of the *Topic*.
- `type_name`: Data type of the *Topic*.
- `profile_name`: *Topic* profile name.
- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all)

ReturnCode_t **`delete_topic`**(*Topic* **topic*)

Deletes an existing *Topic*.

Return RETCODE_BAD_PARAMETER if the topic passed is a nullptr, RETCODE_PRECONDITION_NOT_MET if the topic does not belong to this participant or if it is referenced by any entity and ETCODE_OK if the *Topic* was deleted.

Parameters

- `topic`: to be deleted.

TopicDescription ***`lookup_topicdescription`**(**`const`** std::string &*topic_name*) **`const`**

Looks up an existing, locally created *TopicDescription*, based on its name. May be called on a disabled participant.

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

Remark UNSAFE. It is unsafe to lookup a topic description while another thread is creating a topic.

Parameters

- `topic_name`: Name of the *TopicDescription* to search for.

DomainId_t **`get_domain_id`**() **`const`**

This operation retrieves the domain_id used to create the *DomainParticipant*. The domain_id identifies the DDS domain to which the *DomainParticipant* belongs.

Return The Participant's domain_id

ReturnCode_t **`assert_liveliness`**()

This operation manually asserts the liveliness of the *DomainParticipant*. This is used in combination with the LIVELINESS QoS policy to indicate to the Service that the entity remains active.

This operation needs to only be used if the *DomainParticipant* contains *DataWriter* entities with the LIVELINESS set to MANUAL_BY_PARTICIPANT and it only affects the liveliness of those *DataWriter* entities. Otherwise, it has no effect.

Note Writing data via the write operation on a *DataWriter* asserts liveliness on the *DataWriter* itself and its *DomainParticipant*. Consequently the use of assert_liveliness is only needed if the application is not writing data regularly.

Return RETCODE_OK if the liveliness was asserted, RETCODE_ERROR otherwise.

ReturnCode_t **`set_default_publisher_qos`**(**`const`** *PublisherQos* &*qos*)

This operation sets a default value of the *Publisher* QoS policies which will be used for newly created *Publisher* entities in the case where the QoS policies are defaulted in the create_publisher operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value PUBLISHER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_publisher_qos operation had never been called.

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

Parameters

- `qos`: *PublisherQos* to be set

const *PublisherQos* &**get_default_publisher_qos**() **const**

This operation retrieves the default value of the *Publisher* QoS, that is, the QoS policies which will be used for newly created *Publisher* entities in the case where the QoS policies are defaulted in the create_publisher operation.

The values retrieved get_default_publisher_qos will match the set of values specified on the last successful call to set_default_publisher_qos, or else, if the call was never made, the default values.

Return Current default publisher qos.

ReturnCode_t **get_default_publisher_qos**(*PublisherQos &qos*) **const**

This operation retrieves the default value of the *Publisher* QoS, that is, the QoS policies which will be used for newly created *Publisher* entities in the case where the QoS policies are defaulted in the create_publisher operation.

The values retrieved get_default_publisher_qos will match the set of values specified on the last successful call to set_default_publisher_qos, or else, if the call was never made, the default values.

Return RETCODE_OK

Parameters

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

ReturnCode_t **get_publisher_qos_from_profile**(**const** std::string *&profile_name*, *PublisherQos &qos*) **const**

Fills the *PublisherQos* with the values of the XML profile.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

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

ReturnCode_t **set_default_subscriber_qos**(**const** *SubscriberQos &qos*)

This operation sets a default value of the *Subscriber* QoS policies that will be used for newly created *Subscriber* entities in the case where the QoS policies are defaulted in the create_subscriber operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value SUBSCRIBER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_subscriber_qos operation had never been called.

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

Parameters

- `qos`: *SubscriberQos* to be set

const *SubscriberQos* &**get_default_subscriber_qos**() **const**

This operation retrieves the default value of the *Subscriber* QoS, that is, the QoS policies which will be used for newly created *Subscriber* entities in the case where the QoS policies are defaulted in the create_subscriber operation.

The values retrieved get_default_subscriber_qos will match the set of values specified on the last successful call to set_default_subscriber_qos, or else, if the call was never made, the default values.

Return Current default subscriber qos.

ReturnCode_t **get_default_subscriber_qos**(*SubscriberQos &qos*) **const**

This operation retrieves the default value of the *Subscriber* QoS, that is, the QoS policies which will be used for newly created *Subscriber* entities in the case where the QoS policies are defaulted in the create_subscriber operation.

The values retrieved get_default_subscriber_qos will match the set of values specified on the last successful call to set_default_subscriber_qos, or else, if the call was never made, the default values.

Return RETCODE_OK

Parameters

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

ReturnCode_t **get_subscriber_qos_from_profile**(**const** std::string *&profile_name*, *SubscriberQos &qos*) **const**

Fills the *SubscriberQos* with the values of the XML profile.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

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

ReturnCode_t **set_default_topic_qos**(**const** *TopicQos &qos*)

This operation sets a default value of the *Topic* QoS policies which will be used for newly created *Topic* entities in the case where the QoS policies are defaulted in the create_topic operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value TOPIC_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_topic_qos operation had never been called.

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

Parameters

- `qos`: *TopicQos* to be set

const *TopicQos* &**get_default_topic_qos**() **const**

This operation retrieves the default value of the *Topic* QoS, that is, the QoS policies that will be used for newly created *Topic* entities in the case where the QoS policies are defaulted in the create_topic operation.

The values retrieved get_default_topic_qos will match the set of values specified on the last successful call to set_default_topic_qos, or else, TOPIC_QOS_DEFAULT if the call was never made.

Return Current default topic qos.

ReturnCode_t **get_default_topic_qos**(*TopicQos &qos*) **const**

This operation retrieves the default value of the *Topic* QoS, that is, the QoS policies that will be used for newly created *Topic* entities in the case where the QoS policies are defaulted in the create_topic operation.

The values retrieved get_default_topic_qos will match the set of values specified on the last successful call to set_default_topic_qos, or else, TOPIC_QOS_DEFAULT if the call was never made.

Return RETCODE_OK

Parameters

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

ReturnCode_t **`get_topic_qos_from_profile`**(**const** std::string *&profile_name*, *TopicQos &qos*) **const**

Fills the *TopicQos* with the values of the XML profile.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

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

bool **`contains_entity`**(**const** fastrtps::rtps::InstanceHandle_t *&handle*, bool *recursive* = true) **const**

This operation checks whether or not the given handle represents an *Entity* that was created from the *DomainParticipant*.

Return True if entity is contained. False otherwise.

Parameters

- `handle`: InstanceHandle of the entity to look for.
- `recursive`: The containment applies recursively. That is, it applies both to entities (*TopicDescription*, *Publisher*, or *Subscriber*) created directly using the *DomainParticipant* as well as entities created using a contained *Publisher*, or *Subscriber* as the factory, and so forth. (default: true)

ReturnCode_t **`get_current_time`**(fastrtps::Time_t *&current_time*) **const**

This operation returns the current value of the time that the service uses to time-stamp data-writes and to set the reception-timestamp for the data-updates it receives.

Return RETCODE_OK

Parameters

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

TypeSupport **`find_type`**(**const** std::string *&type_name*) **const**

This method gives access to a registered type based on its name.

Return *TypeSupport* corresponding to the type_name

Parameters

- `type_name`: Name of the type

const fastrtps::rtps::InstanceHandle_t &**`get_instance_handle`**() **const**

Returns the *DomainParticipant*'s handle.

Return InstanceHandle of this *DomainParticipant*.

const fastrtps::rtps::GUID_t &**`guid`**() **const**

Getter for the Participant GUID.

Return A reference to the GUID

std::vector<std::string> **`get_participant_names`**() **const**

Getter for the participant names.

Return Vector with the names

bool **new_remote_endpoint_discovered**(**const** fastrtps::rtps::GUID_t *&partguid*, uint16_t *userId*, fastrtps::rtps::EndpointKind_t *kind*)

This method can be used when using a StaticEndpointDiscovery mechanism different that the one included in FastRTPS, for example when communicating with other implementations. It indicates the Participant that an Endpoint from the XML has been discovered and should be activated.

Return True if correctly found and activated.

Parameters

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

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

Getter for the resource event.

Return A reference to the resource event

fastrtps::rtps::SampleIdentity **get_type_dependencies**(**const** fastrtps::types::TypeIdentifierSeq *&in*) **const**

When a *DomainParticipant* receives an incomplete list of TypeIdentifiers in a PublicationBuiltinTopicData or SubscriptionBuiltinTopicData, it may request the additional type dependencies by invoking the getTypeDependencies operation.

Return SampleIdentity

Parameters

- `in`: TypeIdentifier sequence

fastrtps::rtps::SampleIdentity **get_types**(**const** fastrtps::types::TypeIdentifierSeq *&in*) **const**

A *DomainParticipant* may invoke the operation getTypes to retrieve the TypeObjects associated with a list of TypeIdentifiers.

Return SampleIdentity

Parameters

- `in`: TypeIdentifier sequence

ReturnCode_t **register_remote_type**(**const** fastrtps::types::TypeInformation *&type_information*, **const** std::string *&type_name*, std::function<void)**const** std::string &name, **const** fastrtps::types::DynamicType_ptr type

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

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

Parameters

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

~DomainParticipant()
Destructor.

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

Return true if any, false otherwise.

DomainParticipantFactory

class eprosima::fastdds::dds::**DomainParticipantFactory**
Class *DomainParticipantFactory*

Public Functions

DomainParticipant ***create_participant**(DomainId_t *domain_id*, **const** *DomainParticipantQos* *&qos*, *DomainParticipantListener* **listener* = nullptr, **const** *StatusMask* *&mask* = *StatusMask*::*all*())
Create a Participant.

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

Parameters

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

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

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

Parameters

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

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

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

Parameters

- profile_name: Participant profile name.

- `listener`: *DomainParticipantListener* Pointer (default: nullptr)
- `mask`: *StatusMask* Reference (default: all)

DomainParticipant ***lookup_participant**(DomainId_t *domain_id*) **const**

This operation retrieves a previously created *DomainParticipant* belonging to specified domain_id. If no such *DomainParticipant* exists, the operation will return 'nullptr'. If multiple *DomainParticipant* entities belonging to that domain_id exist, then the operation will return one of them. It is not specified which one.

Return previously created *DomainParticipant* within the specified domain

Parameters

- `domain_id`:

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

Returns all participants that belongs to the specified domain_id.

Return previously created DomainParticipants within the specified domain

Parameters

- `domain_id`:

ReturnCode_t **get_default_participant_qos**(*DomainParticipantQos* &*qos*) **const**

This operation retrieves the default value of the *DomainParticipant* QoS, that is, the QoS policies which will be used for newly created *DomainParticipant* entities in the case where the QoS policies are defaulted in the create_participant operation. The values retrieved get_default_participant_qos will match the set of values specified on the last successful call to set_default_participant_qos, or else, if the call was never made, the default values.

Return RETCODE_OK

Parameters

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

const *DomainParticipantQos* &**get_default_participant_qos**() **const**

This operation retrieves the default value of the *DomainParticipant* QoS, that is, the QoS policies which will be used for newly created *DomainParticipant* entities in the case where the QoS policies are defaulted in the create_participant operation. The values retrieved get_default_participant_qos will match the set of values specified on the last successful call to set_default_participant_qos, or else, if the call was never made, the default values.

Return A reference to the default *DomainParticipantQos*

ReturnCode_t **set_default_participant_qos**(**const** *DomainParticipantQos* &*qos*)

This operation sets a default value of the *DomainParticipant* QoS policies which will be used for newly created *DomainParticipant* entities in the case where the QoS policies are defaulted in the create_participant operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

The special value PARTICIPANT_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_participant_qos operation had never been called.

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

Parameters

- `qos`: *DomainParticipantQos* to be set

ReturnCode_t **`get_participant_qos_from_profile`**(**const** std::string *&profile_name*, *DomainParticipantQos &qos*) **const**

Fills the *DomainParticipantQos* with the values of the XML profile.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

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

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

Remove a Participant and all associated publishers and subscribers.

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

Parameters

- `part`: Pointer to the participant.

ReturnCode_t **`load_profiles`**()

Load profiles from default XML file.

Return RETCODE_OK

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

Load profiles from XML file.

Return RETCODE_OK if it is correctly loaded, RETCODE_ERROR otherwise.

Parameters

- `xml_profile_file`: XML profile file.

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

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

Return RETCODE_OK

Parameters

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

ReturnCode_t **`set_qos`**(**const** *DomainParticipantFactoryQos &qos*)

This operation sets the value of the *DomainParticipantFactory* QoS policies. These policies control the behavior of the object a factory for entities.

Note that despite having QoS, the *DomainParticipantFactory* is not an *Entity*.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return INCONSISTENT_POLICY.

Return RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

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

Public Static Functions

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

Return The *DomainParticipantFactory* singleton.

DomainParticipantFactoryQos

class eprosima::fastdds::dds::**DomainParticipantFactoryQos**
Class *DomainParticipantFactoryQos*, contains all the possible Qos that can be set for a determined participant. Please consult each of them to check for implementation details and default values.

Public Functions

DomainParticipantFactoryQos()
Constructor.

~DomainParticipantFactoryQos()
Destructor.

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

Return *EntityFactoryQosPolicy* reference

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

Return *EntityFactoryQosPolicy* reference

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

Parameters

- `entity_factory`: *EntityFactoryQosPolicy*

DomainParticipantListener

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

Public Functions

DomainParticipantListener()
Constructor.

~DomainParticipantListener()
Destructor.

void **on_participant_discovery**(*DomainParticipant* **participant*, fastrtps::rtps::ParticipantDiscoveryInfo &&*info*)
This method is called when a new Participant is discovered, or a previously discovered participant changes its QOS or is removed.

Parameters

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

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

This method is called when a new Participant is authenticated.

Parameters

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

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

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

Parameters

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

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

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

Parameters

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

void **on_type_discovery**(*DomainParticipant* *participant, **const** fastrtps::rtps::SampleIdentity &request_sample_id, **const** fastrtps::string_255 &topic, **const** fastrtps::types::TypeIdentifier *identifier, **const** fastrtps::types::TypeObject *object, fastrtps::types::DynamicType_ptr dyn_type)

This method is called when a participant discovers a new Type The ownership of all object belongs to the caller so if needs to be used after the method ends, a full copy should be perform (except for dyn_type due to its shared_ptr nature.

For example: fastrtps::types::TypeIdentifier new_type_id = *identifier;

void **on_type_dependencies_reply**(*DomainParticipant* *participant, **const** fastrtps::rtps::SampleIdentity &request_sample_id, **const** fastrtps::types::TypeIdentifierWithSizeSeq &dependencies)

This method is called when the typelookup client received a reply to a getTypeDependencies request.

The user may want to retrieve these new types using the getTypes request and create a new DynamicType using the retrieved TypeObject.

void **on_type_information_received**(*DomainParticipant* *participant, **const** fastrtps::string_255 topic_name, **const** fastrtps::string_255 type_name, **const** fastrtps::types::TypeInformation &type_information)

This method is called when a participant receives a TypeInformation while discovering another participant.

DomainParticipantQos

class `eprosima::fastdds::dds::`**`DomainParticipantQos`**

Class *DomainParticipantQos*, contains all the possible Qos that can be set for a determined participant. Please consult each of them to check for implementation details and default values.

Public Functions

`DomainParticipantQos``()`

Constructor.

`~DomainParticipantQos``()`

Destructor.

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

Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

UserDataQosPolicy &**`user_data`**`()`

Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

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

Setter for *UserDataQosPolicy*

Parameters

- `value`: *UserDataQosPolicy*

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

Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

EntityFactoryQosPolicy &**`entity_factory`**`()`

Getter for *EntityFactoryQosPolicy*

Return *EntityFactoryQosPolicy* reference

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

Setter for *EntityFactoryQosPolicy*

Parameters

- `value`: *EntityFactoryQosPolicy*

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

Getter for ParticipantResourceLimitsQos

Return ParticipantResourceLimitsQos reference

ParticipantResourceLimitsQos &**`allocation`**`()`

Getter for ParticipantResourceLimitsQos

Return ParticipantResourceLimitsQos reference

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

Setter for ParticipantResourceLimitsQos

Parameters

- `allocation`: ParticipantResourceLimitsQos

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

Return PropertyPolicyQos reference

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

Return PropertyPolicyQos reference

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

Parameters

- properties: PropertyPolicyQos

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

Return *WireProtocolConfigQos* reference

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

Return *WireProtocolConfigQos* reference

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

Parameters

- wire_protocol: *WireProtocolConfigQos*

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

Return *TransportConfigQos* reference

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

Return *TransportConfigQos* reference

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

Parameters

- transport: *TransportConfigQos*

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

Return name

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

Return name

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

Return value New name to be set

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

Publisher

DataWriter

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

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

Public Functions

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

This operation enables the *DataWriter*.

Return RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the *Publisher* creating this *DataWriter* is not enabled.

bool **write**(void **data*)

Write data to the topic.

Return True if correct, false otherwise

Parameters

- `data`: Pointer to the data

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

Write data with params to the topic.

Return True if correct, false otherwise

Parameters

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

ReturnCode_t **write**(void **data*, **const** fastrtps::rtps::InstanceHandle_t &*handle*)

Write data with handle.

The special value HANDLE_NIL can be used for the parameter handle.This indicates that the identity of the instance should be automatically deduced from the instance_data (by means of the key).

Return RETCODE_PRECONDITION_NOT_MET if the handle introduced does not match with the one associated to the data, RETCODE_OK if the data is correctly sent and RETCODE_ERROR otherwise.

Parameters

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

fastrtps::rtps::InstanceHandle_t **register_instance**(void **instance*)

Informs that the application will be modifying a particular instance.

It gives an opportunity to the middleware to pre-configure itself to improve performance.

Return Handle containing the instance's key. This handle could be used in successive `write` or `dispose` operations. In case of error, HANDLE_NIL will be returned.

Parameters

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

ReturnCode_t **unregister_instance**(void **instance*, **const** fastrtps::rtps::InstanceHandle_t *&handle*)

This operation reverses the action of `register_instance`.

It should only be called on an instance that is currently registered. Informs the middleware that the *DataWriter* is not intending to modify any more of that data instance. Also indicates that the middleware can locally remove all information regarding that instance.

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

Parameters

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

const fastrtps::rtps::GUID_t &**guid**()

Returns the *DataWriter*'s GUID

Return Reference to the *DataWriter* GUID

fastrtps::rtps::InstanceHandle_t **get_instance_handle**() **const**

Returns the *DataWriter*'s InstanceHandle

Return Copy of the *DataWriter* InstanceHandle

TypeSupport **get_type**() **const**

Get data type associated to the *DataWriter*

Return Copy of the *TypeSupport*

ReturnCode_t **wait_for_acknowledgments**(**const** fastrtps::Duration_t *&max_wait*)

Waits the current thread until all writers have received their acknowledgments.

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

Parameters

- `max_wait`: Maximum blocking time for this operation

ReturnCode_t **get_offered_deadline_missed_status**(fastrtps::OfferedDeadlineMissedStatus *&status*)

Returns the offered deadline missed status.

Return RETCODE_OK

Parameters

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

ReturnCode_t **get_offered_incompatible_qos_status**(OfferedIncompatibleQosStatus *&status*)

Returns the offered incompatible qos status.

Return RETCODE_OK

Parameters

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

ReturnCode_t **set_qos**(**const** *DataWriterQos* *&qos*)

Establishes the *DataWriterQos* for this *DataWriter*.

Return RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *DataWriterQos* to be set

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

Return Reference to the current *DataWriterQos*

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

Return RETCODE_OK

Parameters

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

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

Return Pointer to the associated *Topic*

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

Return Pointer to the *DataWriterListener*

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

Return RETCODE_OK

Parameters

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

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

Return RETCODE_OK

Parameters

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

ReturnCode_t **dispose**(void **data*, **const** fastrtps::rtps::InstanceHandle_t &*handle*)
This operation requests the middleware to delete the data (the actual deletion is postponed until there is no more use for that data in the whole system). In general, applications are made aware of the deletion by means of operations on the *DataReader* objects that already knew that instance. This operation does not modify the value of the instance. The instance parameter is passed just for the purposes of identifying the instance. When this operation is used, the Service will automatically supply the value of the source_timestamp that is made available to *DataReader* objects by means of the source_timestamp attribute inside the *SampleInfo*. The constraints on the values of the handle parameter and the corresponding error behavior are the same specified for the unregister_instance operation.

Return RETCODE_PRECONDITION_NOT_MET if the handle introduced does not match with the one associated to the data, RETCODE_OK if the data is correctly sent and RETCODE_ERROR otherwise.

Parameters

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

ReturnCode_t **`get_liveliness_lost_status`**(LivelinessLostStatus &*status*)
Returns the liveliness lost status.

Return RETCODE_OK

Parameters

- `status`: Liveliness lost status struct

`const` *Publisher* ***`get_publisher`**`()` **`const`**
Getter for the *Publisher* that creates this *DataWriter*.

Return Pointer to the *Publisher*

ReturnCode_t **`assert_liveliness`**`()`
This operation manually asserts the liveliness of the *DataWriter*. This is used in combination with the *LivelinessQosPolicy* to indicate to the Service that the entity remains active. This operation need only be used if the LIVELINESS setting is either MANUAL_BY_PARTICIPANT or MANUAL_BY_TOPIC. Otherwise, it has no effect.

Note Writing data via the write operation on a *DataWriter* asserts liveliness on the *DataWriter* itself and its *DomainParticipant*. Consequently the use of assert_liveliness is only needed if the application is not writing data regularly.

Return RETCODE_OK if asserted, RETCODE_ERROR otherwise

ReturnCode_t **`clear_history`**(size_t **removed*)
Clears the *DataWriter* history.

Return RETCODE_OK if the samples are removed and RETCODE_ERROR otherwise

Parameters

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

DataWriterListener

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

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

Public Functions

DataWriterListener()
Constructor.

~DataWriterListener()
Destructor.

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

Parameters

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

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

Parameters

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

void **on_offered_incompatible_qos**(*DataWriter* *writer, **const** OfferedIncompatibleQosStatus &status)
A method called when an incompatible QoS is offered

Parameters

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

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

Parameters

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

DataWriterQos

class eprosima::fastdds::dds::**DataWriterQos**
Class *DataWriterQos*, containing all the possible Qos that can be set for a determined *DataWriter*. Although these values can be and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

DataWriterQos()
Constructor.

~DataWriterQos() = default
Destructor.

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

Return *DurabilityQosPolicy* reference

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

Return *DurabilityQosPolicy* reference

void **durability**(**const** *DurabilityQosPolicy* &*durability*)
Setter for *DurabilityQosPolicy*

Parameters

- `durability`: new value for the *DurabilityQosPolicy*

DurabilityServiceQosPolicy &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* reference

const *DurabilityServiceQosPolicy* &**durability_service**() **const**
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* reference

void **durability_service**(**const** *DurabilityServiceQosPolicy* &*durability_service*)
Setter for *DurabilityServiceQosPolicy*

Parameters

- `durability_service`: new value for the *DurabilityServiceQosPolicy*

DeadlineQosPolicy &**deadline**()
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* reference

const *DeadlineQosPolicy* &**deadline**() **const**
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* reference

void **deadline**(**const** *DeadlineQosPolicy* &*deadline*)
Setter for *DeadlineQosPolicy*

Parameters

- `deadline`: new value for the *DeadlineQosPolicy*

LatencyBudgetQosPolicy &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* reference

const *LatencyBudgetQosPolicy* &**latency_budget**() **const**
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* reference

void **latency_budget**(**const** *LatencyBudgetQosPolicy* &*latency_budget*)
Setter for *LatencyBudgetQosPolicy*

Parameters

- `latency_budget`: new value for the *LatencyBudgetQosPolicy*

LivelinessQosPolicy &**liveliness**()
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* reference

const *LivelinessQosPolicy* &**liveliness**() **const**
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* reference

void **liveliness**(**const** *LivelinessQosPolicy* &*liveliness*)
Setter for *LivelinessQosPolicy*

Parameters

- `liveliness`: new value for the *LivelinessQosPolicy*

ReliabilityQosPolicy &**reliability**()
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* reference

const *ReliabilityQosPolicy* &**reliability**() **const**
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* reference

void **reliability**(**const** *ReliabilityQosPolicy* &*reliability*)
Setter for *ReliabilityQosPolicy*

Parameters

- `reliability`: new value for the *ReliabilityQosPolicy*

DestinationOrderQosPolicy &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* reference

const *DestinationOrderQosPolicy* &**destination_order**() **const**
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* reference

void **destination_order**(**const** *DestinationOrderQosPolicy* &*destination_order*)
Setter for *DestinationOrderQosPolicy*

Parameters

- `destination_order`: new value for the *DestinationOrderQosPolicy*

HistoryQosPolicy &**history**()
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* reference

const *HistoryQosPolicy* &**history**() **const**
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* reference

void **history**(**const** *HistoryQosPolicy* &*history*)
Setter for *HistoryQosPolicy*

Parameters

- `history`: new value for the *HistoryQosPolicy*

ResourceLimitsQosPolicy &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* reference

const *ResourceLimitsQosPolicy* &**resource_limits**() **const**
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* reference

void **resource_limits**(**const** *ResourceLimitsQosPolicy* &*resource_limits*)
Setter for *ResourceLimitsQosPolicy*

Parameters

- `resource_limits`: new value for the *ResourceLimitsQosPolicy*

TransportPriorityQosPolicy &**transport_priority**()
Getter for *TransportPriorityQosPolicy*

Return *TransportPriorityQosPolicy* reference

const *TransportPriorityQosPolicy* &**transport_priority**() **const**
Getter for *TransportPriorityQosPolicy*

Return *TransportPriorityQosPolicy* reference

void **transport_priority**(**const** *TransportPriorityQosPolicy* &*transport_priority*)
Setter for *TransportPriorityQosPolicy*

Parameters

- `transport_priority`: new value for the *TransportPriorityQosPolicy*

LifespanQosPolicy &**lifespan**()
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* reference

const *LifespanQosPolicy* &**lifespan**() **const**
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* reference

void **lifespan**(**const** *LifespanQosPolicy* &*lifespan*)
Setter for *LifespanQosPolicy*

Parameters

- `lifespan`: new value for the *LifespanQosPolicy*

UserDataQosPolicy &**user_data**()
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

const *UserDataQosPolicy* &**user_data**() **const**
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

void **user_data**(**const** *UserDataQosPolicy* &*user_data*)
Setter for *UserDataQosPolicy*

Parameters

- `user_data`: new value for the *UserDataQosPolicy*

OwnershipQosPolicy &**ownership**()
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* reference

const *OwnershipQosPolicy* &**ownership**() **const**
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* reference

void **ownership**(**const** *OwnershipQosPolicy* &*ownership*)
Setter for *OwnershipQosPolicy*

Parameters

- `ownership`: new value for the *OwnershipQosPolicy*

OwnershipStrengthQosPolicy &**ownership_strength**()
Getter for *OwnershipStrengthQosPolicy*

Return *OwnershipStrengthQosPolicy* reference

const *OwnershipStrengthQosPolicy* &**ownership_strength**() **const**
Getter for *OwnershipStrengthQosPolicy*

Return *OwnershipStrengthQosPolicy* reference

void **ownership_strength**(**const** *OwnershipStrengthQosPolicy* &*ownership_strength*)
Setter for *OwnershipStrengthQosPolicy*

Parameters

- `ownership_strength`: new value for the *OwnershipStrengthQosPolicy*

WriterDataLifecycleQosPolicy &**writer_data_lifecycle**()
Getter for *WriterDataLifecycleQosPolicy*

Return *WriterDataLifecycleQosPolicy* reference

const *WriterDataLifecycleQosPolicy* &**writer_data_lifecycle**() **const**
Getter for *WriterDataLifecycleQosPolicy*

Return *WriterDataLifecycleQosPolicy* reference

void **writer_data_lifecycle**(**const** *WriterDataLifecycleQosPolicy* &*writer_data_lifecycle*)
Setter for *WriterDataLifecycleQosPolicy*

Parameters

- `writer_data_lifecycle`: new value for the *WriterDataLifecycleQosPolicy*

PublishModeQosPolicy &**publish_mode**()
Getter for *PublishModeQosPolicy*

Return *PublishModeQosPolicy* reference

const *PublishModeQosPolicy* &**publish_mode**() **const**
Getter for *PublishModeQosPolicy*

Return *PublishModeQosPolicy* reference

void **publish_mode**(**const** *PublishModeQosPolicy* &*publish_mode*)
Setter for *PublishModeQosPolicy*

Parameters

- `publish_mode`: new value for the *PublishModeQosPolicy*

DataRepresentationQosPolicy &**representation**()
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

const *DataRepresentationQosPolicy* &**representation**() **const**
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

void **representation**(**const** *DataRepresentationQosPolicy* &*representation*)
Setter for *DataRepresentationQosPolicy*

Parameters

- `representation`: new value for the *DataRepresentationQosPolicy*

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

Return PropertyPolicyQos reference

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

Return PropertyPolicyQos reference

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

Parameters

- `properties`: new value for the PropertyPolicyQos

RTPSReliableWriterQos &**reliable_writer_qos**()
Getter for *RTPSReliableWriterQos*

Return *RTPSReliableWriterQos* reference

const *RTPSReliableWriterQos* &**reliable_writer_qos**() **const**
Getter for *RTPSReliableWriterQos*

Return *RTPSReliableWriterQos* reference

void **reliable_writer_qos**(**const** *RTPSReliableWriterQos* &*reliable_writer_qos*)
Setter for *RTPSReliableWriterQos*

Parameters

- `reliable_writer_qos`: new value for the *RTPSReliableWriterQos*

RTPSEndpointQos &**endpoint**()
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* reference

const *RTPSEndpointQos* &**endpoint**() **const**
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* reference

void **endpoint**(**const** *RTPSEndpointQos* &*endpoint*)
Setter for *RTPSEndpointQos*

Parameters

- `endpoint`: new value for the *RTPSEndpointQos*

WriterResourceLimitsQos &**writer_resource_limits**()
Getter for *WriterResourceLimitsQos*

Return *WriterResourceLimitsQos* reference

const *WriterResourceLimitsQos* &**writer_resource_limits**() **const**
Getter for *WriterResourceLimitsQos*

Return *WriterResourceLimitsQos* reference

void **writer_resource_limits**(**const** *WriterResourceLimitsQos* &*writer_resource_limits*)
Setter for *WriterResourceLimitsQos*

Parameters

- `writer_resource_limits`: new value for the *WriterResourceLimitsQos*

fastrtps::rtps::ThroughputControllerDescriptor &**throughput_controller**()
Getter for ThroughputControllerDescriptor

Return ThroughputControllerDescriptor reference

const fastrtps::rtps::ThroughputControllerDescriptor &**throughput_controller**() **const**
Getter for ThroughputControllerDescriptor

Return ThroughputControllerDescriptor reference

void **throughput_controller**(**const** fastrtps::rtps::ThroughputControllerDescriptor &*throughput_controller*)
Setter for ThroughputControllerDescriptor

Parameters

- `throughput_controller`: new value for the ThroughputControllerDescriptor

const DataWriterQos `dds::fastdds::eprosima::`**DATAWRITER_QOS_DEFAULT**

Publisher

class `eprosima::fastdds::dds::`**Publisher** : **public** eprosima::fastdds::dds::*DomainEntity*
Class *Publisher*, used to send data to associated subscribers.

Public Functions

~Publisher()
Destructor.

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

Return RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the participant creating this *Publisher* is not enabled.

const *PublisherQos* &**get_qos**() **const**
Allows accessing the *Publisher* Qos.

Return *PublisherQos* reference

ReturnCode_t **get_qos**(*PublisherQos* &*qos*) **const**
Retrieves the *Publisher* Qos.

Return RETCODE_OK

ReturnCode_t **set_qos**(**const** *PublisherQos* &*qos*)
Allows modifying the *Publisher* Qos. The given Qos must be supported by the *PublisherQos*.

Return RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: *PublisherQos* to be set

const *PublisherListener* ***get_listener**() **const**
Retrieves the attached *PublisherListener*.

Return *PublisherListener* pointer

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

Return RETCODE_OK

Parameters

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

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

Return RETCODE_OK

Parameters

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

DataWriter ***create_datawriter**(*Topic* **topic*, **const** *DataWriterQos* &*qos*, *DataWriterListener* **listener* = nullptr, **const** *StatusMask* &*mask* = *StatusMask*::*all*())
This operation creates a *DataWriter*. The returned *DataWriter* will be attached and belongs to the *Publisher*.

Return Pointer to the created *DataWriter*. nullptr if failed.

Parameters

- `topic`: *Topic* the *DataWriter* will be listening
- `qos`: QoS of the *DataWriter*.
- `listener`: Pointer to the listener (default: nullptr).
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all).

DataWriter ***create_datawriter_with_profile***(*Topic* **topic*, **const** std::string *&profile_name*, *DataWriterListener* **listener* = nullptr, **const** *StatusMask* *&mask* = *StatusMask*::*all*())

This operation creates a *DataWriter*. The returned *DataWriter* will be attached and belongs to the *Publisher*.

Return Pointer to the created *DataWriter*. nullptr if failed.

Parameters

- `topic`: *Topic* the *DataWriter* will be listening
- `profile_name`: *DataWriter* profile name.
- `listener`: Pointer to the listener (default: nullptr).
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all).

ReturnCode_t ***delete_datawriter***(*DataWriter* **writer*)

This operation deletes a *DataWriter* that belongs to the *Publisher*.

The delete_datawriter operation must be called on the same *Publisher* object used to create the *DataWriter*. If delete_datawriter is called on a different *Publisher*, the operation will have no effect and it will return false.

The deletion of the *DataWriter* will automatically unregister all instances. Depending on the settings of the WRITER_DATA_LIFECYCLE *QosPolicy*, the deletion of the *DataWriter* may also dispose all instances.

Return RETCODE_PRECONDITION_NOT_MET if it does not belong to this *Publisher*, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Parameters

- `writer`: *DataWriter* to delete

DataWriter ***lookup_datawriter***(**const** std::string *&topic_name*) **const**

This operation retrieves a previously created *DataWriter* belonging to the *Publisher* that is attached to a *Topic* with a matching topic_name. If no such *DataWriter* exists, the operation will return nullptr.

If multiple *DataWriter* attached to the *Publisher* satisfy this condition, then the operation will return one of them. It is not specified which one.

Return Pointer to a previously created *DataWriter* associated to a *Topic* with the requested topic_name

Parameters

- `topic_name`: Name of the *Topic*

bool ***get_datawriters***(std::vector<*DataWriter**> *&writers*) **const**

Fills the given vector with all the datawriters of this publisher.

Return true

Parameters

- `writers`: Vector where the DataWriters are returned

bool ***has_datawriters***() **const**

This operation checks if the publisher has DataWriters

Return true if the publisher has one or several DataWriters, false otherwise

ReturnCode_t ***wait_for_acknowledgments***(**const** fastrtps::Duration_t *&max_wait*)

This operation blocks the calling thread until either all data written by the reliable *DataWriter* entities is acknowledged by all matched reliable *DataReader* entities, or else the duration specified by the max_wait

parameter elapses, whichever happens first. A return value of true indicates that all the samples written have been acknowledged by all reliable matched data readers; a return value of false indicates that max_wait elapsed before all the data was acknowledged.

Return RETCODE_TIMEOUT if the function takes more than the maximum blocking time established, RETCODE_OK if the *Publisher* receives the acknowledgments and RETCODE_ERROR otherwise.

Parameters

- `max_wait`: Maximum blocking time for this operation

const *DomainParticipant* ***get_participant**() **const**

This operation returns the *DomainParticipant* to which the *Publisher* belongs.

Return Pointer to the *DomainParticipant*

ReturnCode_t **set_default_datawriter_qos**(**const** *DataWriterQos* &*qos*)

This operation sets a default value of the *DataWriter* QoS policies which will be used for newly created *DataWriter* entities in the case where the QoS policies are defaulted in the create_datawriter operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value DATAWRITER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_datawriter_qos operation had never been called.

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

Parameters

- `qos`: *DataWriterQos* to be set

const *DataWriterQos* &**get_default_datawriter_qos**() **const**

This operation returns the default value of the *DataWriter* QoS, that is, the QoS policies which will be used for newly created *DataWriter* entities in the case where the QoS policies are defaulted in the create_datawriter operation.

The values retrieved by get_default_datawriter_qos will match the set of values specified on the last successful call to set_default_datawriter_qos, or else, if the call was never made, the default values.

Return Current default WriterQos

ReturnCode_t **get_default_datawriter_qos**(*DataWriterQos* &*qos*) **const**

This operation retrieves the default value of the *DataWriter* QoS, that is, the QoS policies which will be used for newly created *DataWriter* entities in the case where the QoS policies are defaulted in the create_datawriter operation.

The values retrieved by get_default_datawriter_qos will match the set of values specified on the last successful call to set_default_datawriter_qos, or else, if the call was never made, the default values.

Return RETCODE_OK

Parameters

- `qos`: Reference to the current default WriterQos.

ReturnCode_t **get_datawriter_qos_from_profile**(**const** std::string &*profile_name*, *DataWriterQos* &*qos*) **const**

Fills the *DataWriterQos* with the values of the XML profile.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *DataWriter* profile name.
- `qos`: *DataWriterQos* object where the qos is returned.

const fastrtps::rtps::InstanceHandle_t &**get_instance_handle**() **const**
Returns the *Publisher*'s handle.

Return InstanceHandle of this *Publisher*.

PublisherListener

class eprosima::fastdds::dds::**PublisherListener** : **public** eprosima::fastdds::dds::*DataWriterListener*
Class *PublisherListener*, allows the end user to implement callbacks triggered by certain events. It inherits all the *DataWriterListener* callbacks.

Subclassed by *eprosima::fastdds::dds::DomainParticipantListener*

Public Functions

PublisherListener()
Constructor.

~PublisherListener()
Destructor.

PublisherQos

class eprosima::fastdds::dds::**PublisherQos**
Class *PublisherQos*, containing all the possible Qos that can be set for a determined *Publisher*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

PublisherQos()
Constructor.

~PublisherQos() = default
Destructor.

const *PresentationQosPolicy* &**presentation**() **const**
Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

PresentationQosPolicy &**presentation**()
Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

void **presentation**(**const** *PresentationQosPolicy* &*presentation*)
Setter for *PresentationQosPolicy*

Parameters

- `presentation`: *PresentationQosPolicy*

const *PartitionQosPolicy* &**partition**() **const**
Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

PartitionQosPolicy &**partition**()
Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

void **partition**(**const** *PartitionQosPolicy* &*partition*)
Setter for *PartitionQosPolicy*

Parameters

- partition: *PartitionQosPolicy*

const *GroupDataQosPolicy* &**group_data**() **const**
Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

GroupDataQosPolicy &**group_data**()
Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

void **group_data**(**const** *GroupDataQosPolicy* &*group_data*)
Setter for *GroupDataQosPolicy*

Parameters

- group_data: *GroupDataQosPolicy*

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

Return *EntityFactoryQosPolicy* reference

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

Return *EntityFactoryQosPolicy* reference

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

Parameters

- entity_factory: *EntityFactoryQosPolicy*

const PublisherQos dds::fastdds::eprosima::**PUBLISHER_QOS_DEFAULT**

RTPSReliableWriterQos

class eprosima::fastdds::dds::**RTPSReliableWriterQos**
Qos Policy to configure the DisablePositiveACKsQos and the writer timing attributes.

Public Functions

RTPSReliableWriterQos()
Constructor.

~RTPSReliableWriterQos() = default
Destructor.

Public Members

fastrtps::rtps::WriterTimes **times**
Writer Timing Attributes.

DisablePositiveACKsQosPolicy **disable_positive_acks**
Disable positive acks QoS, implemented in the library.

Subscriber

DataReader

class eprosima::fastdds::dds::**DataReader** : **public** eprosima::fastdds::dds::*DomainEntity*
Class *DataReader*, contains the actual implementation of the behaviour of the *Subscriber*.

Read or take data methods.

Methods to read or take data from the History.

ReturnCode_t **read_next_sample**(void **data*, *SampleInfo* **info*)
This operation copies the next, non-previously accessed Data value from the *DataReader*; the operation also copies the corresponding *SampleInfo*. The implied order among the samples stored in the *DataReader* is the same as for the read operation.

The read_next_sample operation is semantically equivalent to the read operation where the input Data sequence has max_length=1, the sample_states=NOT_READ, the view_states=ANY_VIEW_STATE, and the instance_states=ANY_INSTANCE_STATE.

The read_next_sample operation provides a simplified API to 'read' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the *DataReader*, the operation will return NO_DATA and nothing is copied

Return RETCODE_NO_DATA if the history is empty, RETCODE_OK if the next sample is returned and RETCODE_ERROR otherwise

Parameters

- `data`: Data pointer to store the sample
- `info`: *SampleInfo* pointer to store the sample information

ReturnCode_t **take_next_sample**(void **data*, *SampleInfo* **info*)
This operation copies the next, non-previously accessed Data value from the *DataReader* and 'removes' it from the *DataReader* so it is no longer accessible. The operation also copies the corresponding *SampleInfo*. This operation is analogous to the read_next_sample except for the fact that the sample is 'removed' from the *DataReader*.

The take_next_sample operation is semantically equivalent to the take operation where the input sequence has max_length=1, the sample_states=NOT_READ, the view_states=ANY_VIEW_STATE, and the instance_states=ANY_INSTANCE_STATE.

This operation provides a simplified API to 'take' samples avoiding the need for the application to manage sequences and specify states.

If there is no unread data in the *DataReader*, the operation will return NO_DATA and nothing is copied.

Return RETCODE_NO_DATA if the history is empty, RETCODE_OK if the next sample is returned and RETCODE_ERROR otherwise

Parameters

- `data`: Data pointer to store the sample
- `info`: *SampleInfo* pointer to store the sample information

Public Functions

~DataReader()
Destructor.

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

Return RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the *Subscriber* creating this *DataReader* is not enabled.

bool **wait_for_unread_message**(**const** fastrtps::Duration_t &*timeout*)
Method to block the current thread until an unread message is available

Return true if there is new unread message, false if timeout

Parameters

- `timeout`: Max blocking time for this operation

ReturnCode_t **get_first_untaken_info**(*SampleInfo* **info*)
Returns information about the first untaken sample.

Return RETCODE_OK if sample info was returned. RETCODE_NO_DATA if there is no sample to take.

Parameters

- `[out] info`: Pointer to a SampleInfo_t structure to store first untaken sample information.

const fastrtps::rtps::GUID_t &**guid**()
Get associated GUID

Return Associated GUID

fastrtps::rtps::InstanceHandle_t **get_instance_handle**() **const**
Getter for the associated InstanceHandle.

Return Copy of the InstanceHandle

TypeSupport **type**()
Getter for the data type

Return *TypeSupport* associated to the *DataReader*

const *TopicDescription* ***get_topicdescription**() **const**
Get *TopicDescription*

Return *TopicDescription* pointer

ReturnCode_t **get_requested_deadline_missed_status**(fastrtps::RequestedDeadlineMissedStatus *&status*)
Get the requested deadline missed status.

Return The deadline missed status

ReturnCode_t **get_requested_incompatible_qos_status**(RequestedIncompatibleQosStatus *&status*)
Get the requested incompatible qos status.

Return RETCODE_OK

Parameters

- `[out] status`: Requested incompatible qos status

ReturnCode_t **set_qos**(**const** *DataReaderQos* *&qos*)
Setter for the *DataReaderQos*.

Return RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- `qos`: new value for the *DataReaderQos*

const *DataReaderQos* &**get_qos**() **const**
Getter for the *DataReaderQos*.

Return Pointer to the *DataReaderQos*

ReturnCode_t **get_qos**(*DataReaderQos* *&qos*) **const**
Getter for the *DataReaderQos*.

Return RETCODE_OK

Parameters

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

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

Return RETCODE_OK

Parameters

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

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

Return RETCODE_OK

Parameters

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

const *DataReaderListener* ***get_listener**() **const**

Getter for the *DataReaderListener*.

Return Pointer to the *DataReaderListener*

ReturnCode_t **get_liveliness_changed_status**(*LivelinessChangedStatus* &*status*) **const**

Get the liveliness changed status.

Return RETCODE_OK

Parameters

- `status`: *LivelinessChangedStatus* object where the status is returned

const *Subscriber* ***get_subscriber**() **const**

Getter for the *Subscriber*.

Return *Subscriber* pointer

DataReaderListener

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

Class *DataReaderListener*, it should be used by the end user to implement specific callbacks to certain actions.

Subclassed by *eprosima::fastdds::dds::SubscriberListener*

Public Functions

DataReaderListener()

Constructor.

~DataReaderListener()

Destructor.

void **on_data_available**(*DataReader* **reader*)

Virtual function to be implemented by the user containing the actions to be performed when a new Data Message is received.

Parameters

- `reader`: *DataReader*

void **on_subscription_matched**(*DataReader* **reader*, **const** fastdds::dds::*SubscriptionMatchedStatus* &*info*)

Virtual method to be called when the subscriber is matched with a new Writer (or unmatched); i.e., when a writer publishing in the same topic is discovered.

Parameters

- `reader`: *DataReader*
- `info`: The subscription matched status

void **on_requested_deadline_missed**(*DataReader* *reader, **const** fastrtps::RequestedDeadlineMissedStatus &status)

Virtual method to be called when a topic misses the deadline period

Parameters

- `reader`: *DataReader*
- `status`: The requested deadline missed status

void **on_liveliness_changed**(*DataReader* *reader, **const** fastrtps::LivelinessChangedStatus &status)

Method called when the liveliness status associated to a subscriber changes.

Parameters

- `reader`: The *DataReader*
- `status`: The liveliness changed status

void **on_sample_rejected**(*DataReader* *reader, **const** fastrtps::SampleRejectedStatus &status)

Method called when a sample was rejected.

Parameters

- `reader`: The *DataReader*
- `status`: The rejected status

void **on_requested_incompatible_qos**(*DataReader* *reader, **const** RequestedIncompatibleQosStatus &status)

Method called an incompatible QoS was requested.

Parameters

- `reader`: The *DataReader*
- `status`: The requested incompatible QoS status

void **on_sample_lost**(*DataReader* *reader, **const** SampleLostStatus &status)

Method called when a sample was lost.

Parameters

- `reader`: The *DataReader*
- `status`: The sample lost status

DataReaderQos

class `eprosima::fastdds::dds::`**DataReaderQos**

Class *DataReaderQos*, containing all the possible Qos that can be set for a determined *DataReader*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

DataReaderQos()
Constructor.

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

Return *DurabilityQosPolicy* reference

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

Return *DurabilityQosPolicy* const reference

void **durability**(**const** *DurabilityQosPolicy* &*new_value*)
Setter for *DurabilityQosPolicy*

Parameters

- `new_value`: new value for the *DurabilityQosPolicy*

DeadlineQosPolicy &**deadline**()
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* reference

const *DeadlineQosPolicy* &**deadline**() **const**
Getter for *DeadlineQosPolicy*

Return *DeadlineQosPolicy* const reference

void **deadline**(**const** *DeadlineQosPolicy* &*new_value*)
Setter for *DeadlineQosPolicy*

Parameters

- `new_value`: new value for the *DeadlineQosPolicy*

LatencyBudgetQosPolicy &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* reference

const *LatencyBudgetQosPolicy* &**latency_budget**() **const**
Getter for *LatencyBudgetQosPolicy*

Return *LatencyBudgetQosPolicy* const reference

void **latency_budget**(**const** *LatencyBudgetQosPolicy* &*new_value*)
Setter for *LatencyBudgetQosPolicy*

Parameters

- `new_value`: new value for the *LatencyBudgetQosPolicy*

LivelinessQosPolicy &**liveliness**()
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* reference

const *LivelinessQosPolicy* &**liveliness**() **const**
Getter for *LivelinessQosPolicy*

Return *LivelinessQosPolicy* const reference

void **liveliness**(**const** *LivelinessQosPolicy* &*new_value*)
Setter for *LivelinessQosPolicy*

Parameters

- `new_value`: new value for the *LivelinessQosPolicy*

ReliabilityQosPolicy &**reliability**()
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* reference

const *ReliabilityQosPolicy* &**reliability**() **const**
Getter for *ReliabilityQosPolicy*

Return *ReliabilityQosPolicy* const reference

void **reliability**(**const** *ReliabilityQosPolicy* &*new_value*)
Setter for *ReliabilityQosPolicy*

Parameters

- `new_value`: new value for the *ReliabilityQosPolicy*

DestinationOrderQosPolicy &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* reference

const *DestinationOrderQosPolicy* &**destination_order**() **const**
Getter for *DestinationOrderQosPolicy*

Return *DestinationOrderQosPolicy* const reference

void **destination_order**(**const** *DestinationOrderQosPolicy* &*new_value*)
Setter for *DestinationOrderQosPolicy*

Parameters

- `new_value`: new value for the *DestinationOrderQosPolicy*

HistoryQosPolicy &**history**()
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* reference

const *HistoryQosPolicy* &**history**() **const**
Getter for *HistoryQosPolicy*

Return *HistoryQosPolicy* const reference

void **history**(**const** *HistoryQosPolicy* &*new_value*)
Setter for *HistoryQosPolicy*

Parameters

- `new_value`: new value for the *HistoryQosPolicy*

ResourceLimitsQosPolicy &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* reference

const *ResourceLimitsQosPolicy* &**resource_limits**() **const**
Getter for *ResourceLimitsQosPolicy*

Return *ResourceLimitsQosPolicy* const reference

void **resource_limits**(**const** *ResourceLimitsQosPolicy* &*new_value*)
Setter for *ResourceLimitsQosPolicy*

Parameters

- `new_value`: new value for the *ResourceLimitsQosPolicy*

UserDataQosPolicy &**user_data**()
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* reference

const *UserDataQosPolicy* &**user_data**() **const**
Getter for *UserDataQosPolicy*

Return *UserDataQosPolicy* const reference

void **user_data**(**const** *UserDataQosPolicy* &*new_value*)
Setter for *UserDataQosPolicy*

Parameters

- `new_value`: new value for the *UserDataQosPolicy*

OwnershipQosPolicy &**ownership**()
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* reference

const *OwnershipQosPolicy* &**ownership**() **const**
Getter for *OwnershipQosPolicy*

Return *OwnershipQosPolicy* const reference

void **ownership**(**const** *OwnershipQosPolicy* &*new_value*)
Setter for *OwnershipQosPolicy*

Parameters

- `new_value`: new value for the *OwnershipQosPolicy*

TimeBasedFilterQosPolicy &**time_based_filter**()
Getter for *TimeBasedFilterQosPolicy*

Return *TimeBasedFilterQosPolicy* reference

const *TimeBasedFilterQosPolicy* &**time_based_filter**() **const**
Getter for *TimeBasedFilterQosPolicy*

Return *TimeBasedFilterQosPolicy* const reference

void **time_based_filter**(**const** *TimeBasedFilterQosPolicy* &*new_value*)
Setter for *TimeBasedFilterQosPolicy*

Parameters

- `new_value`: new value for the *TimeBasedFilterQosPolicy*

ReaderDataLifecycleQosPolicy &**reader_data_lifecycle**()
Getter for *ReaderDataLifecycleQosPolicy*

Return *ReaderDataLifecycleQosPolicy* reference

const *ReaderDataLifecycleQosPolicy* &**reader_data_lifecycle**() **const**
Getter for *ReaderDataLifecycleQosPolicy*

Return *ReaderDataLifecycleQosPolicy* const reference

void **reader_data_lifecycle**(**const** *ReaderDataLifecycleQosPolicy* &*new_value*)
Setter for *ReaderDataLifecycleQosPolicy*

Parameters

- `new_value`: new value for the *ReaderDataLifecycleQosPolicy*

LifespanQosPolicy &**lifespan**()
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* reference

const *LifespanQosPolicy* &**lifespan**() **const**
Getter for *LifespanQosPolicy*

Return *LifespanQosPolicy* const reference

void **lifespan**(**const** *LifespanQosPolicy* &*new_value*)
Setter for *LifespanQosPolicy*

Parameters

- `new_value`: new value for the *LifespanQosPolicy*

DurabilityServiceQosPolicy &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* reference

const *DurabilityServiceQosPolicy* &**durability_service**() **const**
Getter for *DurabilityServiceQosPolicy*

Return *DurabilityServiceQosPolicy* const reference

void **durability_service**(**const** *DurabilityServiceQosPolicy* &*new_value*)
Setter for *DurabilityServiceQosPolicy*

Parameters

- `new_value`: new value for the *DurabilityServiceQosPolicy*

RTPSReliableReaderQos &**reliable_reader_qos**()
Getter for *RTPSReliableReaderQos*

Return *RTPSReliableReaderQos* reference

const *RTPSReliableReaderQos* &**reliable_reader_qos**() **const**
Getter for *RTPSReliableReaderQos*

Return *RTPSReliableReaderQos* const reference

void **reliable_reader_qos**(**const** *RTPSReliableReaderQos* &*new_value*)
Setter for *RTPSReliableReaderQos*

Parameters

- `new_value`: new value for the *RTPSReliableReaderQos*

TypeConsistencyQos &**type_consistency**()
Getter for *TypeConsistencyQos*

Return *TypeConsistencyQos* reference

const *TypeConsistencyQos* &**type_consistency**() **const**
Getter for *TypeConsistencyQos*

Return *TypeConsistencyQos* const reference

void **type_consistency**(**const** *TypeConsistencyQos* &*new_value*)
Setter for *TypeConsistencyQos*

Parameters

- `new_value`: new value for the *TypeConsistencyQos*

bool **expects_inline_qos**() **const**
Getter for expectsInlineQos_

Return expectsInlineQos_

void **expects_inline_qos**(bool *new_value*)
Setter for expectsInlineQos_

Parameters

- `new_value`: new value for the expectsInlineQos_

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

Return PropertyPolicyQos reference

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

Return PropertyPolicyQos const reference

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

Parameters

- `new_value`: new value for the PropertyPolicyQos

RTPSEndpointQos &**endpoint**()
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* reference

const *RTPSEndpointQos* &**endpoint**() **const**
Getter for *RTPSEndpointQos*

Return *RTPSEndpointQos* const reference

void **endpoint**(**const** *RTPSEndpointQos* &*new_value*)
Setter for *RTPSEndpointQos*

Parameters

- `new_value`: new value for the *RTPSEndpointQos*

ReaderResourceLimitsQos &**reader_resource_limits**()
Getter for *ReaderResourceLimitsQos*

Return *ReaderResourceLimitsQos* reference

const *ReaderResourceLimitsQos* &**reader_resource_limits**() **const**
Getter for *ReaderResourceLimitsQos*

Return *ReaderResourceLimitsQos* const reference

void **reader_resource_limits**(**const** *ReaderResourceLimitsQos* &*new_value*)
Setter for *ReaderResourceLimitsQos*

Parameters

- `new_value`: new value for the *ReaderResourceLimitsQos*

const DataReaderQos `dds::fastdds::eprosima::`**DATAREADER_QOS_DEFAULT**

InstanceStateKind

enum `dds::fastdds::eprosima::`**InstanceStateKind**

Values:

enumerator ALIVE

enumerator NOT_ALIVE_DISPOSED

enumerator NOT_ALIVE_NO_WRITERS

ReaderResourceLimitsQos

class `eprosima::fastdds::dds::`**ReaderResourceLimitsQos**

Qos Policy to configure the limit of the reader resources.

Public Functions

ReaderResourceLimitsQos()

Constructor.

~ReaderResourceLimitsQos() = default

Destructor.

Public Members

fastrtps::ResourceLimitedContainerConfig **matched_publisher_allocation**

Matched publishers allocation limits.

RTPSReliableReaderQos

class `eprosima::fastdds::dds::`**RTPSReliableReaderQos**

Qos Policy to configure the DisablePositiveACKsQos and the reader attributes.

Public Functions

RTPSReliableReaderQos()

Constructor.

~RTPSReliableReaderQos() = default

Destructor.

Public Members

fastrtps::rtps::ReaderTimes **times**
Times associated with the Reliable Readers events.

DisablePositiveACKsQosPolicy **disable_positive_ACKs**
Control the sending of positive ACKs.

SampleInfo

struct `eprosima::fastdds::dds::`**SampleInfo**
SampleInfo is the information that accompanies each sample that is 'read' or 'taken.
,

Public Members

SampleStateKind **sample_state**
indicates whether or not the corresponding data sample has already been read

ViewStateKind **view_state**
indicates whether the *DataReader* has already seen samples for the most-current generation of the related instance.

InstanceStateKind **instance_state**
indicates whether the instance is currently in existence or, if it has been disposed, the reason why it was disposed.

int32_t **disposed_generation_count**
number of times the instance had become alive after it was disposed

int32_t **no_writers_generation_count**
number of times the instance had become alive after it was disposed because no writers

int32_t **sample_rank**
number of samples related to the same instance that follow in the collection

int32_t **generation_rank**
the generation difference between the time the sample was received, and the time the most recent sample in the collection was received.

int32_t **absoulte_generation_rank**
the generation difference between the time the sample was received, and the time the most recent sample was received. The most recent sample used for the calculation may or may not be in the returned collection

fastrtps::rtps::Time_t **source_timestamp**
time provided by the *DataWriter* when the sample was written

fastrtps::rtps::InstanceHandle_t **instance_handle**
identifies locally the corresponding instance

fastrtps::rtps::InstanceHandle_t **publication_handle**
identifies locally the *DataWriter* that modified the instance

Is the same InstanceHandle_t that is returned by the operation get_matched_publications on the *DataReader*

bool **valid_data**
whether the DataSample contains data or is only used to communicate of a change in the instance

fastrtps::rtps::SampleIdentity **sample_identity**
Sample Identity (Extension for RPC)

fastrtps::rtps::SampleIdentity **related_sample_identity**
Related Sample Identity (Extension for RPC)

SampleStateKind

enum dds::fastdds::eprosima::**SampleStateKind**
Values:

enumerator READ

enumerator NOT_READ

Subscriber

class eprosima::fastdds::dds::**Subscriber** : **public** eprosima::fastdds::dds::*DomainEntity*
Class *Subscriber*, contains the public API that allows the user to control the reception of messages. This class should not be instantiated directly. DomainRTPSParticipant class should be used to correctly create this element.

Public Functions

~Subscriber()
Destructor.

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

Return RETCODE_OK is successfully enabled. RETCODE_PRECONDITION_NOT_MET if the participant creating this *Subscriber* is not enabled.

const *SubscriberQos* &**get_qos**() **const**
Allows accessing the *Subscriber* Qos.

Return *SubscriberQos* reference

ReturnCode_t **get_qos**(*SubscriberQos* &*qos*) **const**
Retrieves the *Subscriber* Qos.

Return RETCODE_OK

Parameters

- qos: *SubscriberQos* where the qos is returned

ReturnCode_t **set_qos**(**const** *SubscriberQos* &*qos*)
Allows modifying the *Subscriber* Qos. The given Qos must be supported by the *SubscriberQos*.

Return RETCODE_IMMUTABLE_POLICY if any of the Qos cannot be changed, RETCODE_INCONSISTENT_POLICY if the Qos is not self consistent and RETCODE_OK if the qos is changed correctly.

Parameters

- qos: new value for *SubscriberQos*

const *SubscriberListener* ***get_listener**() **const**
Retrieves the attached *SubscriberListener*.

Return Pointer to the *SubscriberListener*

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

Return RETCODE_OK

Parameters

- `listener`: new value for *SubscriberListener*

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

Return RETCODE_OK

Parameters

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

DataReader ***create_datareader**(*TopicDescription* *topic, **const** *DataReaderQos* &reader_qos, *DataReaderListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())
This operation creates a *DataReader*. The returned *DataReader* will be attached and belong to the *Subscriber*.

Return Pointer to the created *DataReader*. nullptr if failed.

Parameters

- `topic`: *Topic* the *DataReader* will be listening.
- `reader_qos`: QoS of the *DataReader*.
- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all).

DataReader ***create_datareader_with_profile**(*TopicDescription* *topic, **const** std::string &profile_name, *DataReaderListener* *listener = nullptr, **const** *StatusMask* &mask = *StatusMask*::*all*())
This operation creates a *DataReader*. The returned *DataReader* will be attached and belongs to the *Subscriber*.

Return Pointer to the created *DataReader*. nullptr if failed.

Parameters

- `topic`: *Topic* the *DataReader* will be listening.
- `profile_name`: *DataReader* profile name.
- `listener`: Pointer to the listener (default: nullptr)
- `mask`: *StatusMask* that holds statuses the listener responds to (default: all).

ReturnCode_t **delete_datareader**(*DataReader* *reader)
This operation deletes a *DataReader* that belongs to the *Subscriber*.

The delete_datareader operation must be called on the same *Subscriber* object used to create the *DataReader*. If delete_datareader is called on a different *Subscriber*, the operation will have no effect and it will return an error.

Return RETCODE_PRECONDITION_NOT_MET if the datareader does not belong to this subscriber, RETCODE_OK if it is correctly deleted and RETCODE_ERROR otherwise.

Parameters

- `reader`: *DataReader* to delete

DataReader ***lookup_datareader**(**const** std::string &*topic_name*) **const**

This operation retrieves a previously-created *DataReader* belonging to the *Subscriber* that is attached to a *Topic* with a matching topic_name. If no such *DataReader* exists, the operation will return nullptr.

If multiple DataReaders attached to the *Subscriber* satisfy this condition, then the operation will return one of them. It is not specified which one.

Return Pointer to a previously created *DataReader* created on a *Topic* with that topic_name

Parameters

- `topic_name`: Name of the topic associated to the *DataReader*

ReturnCode_t **get_datareaders**(std::vector<*DataReader**> &*readers*) **const**

This operation allows the application to access the *DataReader* objects.

Return RETCODE_OK

Parameters

- `readers`: Vector of *DataReader* where the list of existing readers is returned

bool **has_datareaders**() **const**

This operation checks if the subscriber has DataReaders

Return true if the subscriber has one or several DataReaders, false in other case

ReturnCode_t **notify_datareaders**() **const**

This operation invokes the operation on_data_available on the *DataReaderListener* objects attached to contained *DataReader* entities.

This operation is typically invoked from the on_data_on_readers operation in the *SubscriberListener*. That way the *SubscriberListener* can delegate to the *DataReaderListener* objects the handling of the data.

Return RETCODE_OK

ReturnCode_t **set_default_datareader_qos**(**const** *DataReaderQos* &*qos*)

This operation sets a default value of the *DataReader* QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

This operation will check that the resulting policies are self consistent; if they are not, the operation will have no effect and return false.

The special value DATAREADER_QOS_DEFAULT may be passed to this operation to indicate that the default QoS should be reset back to the initial values the factory would use, that is the values that would be used if the set_default_datareader_qos operation had never been called.

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

Parameters

- `qos`: new value for *DataReaderQos* to set as default

const *DataReaderQos* &**get_default_datareader_qos**() **const**

This operation returns the default value of the *DataReader* QoS, that is, the QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Return Current default *DataReaderQos*.

DataReaderQos &**get_default_datareader_qos**()

This operation returns the default value of the *DataReader* QoS, that is, the QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Return Current default *DataReaderQos*.

ReturnCode_t **get_default_datareader_qos**(*DataReaderQos &qos*) **const**

This operation retrieves the default value of the *DataReader* QoS, that is, the QoS policies which will be used for newly created *DataReader* entities in the case where the QoS policies are defaulted in the create_datareader operation.

The values retrieved get_default_datareader_qos will match the set of values specified on the last successful call to get_default_datareader_qos, or else, if the call was never made, the default values.

Return RETCODE_OK

Parameters

- `qos`: *DataReaderQos* where the default_qos is returned

ReturnCode_t **get_datareader_qos_from_profile**(**const** std::string *&profile_name*, *DataReaderQos &qos*) **const**

Fills the *DataReaderQos* with the values of the XML profile.

Return RETCODE_OK if the profile exists. RETCODE_BAD_PARAMETER otherwise.

Parameters

- `profile_name`: *DataReader* profile name.
- `qos`: *DataReaderQos* object where the qos is returned.

const *DomainParticipant* ***get_participant**() **const**

This operation returns the *DomainParticipant* to which the *Subscriber* belongs.

Return *DomainParticipant* Pointer

const fastrtps::rtps::InstanceHandle_t &**get_instance_handle**() **const**

Returns the *Subscriber*'s handle.

Return InstanceHandle of this *Subscriber*.

SubscriberListener

class `eprosima::fastdds::dds::`**`SubscriberListener`** : **public** eprosima::fastdds::dds::*DataReaderListener*

Class *SubscriberListener*, it should be used by the end user to implement specific callbacks to certain actions. It also inherits all *DataReaderListener* callbacks.

Subclassed by *eprosima::fastdds::dds::DomainParticipantListener*

Public Functions

`SubscriberListener``()`

Constructor.

`~SubscriberListener``()`

Destructor.

void **`on_data_on_readers`**`(`*Subscriber* **sub*`)`

Virtual function to be implemented by the user containing the actions to be performed when a new Data Message is available on any reader.

Parameters

- `sub`: *Subscriber*

SubscriberQos

class `eprosima::fastdds::dds::`**`SubscriberQos`**

Class *SubscriberQos*, contains all the possible Qos that can be set for a determined *Subscriber*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

`SubscriberQos``()`

Constructor.

`~SubscriberQos``()`

Destructor.

const *PresentationQosPolicy* &**`presentation`**`()` **const**

Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

PresentationQosPolicy &**`presentation`**`()`

Getter for *PresentationQosPolicy*

Return *PresentationQosPolicy* reference

void **`presentation`**`(`**const** *PresentationQosPolicy* &*presentation*`)`

Setter for *PresentationQosPolicy*

Parameters

- `presentation`: new value for the *PresentationQosPolicy*

const *PartitionQosPolicy* &**partition**() **const**
Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

PartitionQosPolicy &**partition**()
Getter for *PartitionQosPolicy*

Return *PartitionQosPolicy* reference

void **partition**(**const** *PartitionQosPolicy* &*partition*)
Setter for *PartitionQosPolicy*

Parameters

- `partition`: new value for the *PartitionQosPolicy*

const *GroupDataQosPolicy* &**group_data**() **const**
Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

GroupDataQosPolicy &**group_data**()
Getter for *GroupDataQosPolicy*

Return *GroupDataQosPolicy* reference

void **group_data**(**const** *GroupDataQosPolicy* &*group_data*)
Setter for *GroupDataQosPolicy*

Parameters

- `group_data`: new value for the *GroupDataQosPolicy*

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

Return *EntityFactoryQosPolicy* reference

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

Return *EntityFactoryQosPolicy* reference

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

Parameters

- `entity_factory`: new value for the *EntityFactoryQosPolicy*

const SubscriberQos `dds::fastdds::eprosima::`**SUBSCRIBER_QOS_DEFAULT**

TypeConsistencyQos

class `eprosima::fastdds::dds::`**TypeConsistencyQos** : **public** eprosima::fastdds::dds::*QosPolicy*
Qos Policy to configure the XTypes Qos associated to the *DataReader*.

Public Functions

TypeConsistencyQos()
Constructor.

~TypeConsistencyQos() = default
Destructor.

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

Public Members

TypeConsistencyEnforcementQosPolicy **type_consistency**
Type consistency enforcement Qos.

DataRepresentationQosPolicy **representation**
Data Representation Qos.

ViewStateKind

enum dds::fastdds::eprosima::**ViewStateKind**
Values:

enumerator NEW

enumerator NOT_NEW

Topic

Topic

class eprosima::fastdds::dds::**Topic** : **public** eprosima::fastdds::dds::*DomainEntity*, **public** eprosima::fastdds::dds
Class *TopicDescription*, represents the fact that both publications and subscriptions are tied to a single data-type

Public Functions

~Topic()
Destructor.

DomainParticipant ***get_participant**() **const override**
Getter for the *DomainParticipant*.

Return *DomainParticipant* pointer

ReturnCode_t **get_inconsistent_topic_status**(InconsistentTopicStatus &*status*)
Allows the application to retrieve the INCONSISTENT_TOPIC_STATUS status of a *Topic*.

Return RETCODE_OK

Parameters

- status: [out] Status to be retrieved.

const *TopicQos* &**get_qos**() **const**
Allows accessing the *Topic* Qos.

Return reference to *TopicQos*

ReturnCode_t **get_qos**(*TopicQos &qos*) **const**
Retrieves the *Topic* Qos.

Return RETCODE_OK

Parameters

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

ReturnCode_t **set_qos**(**const** *TopicQos &qos*)
Allows modifying the *Topic* Qos. The given Qos must be supported by the *Topic*.

Parameters

- `qos`: new *TopicQos* value to set for the *Topic*.

Return Value

- `RETCODE_IMMUTABLE_POLICY`: if a change was not allowed.
- `RETCODE_INCONSISTENT_POLICY`: if new qos has inconsistent values.
- `RETCODE_OK`: if qos was updated.

const *TopicListener* ***get_listener**() **const**
Retrieves the attached *TopicListener*.

Return pointer to *TopicListener*

ReturnCode_t **set_listener**(*TopicListener *listener*, **const** *StatusMask &mask* = *StatusMask*::*all*())
Modifies the *TopicListener*.

Return RETCODE_OK

Parameters

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

TopicDescriptionImpl ***get_impl**() **const override**
Getter for the TopicDescriptionImpl.

Return pointer to TopicDescriptionImpl

TopicDataType

class `eprosima::fastdds::dds::`**TopicDataType**
Class *TopicDataType* used to provide the DomainRTPSParticipant with the methods to serialize, deserialize and get the key of a specific data type. The user should created a class that inherits from this one, where Serialize and deserialize methods MUST be implemented. ,

Subclassed by eprosima::fastdds::dds::builtin::TypeLookup_ReplyPubSubType, eprosima::fastdds::dds::builtin::TypeLookup_RequestPubSubType

Public Functions

TopicDataType()
Constructor.

~TopicDataType()
Destructor.

bool **serialize**(void **data*, fastrtps::rtps::SerializedPayload_t **payload*) = 0
Serialize method, it should be implemented by the user, since it is abstract. It is VERY IMPORTANT that the user sets the SerializedPayload length correctly.

Return True if correct.

Parameters

- `[in] data`: Pointer to the data
- `[out] payload`: Pointer to the payload

bool **deserialize**(fastrtps::rtps::SerializedPayload_t **payload*, void **data*) = 0
Deserialize method, it should be implemented by the user, since it is abstract.

Return True if correct.

Parameters

- `[in] payload`: Pointer to the payload
- `[out] data`: Pointer to the data

std::function<uint32_t()> **getSerializedSizeProvider**
void **data* = 0Gets the SerializedSizeProvider function.

Return function

Parameters

- `data`: Pointer

void ***createData**() = 0
Create a Data Type.

Return Void pointer to the created object.

void **deleteData**(void **data*) = 0
Remove a previously created object.

Parameters

- `data`: Pointer to the created Data.

bool **getKey**(void **data*, fastrtps::rtps::InstanceHandle_t **ihandle*, bool *force_md5* = false) = 0
Get the key associated with the data.

Return True if correct.

Parameters

- `[in] data`: Pointer to the data.
- `[out] ihandle`: Pointer to the Handle.
- `[in] force_md5`: Force MD5 checking.

void **setName**(**const** char *nam)
Set topic data type name

Parameters

- nam: *Topic* data type name

const char ***getName**() **const**
Get topic data type name

Return *Topic* data type name

bool **auto_fill_type_object**() **const**
Get the type object auto-fill configuration

Return true if the type object should be auto-filled

void **auto_fill_type_object**(bool *auto_fill_type_object*)
Set the type object auto-fill configuration

Parameters

- auto_fill_type_object: new value to set

bool **auto_fill_type_information**() **const**
Get the type information auto-fill configuration

Return true if the type information should be auto-filled

void **auto_fill_type_information**(bool *auto_fill_type_information*)
Set type information auto-fill configuration

Parameters

- auto_fill_type_information: new value to set

const std::shared_ptr<*TypeIdV1*> **type_identifier**() **const**
Get the type identifier

Return *TypeIdV1*

void **type_identifier**(**const** *TypeIdV1* &*id*)
Set type identifier

Parameters

- id: new value for *TypeIdV1*

void **type_identifier**(**const** std::shared_ptr<*TypeIdV1*> *id*)
Set type identifier

Parameters

- id: shared pointer to *TypeIdV1*

const std::shared_ptr<*TypeObjectV1*> **type_object**() **const**
Get the type object

Return *TypeObjectV1*

void **type_object**(**const** *TypeObjectV1* &*object*)
Set type object

Parameters

- object: new value for *TypeObjectV1*

void **type_object**(std::shared_ptr<*TypeObjectV1*> *object*)
Set type object

Parameters

- `object`: shared pointer to *TypeObjectV1*

const std::shared_ptr<xtypes::*TypeInformation*> **type_information**() **const**
Get the type information

Return TypeInformation

void **type_information**(**const** xtypes::*TypeInformation* &*info*)
Set type information

Parameters

- `info`: new value for TypeInformation

void **type_information**(std::shared_ptr<xtypes::*TypeInformation*> *info*)
Set type information

Parameters

- `info`: shared pointer to TypeInformation

Public Members

uint32_t **m_typeSize**
Maximum serialized size of the type in bytes. If the type has unbounded fields, and therefore cannot have a maximum size, use 0.

bool **m_isGetKeyDefined**
Indicates whether the method to obtain the key has been implemented.

TopicDescription

class `eprosima::fastdds::dds::`**TopicDescription**
Class *TopicDescription*, represents the fact that both publications and subscriptions are tied to a single data-type

Subclassed by *eprosima::fastdds::dds::Topic*

Public Functions

DomainParticipant ***get_participant**() **const** = 0
Get the *DomainParticipant* to which the *TopicDescription* belongs.

Return The *DomainParticipant* to which the *TopicDescription* belongs.

const std::string &**get_name**() **const**
Get the name used to create this *TopicDescription*.

Return the name used to create this *TopicDescription*.

const std::string &**get_type_name**() **const**
Get the associated type name.

Return the type name.

TopicDescriptionImpl ***get_impl**() **const** = 0
Get the TopicDescriptionImpl

Return pointer to TopicDescriptionImpl

TopicListener

class `eprosima::fastdds::dds::`**`TopicListener`**

Class *TopicListener*, it should be used by the end user to implement specific callbacks to certain actions.

Subclassed by *eprosima::fastdds::dds::DomainParticipantListener*

Public Functions

`TopicListener``()`

Constructor.

`~TopicListener``()`

Destructor.

void **`on_inconsistent_topic`**`(`*Topic* **topic*, InconsistentTopicStatus *status*`)`

Virtual function to be implemented by the user containing the actions to be performed when another topic exists with the same name but different characteristics.

Parameters

- `topic`: *Topic*
- `status`: The inconsistent topic status

TopicQos

class `eprosima::fastdds::dds::`**`TopicQos`**

Class *TopicQos*, containing all the possible Qos that can be set for a determined *Topic*. Although these values can be set and are transmitted during the Endpoint Discovery Protocol, not all of the behaviour associated with them has been implemented in the library. Please consult each of them to check for implementation details and default values.

Public Functions

`TopicQos``()`

Constructor.

`const` *TopicDataQosPolicy* &**`topic_data`**`()` **`const`**

Getter for *TopicDataQosPolicy*

Return TopicDataQos reference

TopicDataQosPolicy &**`topic_data`**`()`

Getter for *TopicDataQosPolicy*

Return TopicDataQos reference

void **`topic_data`**`(`**`const`** *TopicDataQosPolicy* &*value*`)`

Setter for *TopicDataQosPolicy*

Parameters

- `value`: new value for the *TopicDataQosPolicy*

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

Return DurabilityQos reference

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

Return DurabilityQos reference

void **durability**(**const** *DurabilityQosPolicy* &*durability*)
Setter for *DurabilityQosPolicy*

Parameters

- `durability`: new value for the *DurabilityQosPolicy*

const *DurabilityServiceQosPolicy* &**durability_service**() **const**
Getter for *DurabilityServiceQosPolicy*

Return DurabilityServiceQos reference

DurabilityServiceQosPolicy &**durability_service**()
Getter for *DurabilityServiceQosPolicy*

Return DurabilityServiceQos reference

void **durability_service**(**const** *DurabilityServiceQosPolicy* &*durability_service*)
Setter for *DurabilityServiceQosPolicy*

Parameters

- `durability_service`: new value for the *DurabilityServiceQosPolicy*

const *DeadlineQosPolicy* &**deadline**() **const**
Getter for *DeadlineQosPolicy*

Return DeadlineQos reference

DeadlineQosPolicy &**deadline**()
Getter for *DeadlineQosPolicy*

Return DeadlineQos reference

void **deadline**(**const** *DeadlineQosPolicy* &*deadline*)
Setter for *DeadlineQosPolicy*

Parameters

- `deadline`: new value for the *DeadlineQosPolicy*

const *LatencyBudgetQosPolicy* &**latency_budget**() **const**
Getter for *LatencyBudgetQosPolicy*

Return LatencyBudgetQos reference

LatencyBudgetQosPolicy &**latency_budget**()
Getter for *LatencyBudgetQosPolicy*

Return LatencyBudgetQos reference

void **latency_budget**(**const** *LatencyBudgetQosPolicy* &*latency_budget*)
Setter for *LatencyBudgetQosPolicy*

Parameters

- `latency_budget`: new value for the *LatencyBudgetQosPolicy*

const *LivelinessQosPolicy* &**liveliness**() **const**
Getter for *LivelinessQosPolicy*

Return LivelinessQos reference

LivelinessQosPolicy &**liveliness**()
Getter for *LivelinessQosPolicy*

Return LivelinessQos reference

void **liveliness**(**const** *LivelinessQosPolicy* &*liveliness*)
Setter for *LivelinessQosPolicy*

Parameters

- `liveliness`: new value for the *LivelinessQosPolicy*

const *ReliabilityQosPolicy* &**reliability**() **const**
Getter for *ReliabilityQosPolicy*

Return ReliabilityQos reference

ReliabilityQosPolicy &**reliability**()
Getter for *ReliabilityQosPolicy*

Return ReliabilityQos reference

void **reliability**(**const** *ReliabilityQosPolicy* &*reliability*)
Setter for *ReliabilityQosPolicy*

Parameters

- `reliability`: new value for the *ReliabilityQosPolicy*

const *DestinationOrderQosPolicy* &**destination_order**() **const**
Getter for *DestinationOrderQosPolicy*

Return DestinationOrderQos reference

DestinationOrderQosPolicy &**destination_order**()
Getter for *DestinationOrderQosPolicy*

Return DestinationOrderQos reference

void **destination_order**(**const** *DestinationOrderQosPolicy* &*destination_order*)
Setter for *DestinationOrderQosPolicy*

Parameters

- `destination_order`: new value for the *DestinationOrderQosPolicy*

const *HistoryQosPolicy* &**history**() **const**
Getter for *HistoryQosPolicy*

Return HistoryQos reference

HistoryQosPolicy &**history**()
Getter for *HistoryQosPolicy*

Return HistoryQos reference

void **history**(**const** *HistoryQosPolicy* &*history*)
Setter for *HistoryQosPolicy*

Parameters

- `history`: new value for the *HistoryQosPolicy*

const *ResourceLimitsQosPolicy* &**resource_limits**() **const**
Getter for *ResourceLimitsQosPolicy*

Return ResourceLimitsQos reference

ResourceLimitsQosPolicy &**resource_limits**()
Getter for *ResourceLimitsQosPolicy*

Return ResourceLimitsQos reference

void **resource_limits**(**const** *ResourceLimitsQosPolicy* &*resource_limits*)
Setter for *ResourceLimitsQosPolicy*

Parameters

- `resource_limits`: new value for the *ResourceLimitsQosPolicy*

const *TransportPriorityQosPolicy* &**transport_priority**() **const**
Getter for *TransportPriorityQosPolicy*

Return TransportPriorityQos reference

TransportPriorityQosPolicy &**transport_priority**()
Getter for *TransportPriorityQosPolicy*

Return TransportPriorityQos reference

void **transport_priority**(**const** *TransportPriorityQosPolicy* &*transport_priority*)
Setter for *TransportPriorityQosPolicy*

Parameters

- `transport_priority`: new value for the *TransportPriorityQosPolicy*

const *LifespanQosPolicy* &**lifespan**() **const**
Getter for *LifespanQosPolicy*

Return LifespanQos reference

LifespanQosPolicy &**lifespan**()
Getter for *LifespanQosPolicy*

Return LifespanQos reference

void **lifespan**(**const** *LifespanQosPolicy* &*lifespan*)
Setter for *LifespanQosPolicy*

Parameters

- `lifespan`: new value for the *LifespanQosPolicy*

const *OwnershipQosPolicy* &**ownership**() **const**
Getter for *OwnershipQosPolicy*

Return OwnershipQos reference

OwnershipQosPolicy &**ownership**()
Getter for *OwnershipQosPolicy*

Return OwnershipQos reference

void **ownership**(**const** *OwnershipQosPolicy* &*ownership*)
Setter for *OwnershipQosPolicy*

Parameters

- `ownership`: new value for the *OwnershipQosPolicy*

const *DataRepresentationQosPolicy* &**representation**() **const**
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

DataRepresentationQosPolicy &**representation**()
Getter for *DataRepresentationQosPolicy*

Return *DataRepresentationQosPolicy* reference

void **representation**(**const** *DataRepresentationQosPolicy* &*representation*)
Setter for *DataRepresentationQosPolicy*

Parameters

- `representation`: new value for the *DataRepresentationQosPolicy*

const TopicQos `dds::fastdds::eprosima::`**TOPIC_QOS_DEFAULT**

TypeIdV1

class `eprosima::fastdds::dds::`**TypeIdV1** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosima::fastdds::d
Class *TypeIdV1*

Public Functions

TypeIdV1()
Constructor without parameters.

TypeIdV1(**const** *TypeIdV1* &*type*)
Copy constructor.

Parameters

- `type`: Another instance of *TypeIdV1*

TypeIdV1(**const** fastrtps::types::TypeIdentifier &*identifier*)
Constructor using a TypeIndentifier.

Parameters

- `identifier`: TypeIdentifier to be set

TypeIdV1(*TypeIdV1* &&*type*)
Move constructor.

Parameters

- `type`: Another instance of *TypeIdV1*

~TypeIdV1() **override** = default
Destructor.

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

const fastrtps::types::TypeIdentifier &**get**() **const**
Getter for the TypeIndentifier.

Return TypeIdentifier reference

Public Members

fastrtps::types::TypeIdentifier **m_type_identifier**
Type Identifier.

TypeInformation

class eprosima::fastdds::dds::xtypes::**TypeInformation** : **public** eprosima::fastdds::dds::Parameter_t, **publi**
Class *xtypes::TypeInformation*

Public Functions

TypeInformation()
Constructor.

TypeInformation(**const** *TypeInformation* &*type*)
Copy constructor.

Parameters

- `type`: Another instance of *TypeInformation*

TypeInformation(**const** fastrtps::types::TypeInformation &*info*)
Constructor using a fastrtps::types::TypeInformation.

Parameters

- `info`: fastrtps::types::TypeInformation to be set

TypeInformation(*TypeInformation* &&*type*)
Move Constructor.

Parameters

- `type`: Another instance of *TypeInformation*

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

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

bool **assigned**() **const**
Check if it is assigned.

Return true if assigned, false if not

void **assigned**(bool *value*)
Setter for assigned boolean.

Parameters

- `value`: Boolean to be set

Public Members

fastrtps::types::TypeInformation **type_information**
Type Information.

TypeObjectV1

class eprosima::fastdds::dds::**TypeObjectV1** : **public** eprosima::fastdds::dds::Parameter_t, **public** eprosima::fast
Class *TypeObjectV1*

Public Functions

TypeObjectV1()
Constructor.

TypeObjectV1(**const** *TypeObjectV1* &*type*)
Copy constructor.

Parameters

- type: Another instance of *TypeObjectV1*

TypeObjectV1(**const** fastrtps::types::TypeObject &*type*)
Constructor using a TypeObject.

Parameters

- type: TypeObject to be set

TypeObjectV1(*TypeObjectV1* &&*type*)
Move constructor.

Parameters

- type: Another instance of *TypeObjectV1*

~TypeObjectV1() **override** = default
Destructor.

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

const fastrtps::types::TypeObject &**get**() **const**
Getter for the TypeObject.

Return TypeObject reference

Public Members

fastrtps::types::TypeObject **m_type_object**
Type Object.

TypeSupport

class eprosima::fastdds::dds::**TypeSupport** : **public** std::shared_ptr<fastdds::dds::*TopicDataType*>
Class *TypeSupport* used to provide the DomainRTPSParticipant with the methods to serialize, deserialize and get the key of a specific data type. The user should created a class that inherits from this one, where Serialize and deserialize methods MUST be implemented.

Note This class inherits from std::shared_ptr<TopicDataType>.

Subclassed by eprosima::fastdds::dds::builtin::TypeLookup_ReplyTypeSupport, eprosima::fastdds::dds::builtin::TypeLookup_RequestTypeSupport

Public Functions

TypeSupport()
Constructor.

TypeSupport(**const** *TypeSupport* &*type*)
Copy Constructor.

Parameters

- type: Another instance of *TypeSupport*

TypeSupport(fastdds::dds::*TopicDataType* **ptr*)
TypeSupport constructor that receives a *TopicDataType* pointer.

The passed pointer will be managed by the *TypeSupport* object, so creating two *TypeSupport* from the same pointer or deleting the passed pointer will produce a runtime error.

Parameters

- ptr:

TypeSupport(fastrtps::types::DynamicPubSubType *ptr*)
TypeSupport constructor that receives a DynamicPubSubType.

It will copy the instance so the user will keep the ownership of his object.

Parameters

- ptr:

ReturnCode_t **register_type**(*DomainParticipant* **participant*) **const**
Registers the type on a participant.

Return RETCODE_BAD_PARAMETER if the type name is empty, RETCODE_PRECONDITION_NOT_MET if there is another type with the same name registered on the *DomainParticipant* and RETCODE_OK if it is registered correctly

Parameters

- participant: *DomainParticipant* where the type is going to be registered

ReturnCode_t **register_type** (*DomainParticipant* **participant*, std::string *type_name*) **const**
Registers the type on a participant.

Return RETCODE_BAD_PARAMETER if the type name is empty, RETCODE_PRECONDITION_NOT_MET if there is another type with the same name registered on the *DomainParticipant* and RETCODE_OK if it is registered correctly

Parameters

- `participant`: *DomainParticipant* where the type is going to be registered
- `type_name`: Name of the type to register

const std::string &**get_type_name** () **const**
Getter for the type name.

Return name of the data type

bool **serialize** (void **data*, fastrtps::rtps::SerializedPayload_t **payload*)
Serializes the data.

Return true if it is serialized correctly, false if not

Parameters

- `data`: Pointer to data
- `payload`: Pointer to payload

bool **deserialize** (fastrtps::rtps::SerializedPayload_t **payload*, void **data*)
Deserializes the data.

Return true if it is deserialized correctly, false if not

Parameters

- `payload`: Pointer to payload
- `data`: Pointer to data

std::function<uint32_t () > **get_serialized_size_provider**
void **data*Getter for the SerializedSizeProvider.

Return function

Parameters

- `data`: Pointer to data

void ***create_data** ()
Creates new data.

Return Pointer to the data

void **delete_data** (void **data*)
Deletes data.

Parameters

- `data`: Pointer to the data to delete

bool **get_key** (void **data*, fastrtps::rtps::InstanceHandle_t **i_handle*, bool *force_md5* = false)
Getter for the data key.

Return true if the key is returned, false if not

Parameters

- `data`: Pointer to data
- `i_handle`: InstanceHandle pointer to store the key
- `force_md5`: boolean to force md5 (default: false)

bool **empty** () **const**
Check if the *TypeSupport* is empty.

Return true if empty, false if not

6.21 Introduction

eProsima FASTRTPSGEN is a Java application that generates source code using the data types defined in an IDL file. This generated source code can be used in your applications in order to publish and subscribe to a topic of your defined type.

To declare your structured data, you have to use IDL (Interface Definition Language) format. IDL is a specification language, made by OMG (Object Management Group), which describes an interface in a language-independent way, enabling communication between software components that do not share the same language.

eProsima FASTRTPSGEN is a tool that reads IDL files and parses a subset of the OMG IDL specification to generate serialization source code. This subset includes the data type descriptions included in *Defining a data type via IDL*. The rest of the file content is ignored.

eProsima FASTRTPSGEN generated source code uses Fast CDR: a C++11 library that provides a serialization mechanism. In this case, as indicated by the RTPS specification document, the serialization mechanism used is CDR. The standard CDR (Common Data Representation) is a transfer syntax low-level representation for transfer between agents, mapping from data types defined in OMG IDL to byte streams.

One of the main features of eProsima FASTRTPSGEN is to avoid the users the trouble of knowing anything about serialization or deserialization procedures. It also provides an initial implementation of a publisher and a subscriber using eProsima RTPS library.

6.21.1 Compile

In order to compile *fastrtpsgen* you first need to have gradle and java JDK installed (please, check the JDK recommended version for the gradle version you have installed).

To compile *fastrtpsgen* java application, you will need to download its source code from the Fast-RPTS-Gen repository and with `--recursive` option and compile it calling `gradle assemble`. For more details see *Compile*.

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

The generated java application can be found at `share/fastrtps` and more user friendly scripts at `scripts` folders. If you want to make these scripts available from anywhere you can add the `scripts` folder path to your `PATH` environment variable.

6.22 Building a publish/subscribe application

First of all you need to follow the steps outlined in *Requirements* for the installation of required packages, and in *Installation from Binaries* or *Installation from Sources* for the installation of *eprosima Fast RTPS* and all its dependencies. If you have followed the steps, you should have Fast DDS, Fast CDR, and Fast-RTPS-Gen installed.

To build a minimal application, you must first define the Topic. To do this, the data type of the Topic is defined by means of an IDL file. An example of this is shown below. In this case, a single string is sufficient. Topics are explained in more detail in *Topic*, while the topic data types to be defined using IDL are presented in *Introduction*.

```
mkdir HelloWorldExample && cd HelloWorldExample
```

```
// HelloWorld.idl
struct HelloWorld
{
    string message;
};
```

Now we need to translate this file to something Fast RTPS understands. For this we have a code generation tool called Fast DDS-Gen (see *Execution and IDL Definition*), which can do two different things:

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

It is the second option which is used to create this publish/subscribe application, while the first option is applied in this other tutorial: *Writing a simple publisher and subscriber application*.

6.22.1 Generating a minimal functional example

The Fast DDS-Gen tool is used to build the publication/subscription application. This is a Java application for building a fully functional publication/subscription application from an Interface Definition Language (IDL) file that defines the Topic under which messages are published and received. The application generated allows the creation of as many publishers and subscribers as desired, all belonging to the same Domain and communicating using the same Topic.

First, locate the installation directories of Fast DDS, Fast CDR and add these directories to your PATH.

- If you have followed the **colcon installation**, run the following command:

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

- However, if you have followed the **manual installation** the libraries are already accessible.

The application files are generated using the following command. The *-example* option creates an example application, and the CMake files needed to build it.

- On Linux:
  - If you have followed the **colcon installation**:

```
<path-to-Fast-DDS-workspace>/src/fastrtpsgen/scripts/fastrtpsgen -example␣
↪CMake HelloWorld.idl
```

  - However, if you have followed the **manual installation**, run:

```
<path-to-Fast-RTPS-Gen>/scripts/fastrtpsgen -example CMake HelloWorld.idl
```

- On Windows:

- If you have followed the **colcon installation**:

```
<path-to-Fast-DDS-workspace>/src/fastrtpsgen/scripts/fastrtpsgen.bat -
↪example CMake HelloWorld.idl
```

- However, if you have followed the **manual installation**, run:

```
<path-to-Fast-RTPS-Gen>/scripts/fastrtpsgen.bat -example CMake HelloWorld.
↪idl
```

Finally, we compile the generated code.

- On Linux:

```
mkdir build && cd build
cmake ..
make
```

- On Windows:

```
mkdir build && cd build
cmake -G "Visual Studio 15 2017 Win64" ..
cmake --build .
```

The application build can be used to spawn any number of publishers and subscribers associated with your topic.

- On Linux:

```
./HelloWorld publisher
./HelloWorld subscriber
```

- On Windows:

```
HelloWorld.exe publisher
HelloWorld.exe subscriber
```

Each time you press <Enter> on the Publisher, a new datagram is generated, sent over the network and receiver by Subscribers currently online. If more than one subscriber is available, it can be seen that the message is equally received on all listening nodes.

You can also modify any values on your custom, IDL-generated data type.

```
    HelloWorld sample; //Auto-generated container class for topic data from␣
↪FastRTPSGen
    sample.msg("Hello there!"); // Add contents to the message
    publisher->write(&sample); //Publish
```

Take a look at the eProsima Fast RTPS examples on github for ideas on how to improve this basic application through different configuration options, and for examples of advanced Fast DDS features.

You may need to set up a special rule in your Firewall for *eprosima Fast DDS* to work correctly on Windows.

6.23 Execution and IDL Definition

6.23.1 Building publisher/subscriber code

This section guides you through the usage of this Java application and briefly describes the generated files.

Once you added `scripts` folder to your `PATH`, the Java application can be executed using the following scripts depending on if you are on Windows or Linux:

```
> fastrtpsgen.bat
$ fastrtpsgen
```

In case you didn't modified your `PATH` you can find these scripts in your `<fastrtpsgen_directory>/scripts` folder.

The expected argument list of the application is:

```
fastrtpsgen [<options>] <IDL file> [<IDL file> ...]
```

Where the option choices are:

Option	Description
-help	Shows the help information.
-version	Shows the current version of eProsima FASTRTPSGEN.
-d <directory>	Sets the output directory where the generated files are created.
-I <directory>	Add directory to preprocessor include paths.
-t <directory>	Sets a specific directory as a temporary directory.
-example <platform>	Generates an example and a solution to compile the generated source code for a specific platform. The help command shows the supported platforms.
-replace	Replaces the generated source code files even if they exist.
-ppDisable	Disables the preprocessor.
-ppPath	Specifies the preprocessor path.
-typeobject	Generates *TypeObject* files for the IDL provided and modifies MyType constructor to register the TypeObject representation into the factory.

For more information about TypeObject representation read *Dynamic Topic Types*.

6.23.2 Defining a data type via IDL

The following table shows the basic IDL types supported by *fastrtpsgen* 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

fastrtpsgen supports unidimensional and multidimensional arrays. Arrays are always mapped to std::array containers. The following table shows the array types supported and how they map.

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

fastrtpsgen supports sequences, which map into the STD vector container. The following table represents how the map between IDL and C++11 is handled.

IDL	C++11
sequence<char>	std::vector<char>
sequence<octet>	std::vector<uint8_t>
sequence<short>	std::vector<int16_t>
sequence<unsigned short>	std::vector<uint16_t>
sequence<long>	std::vector<int32_t>
sequence<unsigned long>	std::vector<uint32_t>
sequence<long long>	std::vector<int64_t>
sequence<unsigned long long>	std::vector<uint64_t>
sequence<float>	std::vector<float>
sequence<double>	std::vector<double>

Maps

fastrtpsgen supports maps, which are equivalent to the STD map container. The equivalence between types is handled in the same way as for *sequences*.

IDL	C++11
map<char, unsigned long long>	std::map<char, uint64_T>

Structures

You can define an IDL structure with a set of members with multiple types. It will be converted into a C++ class with each member mapped as an attribute plus methods to *get* and *set* each member.

The following IDL structure:

```
struct Structure
{
    octet octet_value;
    long long_value;
    string string_value;
};
```

Would be converted to:

```
class Structure
{
public:
   Structure();
   ~Structure();
   Structure(const Structure &x);
   Structure(Structure &&x);
   Structure& operator=( const Structure &x);
   Structure& operator=(Structure &&x);

   void octet_value(uint8_t _octet_value);
   uint8_t octet_value() const;
   uint8_t& octet_value();
   void long_value(int64_t _long_value);
   int64_t long_value() const;
   int64_t& long_value();
   void string_value(const std::string
      &_string_value);
   void string_value(std::string &&_string_value);
   const std::string& string_value() const;
   std::string& string_value();

private:
   uint8_t m_octet_value;
   int64_t m_long_value;
   std::string m_string_value;
};
```

Structures can inherit from other structures, extending their member set.

```
struct ParentStruct
{
```

(continues on next page)

(continued from previous page)

```
    octet parent_member;
};

struct ChildStruct : ParentStruct
{
    long child_member;
};
```

In this case, the resulting C++ code will be:

```
class ParentStruct
{
    ...
};

class ChildStruct : public ParentStruct
{
    ...
};
```

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 access functions to the union members and the discriminant.

The following IDL union:

```
union Union switch(long)
{
   case 1:
    octet octet_value;
  case 2:
    long long_value;
  case 3:
    string string_value;
};
```

Would be converted to:

```
class Union
{
public:
   Union();
   ~Union();
   Union(const Union &x);
   Union(Union &&x);
   Union& operator=(const Union &x);
   Union& operator=(Union &&x);

   void d(int32t __d);
   int32_t _d() const;
   int32_t& _d();

   void octet_value(uint8_t _octet_value);
```

(continues on next page)

(continued from previous page)

```
   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, int> c;
};
```

The type MyBitset will store a total of 25 bits (3 + 10 + 12) and will require 32 bits in memory (lowest primitive type to store the bitset's size).

- The bitfield 'a' allows us to access to the first 3 bits (0..2).
- The bitfield 'b' allows us to access to the next 10 bits (3..12).
- The bitfield 'c' allows us to access to the next 12 bits (13..24).

The resulting C++ code will be similar to:

```
class MyBitset
{
public:
    void a(char _a);
    char a() const;

    void b(uint16_t _b);
    uint16_t b() const;

    void c(int32_t _c);
    int32_t c() const;
private:
    std::bitset<25> m_bitset;
};
```

Internally is stored as a std::bitset. For each bitfield, getter and setter methods are generated with the smaller possible primitive unsigned type to access it. In the case of bitfield 'c', the user has established that this accessing type will be **int**, so the generated code uses **int32_t** instead of automatically use **uint16_t**.

Bitsets can inherit from other bitsets, extending their member set.

```
bitset ParentBitset
{
    bitfield<3> parent_member;
};

bitset ChildBitset : ParentBitset
{
    bitfield<10> child_member;
};
```

In this case, the resulting C++ code will be:

```
class ParentBitset
{
    ...
};

class ChildBitset : public ParentBitset
{
    ...
};
```

Note that in this case, ChildBitset will have two `std::bitset` members, one belonging to ParentBitset and the other belonging to ChildBitset.

Enumerations

An enumeration in IDL format is a collection of identifiers that have a numeric value associated. 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.

Keyed Types

In order to use keyed topics, the user should define some key members inside the structure. This is achieved by writing "@Key" before the members of the structure you want to use 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;
};
```

fastrtpsgen 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.

Including other IDL files

You can include another IDL files in yours in order to use data types defined in them. *fastrtpsgen* uses a C/C++ preprocessor for this purpose, and you can use `#include` directive to include an IDL file.

```
#include "OtherFile.idl"
#include <AnotherFile.idl>
```

If *fastrtpsgen* doesn't find a C/C++ preprocessor in default system paths, you could specify the preprocessor path using parameter `-ppPath`. If you want to disable the usage of the preprocessor, you could use the parameter `-ppDisable`.

Annotations

The application allows the user to define and use their own annotations as defined in the IDL 4.2 standard. User annotations will be passed to TypeObject generated code if the `-typeobject` argument was used.

```
@annotation MyAnnotation
{
    long value;
    string name;
};
```

Additionally, the following standard annotations are builtin (recognized and passed to TypeObject when unimplemented).

Annotation	Implemented behavior
@id	Unimplemented.
@autoid	Unimplemented.
@optional	Unimplemented.
@extensibility	Unimplemented.
@final	Unimplemented.
@appendable	Unimplemented.
@mutable	Unimplemented.
@position	Used by *bitmasks*.
@value	Allows to set a constant value to any element.
@key	Alias for eProsima's @Key annotation.
@must_understand	Unimplemented.
@default_literal	Allows selecting one member as the default within a collection.
@default	Allows specifying the default value of the annotated element.
@range	Unimplemented.
@min	Unimplemented.
@max	Unimplemented.
@unit	Unimplemented.
@bit_bound	Allows setting a size to a *bitmasks*.
@external	Unimplemented.
@nested	Unimplemented.
@verbatim	Unimplemented.
@service	Unimplemented.
@oneway	Unimplemented.
@ami	Unimplemented.
@non_serialized	The annotated member will be omitted from serialization.

Most unimplemented annotations are related to Extended Types.

IDL 4.2 aliases

IDL 4.2 allows using the following names for primitive types:

int8
uint8
int16
uint16
int32
uint32
int64
uint64

Forward declaration

The application allows forward declarations:

```
struct ForwardStruct;

union ForwardUnion;

struct ForwardStruct
{
    ForwardUnion fw_union;
};

union ForwardUnion switch (long)
{
    case 0:
        ForwardStruct fw_struct;
    default:
        string empty;
};
```

As the example shows, this allows declaring inter-dependant structures, unions, etc.

6.24 Version 2.0.0

This release has the following **API breaks**:

- eClock API, which was deprecated on v1.9.1, has been removed
- *eprosima::fastrtps::rtps::RTPSDomain::createParticipant* methods now have an additional first argument *domain_id*
- Data member *domainId* has been removed from *eprosima::fastrtps::rtps::RTPSParticipantAttributes* and added to *eprosima::fastrtps::ParticipantAttributes*

Users should also be aware of the following **deprecation announcement**:

- All classes inside the namespace *eprosima::fastrtps* should be considered deprecated. Equivalent functionality is offered through namespace *eprosima::fastdds*.
- Namespaces beneath *eprosima::fastrtps* are not included in this deprecation, i.e. *eprosima::fastrtps::rtps* can still be used)

This release adds the following **features**:

- Added support for register/unregister/dispose instance
- Added DDS compliant API. This new API exposes all the functionality of the Publisher-Subscriber Fast RTPS API adhering to the Data Distribution Service (DDS) version 1.4 specification
- Added Security Logging Plugin (contributed by Cannonical Ltd.)
- Bump to FastCDR v1.0.14

It also includes the following bug fixes and improvements:

- Support for OpenSSL 1.1.1d and higher
- Support for latest versions of gtest
- Support for FreeBSD
- Fault tolerance improvements to Shared Memory transport
- Fixed segfault when no network interfaces are detected
- Correctly ignoring length of *PID_SENTINEL* on parameter list
- Improved traffic on PDP simple mode
- Reduced CPU and memory usage

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.25 Previous versions

6.25.1 Version 1.10.0

This release adds the following features:

- New built-in *Shared memory Transport (SHM)*
- Transport API refactored to support locator iterators
- Added subscriber API to retrieve info of first non-taken sample
- Added parameters to fully avoid dynamic allocations
- History of built-in endpoints can be configured
- Bump to FastCDR v1.0.13.
- Bump to Fast-RTPS-Gen v1.0.4.
- Require CMake 3.5 but use policies from 3.13

It also includes the following bug fixes and improvements:

- Fixed alignment on parameter lists
- Fixed error sending more than 256 fragments.
- Fix handling of STRICT_REALTIME.
- Fixed submessage_size calculation on last data_frag.
- Solved an issue when recreating a publishing participant with the same GUID.

- Solved an issue where a publisher could block on write for a long time when a new subscriber (late joiner) is matched, if the publisher had already sent a large number of messages.
- Correctly handling the case where lifespan expires at the same time on several samples.
- Solved some issues regarding liveliness on writers with no readers.
- Correctly removing changes from histories on keyed topics.
- Not reusing cache change when sample does not fit.
- Fixed custom wait_until methods when time is in the past.
- Several data races and ABBA locks fixed.
- Reduced CPU and memory usage.
- Reduced flakiness of liveliness tests.
- Allow for more use cases on performance tests.

Several bug fixes on discovery server:

- Fixed local host communications.
- Correctly trimming server history.
- Fixed backup server operation.
- Fixed timing issues.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*. If you are upgrading from a version older than 1.10.0, regenerating the code is *recommended*.

6.25.2 Version 1.9.4

This release adds the following features:

- Intra-process delivery mechanism is now active by default.
- Synchronous writers are now allowed to send fragments.
- New memory mode DYNAMIC_RESERVE on history pool.
- Performance tests can now be run on Windows and Mac.
- XML profiles for requester and replier.

It also includes the following bug fixes and improvements:

- Bump to FastCDR v1.0.12.
- Bump to Fast-RTPS-Gen v1.0.3.
- Fixed deadlock between PDP and StatefulReader.
- Improved CPU usage and allocations on timed events management.
- Performance improvements on reliable writers.
- Fixing bugs when Intra-process delivery is activated.
- Reducing dynamic allocations and memory footprint.
- Improvements and fixes on performance tests.
- Other minor bug fixes and improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.3 Version 1.9.3

This release adds the following features:

- Participant discovery filtering flags.
- Intra-process delivery mechanism opt-in.

It also includes the following bug fixes and improvements:

- Bump to Fast-RTPS-Gen v1.0.2.
- Bring back compatibility with XTypes 1.1 on PID_TYPE_CONSISTENCY.
- Ensure correct alignment when reading a parameter list.
- Add CHECK_DOCUMENTATION *cmake* option.
- EntityId_t and GuidPrefix_t have now their own header files.
- Fix potential race conditions and deadlocks.
- Improve the case where *check_acked_status* is called between reader matching process and its acknack reception.
- RTPSMessageGroup_t instances now use the thread-local storage.
- FragmentedChangePitStop manager removed.
- Remove the data fragments vector on CacheChange_t.
- Only call find_package for TinyXML2 if third-party options are off
- Allow XMLProfileManager methods to not show error log messages if a profile is not found.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.4 Version 1.9.2

This release includes the following feature:

- Multiple initial PDP announcements.
- Flag to avoid builtin multicast.

It also adds the following bug fixes and improvements:

- Bump to Fast-RTPS-Gen v1.0.1.
- Bump to IDL-Parser v1.0.1.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.5 Version 1.9.1

This release includes the following features:

- Fast-RTPS-Gen is now an independent project.
- Header **eClock.h** is now marked as deprecated.

It also adds the following bug fixes and improvements:

- Bump to FastCDR v1.0.11.
- Installation from sources documentation fixed.
- Fixed assertion on WriterProxy.
- Fixed potential fall through while parsing Parameters.
- Removed deprecated guards causing compilation errors in some 32 bits platforms.
- *addTOCDRMessage* method is now exported in the DLL, fixing issues related with Parameters' constructors.
- Improve windows performance by avoiding usage of *_Cnd_timedwait* method.
- Fixed reported communication issues by sending multicast through *localhost* too.
- Fixed potential race conditions and deadlocks.
- Eliminating use of *acceptMsgDirectTo*.
- Discovery Server framework reconnect/recreate strategy.
- Removed unused folders.
- Restored subscriber API.
- SequenceNumber_t improvements.
- Added STRICT_REALTIME *cmake* option.
- SubscriberHistory improvements.
- Assertion of participant liveliness by receiving RTPS messages from the remote participant.
- Fixed error while setting next deadline event in *create_new_change_with_params*.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.6 Version 1.9.0

This release includes the following features:

- Partial implementation of allocation QoS.
- Implementation of Discovery Server.
- Implementation of non-blocking calls.

It also adds the following bug fixes and improvements:

- Added sliding window to BitmapRange.
- Modified default behavior for unknown writers.
- A *Flush()* method was added to the logger to ensure all info is logged.
- A test for loading *Duration_t* from XML was added.

- Optimized WLP when removing local writers.
- Some liveliness tests were updated so that they are more stable on Windows.
- A fix was added to *CMakeLists.txt* for installing static libraries.
- A fix was added to performance tests so that they can run on the RT kernel.
- Fix for race condition on built-in protocols creation.
- Fix for setting *nullptr* in a *fixed_string*.
- Fix for v1.8.1 not building with -DBUILD_JAVA=ON.
- Fix for GAP messages not being sent in some cases.
- Fix for coverity report.
- Several memory issues fixes.
- *fastrtps.repos* file was updated.
- Documentation for building with Colcon was added.
- Change CMake configuration directory if INSTALLER_PLATFORM is set.
- IDL sub-module updated to current version.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.7 Version 1.8.4

This release adds the following **feature**:

- XML profiles for *requester* and *replier*

It also has the following **important bug fixes**:

- Solved an issue when recreating a publishing participant with the same GUID (either on purpose or by chance)
- Solved an issue where a publisher could block on *write* for a long time when, after a large number of samples have been sent, a new subscriber is matched.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*

6.25.8 Version 1.8.3

This release adds the following bug fixes and improvements:

- Fix serialization of TypeConsistencyEnforcementQosPolicy.
- Bump to Fast-RTPS-Gen v1.0.2.
- Bump to IDL-Parser v1.0.1.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*

6.25.9 Version 1.8.2

This release includes the following features:

- Modified unknown writers default behavior.
- Multiple initial PDP announcements.
- Flag to avoid builtin multicast.
- *STRICT_REALTIME* compilation flag.

It also adds the following bug fixes and improvements:

- Fix for setting *nullptr* in a fixed string.
- Fix for not sending GAP in several cases.
- Solve *Coverity* report issues.
- Fix issue of *fastrtpsgen* failing to open *IDL.g4* file.
- Fix unnamed lock in *AESGCMGMAC_KeyFactory.cpp*.
- Improve *XMLProfiles* example.
- Multicast is now sent through *localhost* too.
- *BitmapRange* now implements sliding window.
- Improve *SequenceNumber_t* struct.
- Participant's liveliness is now asserted when receiving any RTPS message.
- Fix leak on RemoteParticipantLeaseDuration.
- Modified default values to improve behavior in *Wi-Fi* scenarios.
- *SubscriberHistory* improvements.
- Removed use of *acceptMsgDirectTo*.
- *WLP* improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*

6.25.10 Version 1.8.1

This release includes the following features:

- Implementation of *LivelinessQosPolicy* QoS.

It also adds the following bug fixes and improvements:

- Fix for get_change on history, which was causing issues during discovery.
- Fix for announcement of participant state, which was sending ParticipantBuiltinData twice.
- Fix for closing multicast UDP channel.
- Fix for race conditions in SubscriberHistory, UDPTransportInterface and StatefulReader.
- Fix for lroundl error on Windows in Time_t.
- CDR & IDL submodules update.
- Use of java 1.8 or greater for fastrtpsgen.jar generation.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.11 Version 1.8.0

This release included the following features:

- Implementation of IDL 4.2.
- Implementation of *DeadlineQosPolicy* QoS.
- Implementation of *LifespanQosPolicy* QoS.
- Implementation of *DisablePositiveACKsQosPolicy* QoS.
- Secure sockets on TCP transport (*TLS over TCP*).

It also adds the following improvements and bug fixes:

- Real-time improvements: non-blocking write calls for best-effort writers, addition of fixed size strings, fixed size bitmaps, resource limited vectors, etc.
- Duration parameters now use nanoseconds.
- Configuration of participant mutation tries.
- Automatic calculation of the port when a value of 0 is received on the endpoint custom locators.
- Non-local addresses are now filtered from whitelists.
- Optimization of check for acked status for stateful writers.
- Linked libs are now not exposed when the target is a shared lib.
- Limitation on the domain ID has been added.
- UDP non-blocking send is now optional and configurable via XML.
- Fix for non-deterministic tests.
- Fix for ReaderProxy history being reloaded incorrectly in some cases.
- Fix for RTPS domain hostid being potentially not unique.
- Fix for participants with different lease expiration times failing to reconnect.

Known issues

- When using TPC transport, sometimes callbacks are not invoked when removing a participant due to a bug in ASIO.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.12 Version 1.7.2

This release fixes an important bug:

- Allocation limits on subscribers with a KEEP_LAST QoS was taken from resource limits configuration and didn't take history depth into account.

It also has the following improvements:

- Vendor FindThreads.cmake from CMake 3.14 release candidate to help with sanitizers.
- Fixed format of gradle file.

Some other minor bugs and performance improvements.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.13 Version 1.7.1

This release included the following features:

- LogFileConsumer added to the logging system.
- Handle FASTRTPS_DEFAULT_PROFILES_FILE environment variable indicating the default profiles XML file.
- XML parser made more restrictive and with better error messages.

It also fixes some important bugs: * Fixed discovery issues related to the selected network interfaces on Windows. * Improved discovery times. * Workaround ASIO issue with multicast on QNX systems. * Improved TCP transport performance. * Improved handling of key-only data submessages.

Some other minor bugs and performance improvements.

KNOWN ISSUES

- Allocation limits on subscribers with a KEEP_LAST QoS is taken from resource limits configuration and doesn't take history depth into account.

Note: If you are upgrading from a version older than 1.7.0, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.14 Version 1.7.0

This release included the following features:

- *TCP Transport*.
- *Dynamic Topic Types*.
- Security 1.1 compliance.

Also bug fixing, allocation and performance improvements.

Note: If you are upgrading from an older version, it is **required** to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.15 Version 1.6.0

This release included the following features:

- *Persistence*.
- Security access control plugin API and builtin *Access control plugin: DDS:Access:Permissions* plugin.

Also bug fixing.

Note: If you are upgrading from an older version than 1.4.0, it is advisable to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.16 Version 1.5.0

This release included the following features:

- Configuration of Fast RTPS entities through XML profiles.
- Added heartbeat piggyback support.

Also bug fixing.

Note: If you are upgrading from an older version than 1.4.0, it is advisable to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.17 Version 1.4.0

This release included the following:

- Added secure communications.
- Removed all Boost dependencies. Fast RTPS is not using Boost libraries anymore.
- Added compatibility with Android.
- Bug fixing.

Note: After upgrading to this release, it is advisable to regenerate generated source from IDL files using *fastrtpsgen*.

6.25.18 Version 1.3.1

This release included the following:

- New examples that illustrate how to tweak Fast RTPS towards different applications.
- Improved support for embedded Linux.
- Bug fixing.

6.25.19 Version 1.3.0

This release introduced several new features:

- Unbound Arrays support: Now you can send variable size data arrays.
- Extended Fragmentation Configuration: It allows you to setup a Message/Fragment max size different to the standard 64Kb limit.
- Improved logging system: Get even more introspection about the status of your communications system.
- Static Discovery: Use XML to map your network and keep discovery traffic to a minimum.
- Stability and performance improvements: A new iteration of our built-in performance tests will make benchmarking easier for you.
- ReadTheDocs Support: We improved our documentation format and now our installation and user manuals are available online on ReadTheDocs.

6.25.20 Version 1.2.0

This release introduced two important new features:

- Flow Controllers: A mechanism to control how you use the available bandwidth avoiding data bursts. The controllers allow you to specify the maximum amount of data to be sent in a specific period of time. This is very useful when you are sending large messages requiring fragmentation.
- Discovery Listeners: Now the user can subscribe to the discovery information to know the entities present in the network (Topics, Publishers & Subscribers) dynamically without prior knowledge of the system. This enables the creation of generic tools to inspect your system.

But there is more:

- Full ROS2 Support: Fast RTPS is used by ROS2, the upcoming release of the Robot Operating System (ROS).
- Better documentation: More content and examples.
- Improved performance.
- Bug fixing.

INDEX

D

dds::fastdds::eprosima::DATAREADER_QOS_DEFAULT
(*C++ member*), 406
dds::fastdds::eprosima::DataRepresentationId
(*C++ enum*), 331
dds::fastdds::eprosima::DataRepresentationId::XCDR2_DATA_REPRESENTATION
(*C++ enumerator*), 331
dds::fastdds::eprosima::DataRepresentationId::XCDR_DATA_REPRESENTATION
(*C++ enumerator*), 331
dds::fastdds::eprosima::DataRepresentationId::XML_DATA_REPRESENTATION
(*C++ enumerator*), 331
dds::fastdds::eprosima::DATAWRITER_QOS_DEFAULT
(*C++ member*), 390
dds::fastdds::eprosima::DestinationOrderQosPolicyKind
(*C++ enum*), 333
dds::fastdds::eprosima::DestinationOrderQosPolicyKind::BY_RECEPTION_TIMESTAMP_DESTINATIONORDER
(*C++ enumerator*), 333
dds::fastdds::eprosima::DestinationOrderQosPolicyKind::BY_SOURCE_TIMESTAMP_DESTINATIONORDER
(*C++ enumerator*), 333
dds::fastdds::eprosima::DurabilityQosPolicyKind
(*C++ enum*), 334
dds::fastdds::eprosima::DurabilityQosPolicyKind::PERSISTENT_DURABILITY_QOS
(*C++ enumerator*), 335
dds::fastdds::eprosima::DurabilityQosPolicyKind::TRANSIENT_DURABILITY_QOS
(*C++ enumerator*), 335
dds::fastdds::eprosima::DurabilityQosPolicyKind::TRANSIENT_LOCAL_DURABILITY_QOS
(*C++ enumerator*), 334
dds::fastdds::eprosima::DurabilityQosPolicyKind::VOLATILE_DURABILITY_QOS
(*C++ enumerator*), 334
dds::fastdds::eprosima::HistoryQosPolicyKind
(*C++ enum*), 339
dds::fastdds::eprosima::HistoryQosPolicyKind::KEEP_ALL_HISTORY_QOS
(*C++ enumerator*), 339
dds::fastdds::eprosima::HistoryQosPolicyKind::KEEP_LAST_HISTORY_QOS
(*C++ enumerator*), 339
dds::fastdds::eprosima::InconsistentTopicStatus
(*C++ type*), 358
dds::fastdds::eprosima::InstanceStateKind
(*C++ enum*), 406
dds::fastdds::eprosima::InstanceStateKind::ALIVE
(*C++ enumerator*), 406
dds::fastdds::eprosima::InstanceStateKind::NOT_ALIVE_DISPOSED
(*C++ enumerator*), 406
dds::fastdds::eprosima::InstanceStateKind::NOT_ALIV
(*C++ enumerator*), 406
dds::fastdds::eprosima::LivelinessLostStatus
(*C++ type*), 361
dds::fastdds::eprosima::LivelinessQosPolicyKind
(*C++ enum*), 341
dds::fastdds::eprosima::LivelinessQosPolicyKind::AU
(*C++ enumerator*), 341
dds::fastdds::eprosima::LivelinessQosPolicyKind::MA
(*C++ enumerator*), 341
dds::fastdds::eprosima::LivelinessQosPolicyKind::MA
(*C++ enumerator*), 341
dds::fastdds::eprosima::OfferedDeadlineMissedStatus
(*C++ type*), 359
dds::fastdds::eprosima::OfferedIncompatibleQosStat
(*C++ type*), 359
dds::fastdds::eprosima::OwnershipQosPolicyKind
(*C++ enum*), 342
dds::fastdds::eprosima::OwnershipQosPolicyKind::EXC
(*C++ enumerator*), 342
dds::fastdds::eprosima::OwnershipQosPolicyKind::SHA
(*C++ enumerator*), 342
dds::fastdds::eprosima::PARTICIPANT_QOS_DEFAULT
(*C++ member*), 379
dds::fastdds::eprosima::PresentationQosPolicyAccess
(*C++ enum*), 346
dds::fastdds::eprosima::PresentationQosPolicyAccess
(*C++ enumerator*), 346
dds::fastdds::eprosima::PresentationQosPolicyAccess
(*C++ enumerator*), 346
dds::fastdds::eprosima::PresentationQosPolicyAccess
(*C++ enumerator*), 346
dds::fastdds::eprosima::PropertyPolicyQos
(*C++ type*), 346
dds::fastdds::eprosima::PUBLISHER_QOS_DEFAULT
(*C++ member*), 395
dds::fastdds::eprosima::PublishModeQosPolicyKind
(*C++ enum*), 347
dds::fastdds::eprosima::PublishModeQosPolicyKind::A
(*C++ enumerator*), 347
dds::fastdds::eprosima::PublishModeQosPolicyKind::S

(*C++ enumerator*), 347
dds::fastdds::eprosima::QosPolicyCountSeq (*C++ type*), 360
dds::fastdds::eprosima::ReliabilityQosPolicyKind (*C++ enum*), 350
dds::fastdds::eprosima::ReliabilityQosPolicyKind::BEST_EFFORT (*C++ enumerator*), 350
dds::fastdds::eprosima::ReliabilityQosPolicyKind::RELIABLE (*C++ enumerator*), 350
dds::fastdds::eprosima::RequestedDeadlineMissedStatus (*C++ type*), 360
dds::fastdds::eprosima::RequestedIncompatibleQosStatus (*C++ type*), 361
dds::fastdds::eprosima::SampleLostStatus (*C++ type*), 361
dds::fastdds::eprosima::SampleRejectedStatusKind (*C++ enum*), 361
dds::fastdds::eprosima::SampleRejectedStatusKind::NOT_REJECTED (*C++ enumerator*), 361
dds::fastdds::eprosima::SampleRejectedStatusKind::REJECTED_BY_INSTANCES_LIMIT (*C++ enumerator*), 361
dds::fastdds::eprosima::SampleRejectedStatusKind::REJECTED_BY_SAMPLES_LIMIT (*C++ enumerator*), 361
dds::fastdds::eprosima::SampleRejectedStatusKind::REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT (*C++ enumerator*), 361
dds::fastdds::eprosima::SampleStateKind (*C++ enum*), 408
dds::fastdds::eprosima::SampleStateKind::NOT_READ (*C++ enumerator*), 408
dds::fastdds::eprosima::SampleStateKind::READ (*C++ enumerator*), 408
dds::fastdds::eprosima::SUBSCRIBER_QOS_DEFAULT (*C++ member*), 413
dds::fastdds::eprosima::TOPIC_QOS_DEFAULT (*C++ member*), 423
dds::fastdds::eprosima::TypeConsistencyKind (*C++ enum*), 354
dds::fastdds::eprosima::TypeConsistencyKind::ALLOW_TYPE_COERCION (*C++ enumerator*), 354
dds::fastdds::eprosima::TypeConsistencyKind::DISALLOW_TYPE_COERCION (*C++ enumerator*), 354
dds::fastdds::eprosima::ViewStateKind (*C++ enum*), 414
dds::fastdds::eprosima::ViewStateKind::NEW (*C++ enumerator*), 414
dds::fastdds::eprosima::ViewStateKind::NOT_NEW (*C++ enumerator*), 414

E

eprosima::fastdds::dds::BaseStatus (*C++ struct*), 357
eprosima::fastdds::dds::BaseStatus::total_count (*C++ member*), 357
eprosima::fastdds::dds::BaseStatus::total_count_cha (*C++ member*), 357
eprosima::fastdds::dds::DataReader (*C++ class*), 396
eprosima::fastdds::dds::DataReader::~DataReader (*C++ function*), 397
eprosima::fastdds::dds::DataReader::enable (*C++ function*), 397
eprosima::fastdds::dds::DataReader::get_first_untak (*C++ function*), 397
eprosima::fastdds::dds::DataReader::get_instance_ha (*C++ function*), 397
eprosima::fastdds::dds::DataReader::get_listener (*C++ function*), 399
eprosima::fastdds::dds::DataReader::get_liveliness_ (*C++ function*), 399
eprosima::fastdds::dds::DataReader::get_qos (*C++ function*), 398
eprosima::fastdds::dds::DataReader::get_requested_c (*C++ function*), 398
eprosima::fastdds::dds::DataReader::get_requested_i (*C++ function*), 398
eprosima::fastdds::dds::DataReader::get_subscriber (*C++ function*), 399
eprosima::fastdds::dds::DataReader::get_topicdescri (*C++ function*), 398
eprosima::fastdds::dds::DataReader::guid (*C++ function*), 397
eprosima::fastdds::dds::DataReader::read_next_sampl (*C++ function*), 396
eprosima::fastdds::dds::DataReader::set_listener (*C++ function*), 398
eprosima::fastdds::dds::DataReader::set_qos (*C++ function*), 398
eprosima::fastdds::dds::DataReader::take_next_sampl (*C++ function*), 396
eprosima::fastdds::dds::DataReader::type (*C++ function*), 397
eprosima::fastdds::dds::DataReader::wait_for_unread (*C++ function*), 397
eprosima::fastdds::dds::DataReaderListener (*C++ class*), 399
eprosima::fastdds::dds::DataReaderListener::~DataRe (*C++ function*), 399
eprosima::fastdds::dds::DataReaderListener::DataRea (*C++ function*), 399
eprosima::fastdds::dds::DataReaderListener::on_data (*C++ function*), 399
eprosima::fastdds::dds::DataReaderListener::on_live (*C++ function*), 400
eprosima::fastdds::dds::DataReaderListener::on_requ (*C++ function*), 400
eprosima::fastdds::dds::DataReaderListener::on_requ (*C++ function*), 400

eprosima::fastdds::dds::DataReaderListener
(C++ function), 400
eprosima::fastdds::dds::DataReaderListener
(C++ function), 400
eprosima::fastdds::dds::DataReaderListener
(C++ function), 399
eprosima::fastdds::dds::DataReaderQos
(C++ class), 400
eprosima::fastdds::dds::DataReaderQos::DataReaderQos
(C++ function), 401
eprosima::fastdds::dds::DataReaderQos::deadline
(C++ function), 401
eprosima::fastdds::dds::DataReaderQos::durability_service
(C++ function), 402
eprosima::fastdds::dds::DataReaderQos::durability
(C++ function), 401
eprosima::fastdds::dds::DataReaderQos::endpoint
(C++ function), 404
eprosima::fastdds::dds::DataReaderQos::expression_filter
(C++ function), 405
eprosima::fastdds::dds::DataReaderQos::history
(C++ function), 402
eprosima::fastdds::dds::DataReaderQos::latency_budget
(C++ function), 401
eprosima::fastdds::dds::DataReaderQos::lifespan
(C++ function), 404
eprosima::fastdds::dds::DataReaderQos::liveliness
(C++ function), 401, 402
eprosima::fastdds::dds::DataReaderQos::ownership
(C++ function), 403
eprosima::fastdds::dds::DataReaderQos::partition
(C++ function), 405
eprosima::fastdds::dds::DataReaderQos::reader_data_lifecycle
(C++ function), 403, 404
eprosima::fastdds::dds::DataReaderQos::reader_resource_limits
(C++ function), 405
eprosima::fastdds::dds::DataReaderQos::reliability
(C++ function), 402
eprosima::fastdds::dds::DataReaderQos::reliable_reader_qos
(C++ function), 404
eprosima::fastdds::dds::DataReaderQos::resource_limits
(C++ function), 402, 403
eprosima::fastdds::dds::DataReaderQos::time_based_filter
(C++ function), 403
eprosima::fastdds::dds::DataReaderQos::type_consistency
(C++ function), 404, 405
eprosima::fastdds::dds::DataReaderQos::user_data
(C++ function), 403
eprosima::fastdds::dds::DataRepresentationQosPolicy
(C++ class), 331
eprosima::fastdds::dds::DataRepresentationQosPolicy
(C++ function), 331
eprosima::fastdds::dds::DataRepresentationQosPolicy
(C++ function), 332
eprosima::fastdds::dds::DataRepresentationQosPolicy
(C++ function), 331
eprosima::fastdds::dds::DataRepresentationQosPolicy::m_dataRepresentation
(C++ member), 332
eprosima::fastdds::dds::DataRepresentationQosPolicy
(C++ function), 331
eprosima::fastdds::dds::DataWriter (C++ class), 380
eprosima::fastdds::dds::DataWriter::assert_liveliness
(C++ function), 383
eprosima::fastdds::dds::DataWriter::clear_history
(C++ function), 383
eprosima::fastdds::dds::DataWriter::dispose
(C++ function), 382
eprosima::fastdds::dds::DataWriter::enable
(C++ function), 380
eprosima::fastdds::dds::DataWriter::get_instance_ha
(C++ function), 381
eprosima::fastdds::dds::DataWriter::get_listener
(C++ function), 382
eprosima::fastdds::dds::DataWriter::get_liveliness_
(C++ function), 383
eprosima::fastdds::dds::DataWriter::get_offered_dea
(C++ function), 381
eprosima::fastdds::dds::DataWriter::get_offered_inc
(C++ function), 381
eprosima::fastdds::dds::DataWriter::get_publisher
(C++ function), 383
eprosima::fastdds::dds::DataWriter::get_qos
(C++ function), 382
eprosima::fastdds::dds::DataWriter::get_topic
(C++ function), 382
eprosima::fastdds::dds::DataWriter::get_type
(C++ function), 381
eprosima::fastdds::dds::DataWriter::guid
(C++ function), 381
eprosima::fastdds::dds::DataWriter::register_instar
(C++ function), 380
eprosima::fastdds::dds::DataWriter::set_listener
(C++ function), 382
eprosima::fastdds::dds::DataWriter::set_qos
(C++ function), 381
eprosima::fastdds::dds::DataWriter::unregister_inst
(C++ function), 380
eprosima::fastdds::dds::DataWriter::wait_for_acknov
(C++ function), 381
eprosima::fastdds::dds::DataWriter::write
(C++ function), 380
eprosima::fastdds::dds::DataWriterListener
(C++ class), 383
eprosima::fastdds::dds::DataWriterListener::~DataWr
(C++ function), 384

eprosima::fastdds::dds::DataWriterListener::onPublicationMatched
(*C++ function*), 384
eprosima::fastdds::dds::DataWriterListener::onLivelinessLost
(*C++ function*), 384
eprosima::fastdds::dds::DataWriterListener::onOfferedDeadlineMissed
(*C++ function*), 384
eprosima::fastdds::dds::DataWriterListener::onOfferedIncompatibleQos
(*C++ function*), 384
eprosima::fastdds::dds::DataWriterListener::publication_matched
(*C++ function*), 384
eprosima::fastdds::dds::DataWriterQos
(*C++ class*), 384
eprosima::fastdds::dds::DataWriterQos::~DataWriterQos
(*C++ function*), 385
eprosima::fastdds::dds::DataWriterQos::DataWriterQos
(*C++ function*), 385
eprosima::fastdds::dds::DataWriterQos::deadline
(*C++ function*), 385
eprosima::fastdds::dds::DataWriterQos::destination_order
(*C++ function*), 386
eprosima::fastdds::dds::DataWriterQos::durability
(*C++ function*), 385
eprosima::fastdds::dds::DataWriterQos::durability_service
(*C++ function*), 385
eprosima::fastdds::dds::DataWriterQos::endpoint
(*C++ function*), 389, 390
eprosima::fastdds::dds::DataWriterQos::history
(*C++ function*), 386, 387
eprosima::fastdds::dds::DataWriterQos::latency_budget
(*C++ function*), 385, 386
eprosima::fastdds::dds::DataWriterQos::lifespan
(*C++ function*), 387
eprosima::fastdds::dds::DataWriterQos::liveliness
(*C++ function*), 386
eprosima::fastdds::dds::DataWriterQos::ownership
(*C++ function*), 388
eprosima::fastdds::dds::DataWriterQos::ownership_strength
(*C++ function*), 388
eprosima::fastdds::dds::DataWriterQos::properties
(*C++ function*), 389
eprosima::fastdds::dds::DataWriterQos::publish_mode
(*C++ function*), 388, 389
eprosima::fastdds::dds::DataWriterQos::reliability
(*C++ function*), 386
eprosima::fastdds::dds::DataWriterQos::reliable_writer_qos
(*C++ function*), 389
eprosima::fastdds::dds::DataWriterQos::representation
(*C++ function*), 389
eprosima::fastdds::dds::DataWriterQos::resource_limits
(*C++ function*), 387
eprosima::fastdds::dds::DataWriterQos::throughput_controller
(*C++ function*), 390
eprosima::fastdds::dds::DataWriterQos::transport_priority
(*C++ function*), 387
eprosima::fastdds::dds::DataWriterQos::user_data
(*C++ function*), 387, 388
eprosima::fastdds::dds::DataWriterQos::writer_data_
(*C++ function*), 388
eprosima::fastdds::dds::DataWriterQos::writer_resou
(*C++ function*), 390
eprosima::fastdds::dds::DeadlineMissedStatus
(*C++ struct*), 357
eprosima::fastdds::dds::DeadlineMissedStatus::~Dead
(*C++ function*), 357
eprosima::fastdds::dds::DeadlineMissedStatus::Deadl
(*C++ function*), 357
eprosima::fastdds::dds::DeadlineMissedStatus::last_
(*C++ member*), 357
eprosima::fastdds::dds::DeadlineMissedStatus::total
(*C++ member*), 357
eprosima::fastdds::dds::DeadlineMissedStatus::total
(*C++ member*), 357
eprosima::fastdds::dds::DeadlineQosPolicy
(*C++ class*), 332
eprosima::fastdds::dds::DeadlineQosPolicy::~Deadlin
(*C++ function*), 332
eprosima::fastdds::dds::DeadlineQosPolicy::clear
(*C++ function*), 332
eprosima::fastdds::dds::DeadlineQosPolicy::Deadline
(*C++ function*), 332
eprosima::fastdds::dds::DeadlineQosPolicy::period
(*C++ member*), 332
eprosima::fastdds::dds::DestinationOrderQosPolicy
(*C++ class*), 332
eprosima::fastdds::dds::DestinationOrderQosPolicy:
(*C++ function*), 333
eprosima::fastdds::dds::DestinationOrderQosPolicy:
(*C++ function*), 333
eprosima::fastdds::dds::DestinationOrderQosPolicy:
(*C++ function*), 333
eprosima::fastdds::dds::DestinationOrderQosPolicy:
(*C++ member*), 333
eprosima::fastdds::dds::DisablePositiveACKsQosPolic
(*C++ class*), 333
eprosima::fastdds::dds::DisablePositiveACKsQosPolic
(*C++ function*), 333
eprosima::fastdds::dds::DisablePositiveACKsQosPolic
(*C++ function*), 333
eprosima::fastdds::dds::DisablePositiveACKsQosPolic
(*C++ function*), 333
eprosima::fastdds::dds::DisablePositiveACKsQosPolic
(*C++ member*), 334
eprosima::fastdds::dds::DisablePositiveACKsQosPolic
(*C++ member*), 334
eprosima::fastdds::dds::DomainEntity
(*C++ class*), 330
eprosima::fastdds::dds::DomainEntity::DomainEntity
(*C++ function*), 331

eprosima::fastdds::dds::DomainParticipant (C++ class), 364
eprosima::fastdds::dds::DomainParticipant::delete_publisher (C++ function), 372
eprosima::fastdds::dds::DomainParticipant::delete_subscriber (C++ function), 368
eprosima::fastdds::dds::DomainParticipant::delete_topic (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::delete_type (C++ function), 365
eprosima::fastdds::dds::DomainParticipant::find_type (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::get_current_time (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::get_default_publisher_qos (C++ function), 369
eprosima::fastdds::dds::DomainParticipant::get_default_subscriber_qos (C++ function), 369, 370
eprosima::fastdds::dds::DomainParticipant::get_default_topic_qos (C++ function), 370
eprosima::fastdds::dds::DomainParticipant::get_domain_id (C++ function), 368
eprosima::fastdds::dds::DomainParticipant::get_instance_handle (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::get_listener (C++ function), 365
eprosima::fastdds::dds::DomainParticipant::get_participant_names (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::get_publisher_qos_from_profile (C++ function), 369
eprosima::fastdds::dds::DomainParticipant::get_qos (C++ function), 365
eprosima::fastdds::dds::DomainParticipant::get_resource_event (C++ function), 372
eprosima::fastdds::dds::DomainParticipant::get_subscriber_qos_from_profile (C++ function), 370
eprosima::fastdds::dds::DomainParticipant::get_topic (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::get_type (C++ function), 372
eprosima::fastdds::dds::DomainParticipant::get_type (C++ function), 372
eprosima::fastdds::dds::DomainParticipant::guid (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::has_active_publisher (C++ function), 373
eprosima::fastdds::dds::DomainParticipant::lookup_topicdescription_from_profile (C++ function), 368
eprosima::fastdds::dds::DomainParticipant::new_remote_subscriber (C++ function), 371
eprosima::fastdds::dds::DomainParticipant::register_publisher_profile_with_profile (C++ function), 372
eprosima::fastdds::dds::DomainParticipant::register_topic (C++ function), 367
eprosima::fastdds::dds::DomainParticipant::set_default_publisher_qos_profile (C++ function), 368
eprosima::fastdds::dds::DomainParticipant::set_default_publisher_qos (C++ function), 369
eprosima::fastdds::dds::DomainParticipant::set_default_subscriber_qos (C++ function), 370
eprosima::fastdds::dds::DomainParticipant::set_default_topic_qos (C++ function), 365
eprosima::fastdds::dds::DomainParticipant::set_listener (C++ function), 365
eprosima::fastdds::dds::DomainParticipant::set_qos (C++ function), 365
eprosima::fastdds::dds::DomainParticipant::unregister_type (C++ function), 367
eprosima::fastdds::dds::DomainParticipantFactory (C++ class), 373
eprosima::fastdds::dds::DomainParticipantFactory::delete_publisher (C++ function), 373
eprosima::fastdds::dds::DomainParticipantFactory::delete_subscriber (C++ function), 373
eprosima::fastdds::dds::DomainParticipantFactory::delete_topic (C++ function), 375
eprosima::fastdds::dds::DomainParticipantFactory::get_domain_id (C++ function), 374
eprosima::fastdds::dds::DomainParticipantFactory::get_instance_handle (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactory::get_listener (C++ function), 375
eprosima::fastdds::dds::DomainParticipantFactory::get_participant_names (C++ function), 375
eprosima::fastdds::dds::DomainParticipantFactory::get_publisher_qos_from_profile (C++ function), 375
eprosima::fastdds::dds::DomainParticipantFactory::get_qos (C++ function), 375
eprosima::fastdds::dds::DomainParticipantFactory::get_resource_event (C++ function), 374
eprosima::fastdds::dds::DomainParticipantFactory::get_subscriber_qos_from_profile (C++ function), 374

eprosima::fastdds::dds::DomainParticipantFactory::set_default_participant_qos (C++ function), 374
eprosima::fastdds::dds::DomainParticipantFactoryQos (C++ class), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::~DomainParticipantFactoryQos (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::kind (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::set_entity_factory (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 376
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 377
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 377
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 377
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 377
eprosima::fastdds::dds::DomainParticipantFactoryQos::operator= (C++ function), 377
eprosima::fastdds::dds::DomainParticipantQos (C++ class), 378
eprosima::fastdds::dds::DomainParticipantQos::DomainParticipantQos (C++ function), 378
eprosima::fastdds::dds::DomainParticipantQos::allocation (C++ function), 378
eprosima::fastdds::dds::DomainParticipantQos::DomainParticipantQos (C++ function), 378
eprosima::fastdds::dds::DomainParticipantQos::entity_factory (C++ function), 378
eprosima::fastdds::dds::DomainParticipantQos::name (C++ function), 379
eprosima::fastdds::dds::DomainParticipantQos::properties (C++ function), 378, 379
eprosima::fastdds::dds::DomainParticipantQos::transport (C++ function), 379
eprosima::fastdds::dds::DomainParticipantQos::user_data (C++ function), 378
eprosima::fastdds::dds::DomainParticipantQos::wire_protocol (C++ function), 379
eprosima::fastdds::dds::DurabilityQosPolicy (C++ class), 334
eprosima::fastdds::dds::DurabilityQosPolicy::~DurabilityQosPolicy (C++ function), 334
eprosima::fastdds::dds::DurabilityQosPolicy::clear (C++ function), 334
eprosima::fastdds::dds::DurabilityQosPolicy::durability (C++ function), 334
eprosima::fastdds::dds::DurabilityQosPolicy::operator= (C++ function), 334
eprosima::fastdds::dds::DurabilityQosPolicy::kind (C++ member), 334
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ class), 335
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ function), 335
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ function), 335
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ function), 335
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ member), 335
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ member), 335
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ member), 335
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ member), 336
eprosima::fastdds::dds::DurabilityServiceQosPolicy (C++ member), 335
eprosima::fastdds::dds::Entity (C++ class), 329
eprosima::fastdds::dds::Entity::close (C++ function), 330
eprosima::fastdds::dds::Entity::enable (C++ function), 330
eprosima::fastdds::dds::Entity::Entity (C++ function), 330
eprosima::fastdds::dds::Entity::get_instance_handle (C++ function), 330
eprosima::fastdds::dds::Entity::get_status_changes (C++ function), 330
eprosima::fastdds::dds::Entity::get_status_mask (C++ function), 330
eprosima::fastdds::dds::Entity::is_enabled (C++ function), 330
eprosima::fastdds::dds::EntityFactoryQosPolicy (C++ class), 336
eprosima::fastdds::dds::EntityFactoryQosPolicy::~EntityFactoryQosPolicy (C++ function), 336
eprosima::fastdds::dds::EntityFactoryQosPolicy::autoenable_created_entities (C++ member), 336
eprosima::fastdds::dds::EntityFactoryQosPolicy::EntityFactoryQosPolicy (C++ function), 336

eprosima::fastdds::dds::GenericDataQosPolicy (C++ class), 336
eprosima::fastdds::dds::GenericDataQosPolicy::clear (C++ function), 337
eprosima::fastdds::dds::GenericDataQosPolicy::data_set (C++ function), 337, 338
eprosima::fastdds::dds::GenericDataQosPolicy::data_size (C++ function), 337
eprosima::fastdds::dds::GenericDataQosPolicy::GenericDataQosPolicy (C++ function), 337
eprosima::fastdds::dds::GenericDataQosPolicy::get_value (C++ function), 338
eprosima::fastdds::dds::GenericDataQosPolicy::operator= (C++ function), 337
eprosima::fastdds::dds::GenericDataQosPolicy::set_data_size (C++ function), 337
eprosima::fastdds::dds::GenericDataQosPolicy::set_value (C++ function), 338
eprosima::fastdds::dds::GroupDataQosPolicy (C++ class), 338
eprosima::fastdds::dds::HistoryQosPolicy (C++ class), 338
eprosima::fastdds::dds::HistoryQosPolicy::~HistoryQosPolicy (C++ function), 339
eprosima::fastdds::dds::HistoryQosPolicy::clear (C++ function), 339
eprosima::fastdds::dds::HistoryQosPolicy::depth (C++ member), 339
eprosima::fastdds::dds::HistoryQosPolicy::HistoryQosPolicy (C++ function), 339
eprosima::fastdds::dds::HistoryQosPolicy::kind (C++ member), 339
eprosima::fastdds::dds::IncompatibleQosStatus (C++ struct), 357
eprosima::fastdds::dds::IncompatibleQosStatus::last_policy_id (C++ member), 358
eprosima::fastdds::dds::IncompatibleQosStatus::policies (C++ member), 358
eprosima::fastdds::dds::IncompatibleQosStatus::total_count (C++ member), 358
eprosima::fastdds::dds::IncompatibleQosStatus::total_count_change (C++ member), 358
eprosima::fastdds::dds::LatencyBudgetQosPolicy (C++ class), 339
eprosima::fastdds::dds::LatencyBudgetQosPolicy::~LatencyBudgetQosPolicy (C++ function), 340
eprosima::fastdds::dds::LatencyBudgetQosPolicy::clear (C++ function), 340
eprosima::fastdds::dds::LatencyBudgetQosPolicy::duration (C++ member), 340
eprosima::fastdds::dds::LatencyBudgetQosPolicy::LatencyBudgetQosPolicy (C++ function), 340
eprosima::fastdds::dds::LifespanQosPolicy (C++ class), 340
eprosima::fastdds::dds::LifespanQosPolicy::~LifespanQosPolicy (C++ function), 340
eprosima::fastdds::dds::LifespanQosPolicy::clear (C++ function), 340
eprosima::fastdds::dds::LifespanQosPolicy::duration (C++ member), 340
eprosima::fastdds::dds::LifespanQosPolicy::LifespanQosPolicy (C++ function), 340
eprosima::fastdds::dds::LivelinessChangedStatus (C++ struct), 358
eprosima::fastdds::dds::LivelinessChangedStatus::alive_count (C++ member), 358
eprosima::fastdds::dds::LivelinessChangedStatus::alive_count_change (C++ member), 358
eprosima::fastdds::dds::LivelinessChangedStatus::last_listener (C++ member), 358
eprosima::fastdds::dds::LivelinessChangedStatus::not_alive_count (C++ member), 358
eprosima::fastdds::dds::LivelinessChangedStatus::not_alive_count_change (C++ member), 358
eprosima::fastdds::dds::LivelinessQosPolicy (C++ class), 340
eprosima::fastdds::dds::LivelinessQosPolicy::~LivelinessQosPolicy (C++ function), 341
eprosima::fastdds::dds::LivelinessQosPolicy::announcement_period (C++ member), 341
eprosima::fastdds::dds::LivelinessQosPolicy::clear (C++ function), 341
eprosima::fastdds::dds::LivelinessQosPolicy::kind (C++ member), 341
eprosima::fastdds::dds::LivelinessQosPolicy::lease_duration (C++ member), 341
eprosima::fastdds::dds::LivelinessQosPolicy::LivelinessQosPolicy (C++ function), 341
eprosima::fastdds::dds::MatchedStatus (C++ struct), 359
eprosima::fastdds::dds::MatchedStatus::~MatchedStatus (C++ function), 359
eprosima::fastdds::dds::MatchedStatus::current_count (C++ member), 359
eprosima::fastdds::dds::MatchedStatus::current_count_change (C++ member), 359
eprosima::fastdds::dds::MatchedStatus::MatchedStatus (C++ function), 359
eprosima::fastdds::dds::MatchedStatus::total_count (C++ member), 359
eprosima::fastdds::dds::MatchedStatus::total_count_change (C++ member), 359
eprosima::fastdds::dds::OwnershipQosPolicy (C++ class), 342
eprosima::fastdds::dds::OwnershipQosPolicy::~OwnershipQosPolicy (C++ function), 342
eprosima::fastdds::dds::OwnershipQosPolicy::clear (C++ function), 342

eprosima::fastdds::dds::OwnershipQosPolicy::kind (C++ member), 342
eprosima::fastdds::dds::OwnershipQosPolicy::OwnershipQosPolicy (C++ function), 342
eprosima::fastdds::dds::OwnershipStrengthQosPolicy (C++ class), 342
eprosima::fastdds::dds::OwnershipStrengthQosPolicy::OwnershipStrengthQosPolicy (C++ function), 343
eprosima::fastdds::dds::OwnershipStrengthQosPolicy::operator= (C++ function), 343
eprosima::fastdds::dds::OwnershipStrengthQosPolicy::value (C++ member), 343
eprosima::fastdds::dds::Partition_t (C++ class), 343
eprosima::fastdds::dds::Partition_t::name (C++ function), 343
eprosima::fastdds::dds::Partition_t::Partition_t (C++ function), 343
eprosima::fastdds::dds::Partition_t::size (C++ function), 343
eprosima::fastdds::dds::PartitionQosPolicy (C++ class), 343
eprosima::fastdds::dds::PartitionQosPolicy::PartitionQosPolicy (C++ function), 344
eprosima::fastdds::dds::PartitionQosPolicy::begin (C++ function), 344
eprosima::fastdds::dds::PartitionQosPolicy::clear (C++ function), 345
eprosima::fastdds::dds::PartitionQosPolicy::operator= (C++ class), 345
eprosima::fastdds::dds::PartitionQosPolicy::push_back (C++ function), 345
eprosima::fastdds::dds::PartitionQosPolicy::empty (C++ function), 344
eprosima::fastdds::dds::PartitionQosPolicy::end (C++ function), 344
eprosima::fastdds::dds::PartitionQosPolicy::getNames (C++ function), 345
eprosima::fastdds::dds::PartitionQosPolicy::max_size (C++ function), 344
eprosima::fastdds::dds::PartitionQosPolicy::names (C++ function), 345
eprosima::fastdds::dds::PartitionQosPolicy::Partition_t (C++ function), 344
eprosima::fastdds::dds::PartitionQosPolicy::size (C++ function), 344
eprosima::fastdds::dds::PresentationQosPolicy (C++ class), 345
eprosima::fastdds::dds::PresentationQosPolicy::~PresentationQosPolicy (C++ function), 346
eprosima::fastdds::dds::PresentationQosPolicy::access_scope (C++ member), 346
eprosima::fastdds::dds::PresentationQosPolicy::clear (C++ function), 346
eprosima::fastdds::dds::PresentationQosPolicy::coherent_access (C++ member), 346
eprosima::fastdds::dds::PresentationQosPolicy::ordered_access (C++ member), 346
eprosima::fastdds::dds::PresentationQosPolicy::PresentationQosPolicy (C++ function), 346
eprosima::fastdds::dds::PublicationMatchedStatus (C++ struct), 359
eprosima::fastdds::dds::PublicationMatchedStatus::last_subscription_handle (C++ member), 360
eprosima::fastdds::dds::Publisher (C++ class), 390
eprosima::fastdds::dds::Publisher::~Publisher (C++ function), 390
eprosima::fastdds::dds::Publisher::create_datawriter (C++ function), 391
eprosima::fastdds::dds::Publisher::create_datawriter (C++ function), 392
eprosima::fastdds::dds::Publisher::delete_datawriter (C++ function), 392
eprosima::fastdds::dds::Publisher::enable (C++ function), 390
eprosima::fastdds::dds::Publisher::get_datawriter_qos (C++ function), 393
eprosima::fastdds::dds::Publisher::get_datawriters (C++ function), 392
eprosima::fastdds::dds::Publisher::get_default_datawriter_qos (C++ function), 393
eprosima::fastdds::dds::Publisher::get_instance_handle (C++ function), 394
eprosima::fastdds::dds::Publisher::get_listener (C++ function), 391
eprosima::fastdds::dds::Publisher::get_participant (C++ function), 393
eprosima::fastdds::dds::Publisher::get_qos (C++ function), 390, 391
eprosima::fastdds::dds::Publisher::has_datawriters (C++ function), 392
eprosima::fastdds::dds::Publisher::lookup_datawriter (C++ function), 392
eprosima::fastdds::dds::Publisher::set_default_datawriter_qos (C++ function), 393
eprosima::fastdds::dds::Publisher::set_listener (C++ function), 391
eprosima::fastdds::dds::Publisher::set_qos (C++ function), 391

eprosima::fastdds::dds::Publisher::wait_for_acknowledgments (C++ function), 392
eprosima::fastdds::dds::PublisherListener (C++ class), 394
eprosima::fastdds::dds::PublisherListener::~PublisherListener (C++ function), 394
eprosima::fastdds::dds::PublisherListener::PublisherListener (C++ function), 394
eprosima::fastdds::dds::PublisherQos (C++ class), 394
eprosima::fastdds::dds::PublisherQos::~PublisherQos (C++ function), 394
eprosima::fastdds::dds::PublisherQos::entity_factory (C++ function), 395
eprosima::fastdds::dds::PublisherQos::group_data (C++ function), 395
eprosima::fastdds::dds::PublisherQos::partition (C++ function), 394, 395
eprosima::fastdds::dds::PublisherQos::presentation (C++ function), 394
eprosima::fastdds::dds::PublisherQos::PublisherQos (C++ function), 394
eprosima::fastdds::dds::PublishModeQosPolicy (C++ class), 347
eprosima::fastdds::dds::PublishModeQosPolicy::~PublishModeQosPolicy (C++ function), 347
eprosima::fastdds::dds::PublishModeQosPolicy::clear (C++ function), 347
eprosima::fastdds::dds::PublishModeQosPolicy::kind (C++ member), 347
eprosima::fastdds::dds::PublishModeQosPolicy::PublishModeQosPolicy (C++ function), 347
eprosima::fastdds::dds::QosPolicy (C++ class), 347
eprosima::fastdds::dds::QosPolicy::~QosPolicy (C++ function), 348
eprosima::fastdds::dds::QosPolicy::clear (C++ function), 348
eprosima::fastdds::dds::QosPolicy::hasChanged (C++ member), 348
eprosima::fastdds::dds::QosPolicy::QosPolicy (C++ function), 348
eprosima::fastdds::dds::QosPolicy::send_always (C++ function), 348
eprosima::fastdds::dds::QosPolicyCount (C++ struct), 360
eprosima::fastdds::dds::QosPolicyCount::count (C++ member), 360
eprosima::fastdds::dds::QosPolicyCount::policy_id (C++ member), 360
eprosima::fastdds::dds::QosPolicyCount::QosPolicyCount (C++ function), 360
eprosima::fastdds::dds::ReaderDataLifecycleQosPolicy (C++ class), 348
eprosima::fastdds::dds::ReaderDataLifecycleQosPolic (C++ function), 348
eprosima::fastdds::dds::ReaderDataLifecycleQosPolic (C++ member), 349
eprosima::fastdds::dds::ReaderDataLifecycleQosPolic (C++ member), 349
eprosima::fastdds::dds::ReaderDataLifecycleQosPolic (C++ function), 348
eprosima::fastdds::dds::ReaderResourceLimitsQos (C++ class), 406
eprosima::fastdds::dds::ReaderResourceLimitsQos::~R (C++ function), 406
eprosima::fastdds::dds::ReaderResourceLimitsQos::ma (C++ member), 406
eprosima::fastdds::dds::ReaderResourceLimitsQos::Re (C++ function), 406
eprosima::fastdds::dds::ReliabilityQosPolicy (C++ class), 349
eprosima::fastdds::dds::ReliabilityQosPolicy::~Reli (C++ function), 349
eprosima::fastdds::dds::ReliabilityQosPolicy::clear (C++ function), 349
eprosima::fastdds::dds::ReliabilityQosPolicy::kind (C++ member), 349
eprosima::fastdds::dds::ReliabilityQosPolicy::max_b (C++ member), 349
eprosima::fastdds::dds::ReliabilityQosPolicy::Relia (C++ function), 349
eprosima::fastdds::dds::ResourceLimitsQosPolicy (C++ class), 350
eprosima::fastdds::dds::ResourceLimitsQosPolicy::~R (C++ function), 350
eprosima::fastdds::dds::ResourceLimitsQosPolicy::al (C++ member), 350
eprosima::fastdds::dds::ResourceLimitsQosPolicy::cl (C++ function), 350
eprosima::fastdds::dds::ResourceLimitsQosPolicy::ma (C++ member), 350
eprosima::fastdds::dds::ResourceLimitsQosPolicy::ma (C++ member), 350
eprosima::fastdds::dds::ResourceLimitsQosPolicy::ma (C++ member), 350
eprosima::fastdds::dds::ResourceLimitsQosPolicy::Re (C++ function), 350
eprosima::fastdds::dds::RTPSEndpointQos (C++ class), 351
eprosima::fastdds::dds::RTPSEndpointQos::entity_id (C++ member), 351
eprosima::fastdds::dds::RTPSEndpointQos::history_me (C++ member), 351
eprosima::fastdds::dds::RTPSEndpointQos::multicast_ (C++ member), 351
eprosima::fastdds::dds::RTPSEndpointQos::remote_loc (C++ member), 351

eprosima::fastdds::dds::RTPSEndpointQos::properties (*C++ member*), 351
eprosima::fastdds::dds::RTPSEndpointQos::property_policy (*C++ member*), 351
eprosima::fastdds::dds::RTPSReliableReaderQos (*C++ class*), 406
eprosima::fastdds::dds::RTPSReliableReaderQos::RTPSReliableReaderQos (*C++ function*), 406
eprosima::fastdds::dds::RTPSReliableReaderQos::disable_positive_ACKs (*C++ member*), 407
eprosima::fastdds::dds::RTPSReliableReaderQos::operator= (*C++ function*), 406
eprosima::fastdds::dds::RTPSReliableReaderQos::times (*C++ member*), 407
eprosima::fastdds::dds::RTPSReliableWriterQos (*C++ class*), 395
eprosima::fastdds::dds::RTPSReliableWriterQos::RTPSReliableWriterQos (*C++ function*), 396
eprosima::fastdds::dds::RTPSReliableWriterQos::disable_positive_ACKs (*C++ member*), 396
eprosima::fastdds::dds::RTPSReliableWriterQos::operator= (*C++ function*), 396
eprosima::fastdds::dds::RTPSReliableWriterQos::times (*C++ member*), 396
eprosima::fastdds::dds::SampleInfo (*C++ struct*), 407
eprosima::fastdds::dds::SampleInfo::absolute_reception_rank (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::disposed_counts (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::generation_rank (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::instance_handle (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::instance_state (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::no_writers_generations (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::publication_handle (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::related_sample_identity (*C++ member*), 408
eprosima::fastdds::dds::SampleInfo::sample_identity (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::sample_rank (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::source_timestamp (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::valid_data (*C++ member*), 407
eprosima::fastdds::dds::SampleInfo::view_state (*C++ member*), 407
eprosima::fastdds::dds::SampleRejectedStatus (*C++ struct*), 361
eprosima::fastdds::dds::SampleRejectedStatus::last_reason (*C++ member*), 361
eprosima::fastdds::dds::SampleRejectedStatus::last_sample_handle (*C++ member*), 361
eprosima::fastdds::dds::SampleRejectedStatus::total_count (*C++ member*), 361
eprosima::fastdds::dds::SampleRejectedStatus::total_count_change (*C++ member*), 361
eprosima::fastdds::dds::StatusMask (*C++ class*), 362
eprosima::fastdds::dds::StatusMask::all (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::data_available (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::data_on_readers (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::inconsistent_topic (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::is_active (*C++ function*), 362
eprosima::fastdds::dds::StatusMask::liveliness_changed (*C++ function*), 364
eprosima::fastdds::dds::StatusMask::liveliness_lost (*C++ function*), 364
eprosima::fastdds::dds::StatusMask::MaskType (*C++ type*), 362
eprosima::fastdds::dds::StatusMask::none (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::offered_deadline_missed (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::offered_incompatible_qos (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::operator>> (*C++ function*), 362
eprosima::fastdds::dds::StatusMask::operator<< (*C++ function*), 362
eprosima::fastdds::dds::StatusMask::publication_matched (*C++ function*), 364
eprosima::fastdds::dds::StatusMask::requested_deadline_missed (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::requested_incompatible_qos (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::sample_lost (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::sample_rejected (*C++ function*), 363
eprosima::fastdds::dds::StatusMask::StatusMask (*C++ function*), 362
eprosima::fastdds::dds::StatusMask::subscription_matched (*C++ function*), 364
eprosima::fastdds::dds::Subscriber (*C++ class*), 408

eprosima::fastdds::dds::Subscriber::~Subscriber (C++ function), 408
eprosima::fastdds::dds::Subscriber::create_data_reader (C++ function), 409
eprosima::fastdds::dds::Subscriber::create_data_reader_with_profile (C++ function), 409
eprosima::fastdds::dds::Subscriber::delete_data_reader (C++ function), 409
eprosima::fastdds::dds::Subscriber::enable (C++ function), 408
eprosima::fastdds::dds::Subscriber::get_datareader_qos_from_profile (C++ function), 411
eprosima::fastdds::dds::Subscriber::get_datareaders (C++ function), 410
eprosima::fastdds::dds::Subscriber::get_default_datareader_qos (C++ function), 411
eprosima::fastdds::dds::Subscriber::get_instance_handle (C++ function), 411
eprosima::fastdds::dds::Subscriber::get_listener (C++ function), 409
eprosima::fastdds::dds::Subscriber::get_participant (C++ function), 411
eprosima::fastdds::dds::Subscriber::get_qos (C++ function), 408
eprosima::fastdds::dds::Subscriber::has_datareaders (C++ function), 410
eprosima::fastdds::dds::Subscriber::lookup_datareader (C++ function), 410
eprosima::fastdds::dds::Subscriber::notify_datareaders (C++ function), 410
eprosima::fastdds::dds::Subscriber::set_default_datareader_qos (C++ function), 410
eprosima::fastdds::dds::Subscriber::set_listener (C++ function), 409
eprosima::fastdds::dds::Subscriber::set_qos (C++ function), 408
eprosima::fastdds::dds::SubscriberListener (C++ class), 412
eprosima::fastdds::dds::SubscriberListener::SubscriberListener (C++ function), 412
eprosima::fastdds::dds::SubscriberListener::on_data_on_arrival (C++ function), 412
eprosima::fastdds::dds::SubscriberListener::on_subscription_matched (C++ function), 412
eprosima::fastdds::dds::SubscriberQos (C++ class), 412
eprosima::fastdds::dds::SubscriberQos::~SubscriberQos (C++ function), 412
eprosima::fastdds::dds::SubscriberQos::empty (C++ function), 413
eprosima::fastdds::dds::SubscriberQos::group_data (C++ function), 413
eprosima::fastdds::dds::SubscriberQos::partition (C++ function), 413
eprosima::fastdds::dds::SubscriberQos::presentation (C++ function), 412
eprosima::fastdds::dds::SubscriberQos::SubscriberQos (C++ function), 412
eprosima::fastdds::dds::SubscriptionMatchedStatus (C++ struct), 364
eprosima::fastdds::dds::SubscriptionMatchedStatus::total_count (C++ member), 364
eprosima::fastdds::dds::TimeBasedFilterQosPolicy (C++ class), 351
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::~TimeBasedFilterQosPolicy (C++ function), 351
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::clear (C++ function), 351
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::minimum_separation (C++ member), 352
eprosima::fastdds::dds::TimeBasedFilterQosPolicy::TimeBasedFilterQosPolicy (C++ function), 351
eprosima::fastdds::dds::Topic (C++ class), 414
eprosima::fastdds::dds::Topic::~Topic (C++ function), 414
eprosima::fastdds::dds::Topic::get_impl (C++ function), 415
eprosima::fastdds::dds::Topic::get_inconsistent_topic (C++ function), 414
eprosima::fastdds::dds::Topic::get_listener (C++ function), 415
eprosima::fastdds::dds::Topic::get_participant (C++ function), 414
eprosima::fastdds::dds::Topic::get_qos (C++ function), 414, 415
eprosima::fastdds::dds::Topic::set_listener (C++ function), 415
eprosima::fastdds::dds::Topic::set_qos (C++ function), 415
eprosima::fastdds::dds::TopicDataQosPolicy (C++ class), 352
eprosima::fastdds::dds::TopicDataType (C++ class), 415
eprosima::fastdds::dds::TopicDataType::~TopicDataType (C++ function), 416
eprosima::fastdds::dds::TopicDataType::auto_fill_type_object (C++ function), 417
eprosima::fastdds::dds::TopicDataType::auto_fill_type_size_for_builtin_types (C++ function), 417
eprosima::fastdds::dds::TopicDataType::createData (C++ function), 416
eprosima::fastdds::dds::TopicDataType::deleteData (C++ function), 416
eprosima::fastdds::dds::TopicDataType::deserialize (C++ function), 416
eprosima::fastdds::dds::TopicDataType::getKey (C++ function), 416

eprosima::fastdds::dds::TopicDataType::getName (C++ function), 417
eprosima::fastdds::dds::TopicDataType::getSerializedSizeProvider (C++ function), 416
eprosima::fastdds::dds::TopicDataType::m_isGetKeyDefined (C++ member), 418
eprosima::fastdds::dds::TopicDataType::m_typeSize (C++ member), 418
eprosima::fastdds::dds::TopicDataType::serialize (C++ function), 416
eprosima::fastdds::dds::TopicDataType::setName (C++ function), 417
eprosima::fastdds::dds::TopicDataType::TopicDataType (C++ function), 416
eprosima::fastdds::dds::TopicDataType::type_identifier (C++ function), 417
eprosima::fastdds::dds::TopicDataType::type_information (C++ function), 418
eprosima::fastdds::dds::TopicDataType::type_object (C++ function), 417, 418
eprosima::fastdds::dds::TopicDescription (C++ class), 418
eprosima::fastdds::dds::TopicDescription::get_impl (C++ function), 418
eprosima::fastdds::dds::TopicDescription::get_name (C++ function), 418
eprosima::fastdds::dds::TopicDescription::get_participant (C++ function), 418
eprosima::fastdds::dds::TopicDescription::get_type_name (C++ function), 418
eprosima::fastdds::dds::TopicListener (C++ class), 419
eprosima::fastdds::dds::TopicListener::~TopicListener (C++ function), 419
eprosima::fastdds::dds::TopicListener::on_inconsistent_topic (C++ function), 419
eprosima::fastdds::dds::TopicListener::TopicListener (C++ function), 419
eprosima::fastdds::dds::TopicQos (C++ class), 419
eprosima::fastdds::dds::TopicQos::deadline (C++ function), 420
eprosima::fastdds::dds::TopicQos::destination_order (C++ function), 421
eprosima::fastdds::dds::TopicQos::durability (C++ function), 419, 420
eprosima::fastdds::dds::TopicQos::durability_service (C++ function), 420
eprosima::fastdds::dds::TopicQos::history (C++ function), 421
eprosima::fastdds::dds::TopicQos::latency_budget (C++ function), 420
eprosima::fastdds::dds::TopicQos::lifespan (C++ function), 422
eprosima::fastdds::dds::TopicQos::liveliness (C++ function), 421
eprosima::fastdds::dds::TopicQos::ownership (C++ function), 422
eprosima::fastdds::dds::TopicQos::reliability (C++ function), 421
eprosima::fastdds::dds::TopicQos::representation (C++ function), 423
eprosima::fastdds::dds::TopicQos::resource_limits (C++ function), 422
eprosima::fastdds::dds::TopicQos::topic_data (C++ function), 419
eprosima::fastdds::dds::TopicQos::TopicQos (C++ function), 419
eprosima::fastdds::dds::TopicQos::transport_priorit (C++ function), 422
eprosima::fastdds::dds::TransportConfigQos (C++ class), 352
eprosima::fastdds::dds::TransportConfigQos::~Transp (C++ function), 352
eprosima::fastdds::dds::TransportConfigQos::clear (C++ function), 352
eprosima::fastdds::dds::TransportConfigQos::listen_ (C++ member), 352
eprosima::fastdds::dds::TransportConfigQos::send_so (C++ member), 352
eprosima::fastdds::dds::TransportConfigQos::Transpo (C++ function), 352
eprosima::fastdds::dds::TransportConfigQos::use_bui (C++ member), 352
eprosima::fastdds::dds::TransportConfigQos::user_tr (C++ member), 352
eprosima::fastdds::dds::TransportPriorityQosPolicy (C++ class), 353
eprosima::fastdds::dds::TransportPriorityQosPolicy (C++ function), 353
eprosima::fastdds::dds::TransportPriorityQosPolicy (C++ function), 353
eprosima::fastdds::dds::TransportPriorityQosPolicy (C++ function), 353
eprosima::fastdds::dds::TransportPriorityQosPolicy (C++ member), 353
eprosima::fastdds::dds::TypeConsistencyEnforcement((C++ class), 353
eprosima::fastdds::dds::TypeConsistencyEnforcement((C++ function), 353
eprosima::fastdds::dds::TypeConsistencyEnforcement((C++ function), 353
eprosima::fastdds::dds::TypeConsistencyEnforcement((C++ member), 354
eprosima::fastdds::dds::TypeConsistencyEnforcement((C++ member), 354
eprosima::fastdds::dds::TypeConsistencyEnforcement((C++ member), 354

eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::m_ignoreSupportingBoundsy (C++ member), 354
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::m_kTypeSupport::get_serialized (C++ member), 354
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::m_preventTypeWidening (C++ member), 354
eprosima::fastdds::dds::TypeConsistencyEnforcementQosPolicy::TypeConsistencyEnforcementQosP (C++ function), 353
eprosima::fastdds::dds::TypeConsistencyQos (C++ class), 413
eprosima::fastdds::dds::TypeConsistencyQos::TypeConsistencyQos (C++ function), 414
eprosima::fastdds::dds::TypeConsistencyQos::clear (C++ function), 414
eprosima::fastdds::dds::TypeConsistencyQos::representation (C++ member), 414
eprosima::fastdds::dds::TypeConsistencyQos::type_consistency (C++ member), 414
eprosima::fastdds::dds::TypeConsistencyQos::TypeConsistencyQos (C++ function), 414
eprosima::fastdds::dds::TypeIdV1 (C++ class), 423
eprosima::fastdds::dds::TypeIdV1::~TypeIdV1 (C++ function), 423
eprosima::fastdds::dds::TypeIdV1::clear (C++ function), 423
eprosima::fastdds::dds::TypeIdV1::get (C++ function), 423
eprosima::fastdds::dds::TypeIdV1::m_type_identifier (C++ member), 424
eprosima::fastdds::dds::TypeIdV1::TypeIdV1 (C++ function), 423
eprosima::fastdds::dds::TypeObjectV1 (C++ class), 425
eprosima::fastdds::dds::TypeObjectV1::~TypeObjectV1 (C++ function), 425
eprosima::fastdds::dds::TypeObjectV1::clear (C++ function), 425
eprosima::fastdds::dds::TypeObjectV1::get (C++ function), 425
eprosima::fastdds::dds::TypeObjectV1::m_type_object (C++ member), 426
eprosima::fastdds::dds::TypeObjectV1::TypeObjectV1 (C++ function), 425
eprosima::fastdds::dds::TypeSupport (C++ class), 426
eprosima::fastdds::dds::TypeSupport::create_data (C++ function), 427
eprosima::fastdds::dds::TypeSupport::delete_data (C++ function), 427
eprosima::fastdds::dds::TypeSupport::deserialize (C++ function), 427
eprosima::fastdds::dds::TypeSupport::empty (C++ function), 428
eprosima::fastdds::dds::TypeSupport::getkey (C++ function), 428
eprosima::fastdds::dds::TypeSupport::get_serialized (C++ function), 427
eprosima::fastdds::dds::TypeSupport::get_type_name (C++ function), 427
eprosima::fastdds::dds::TypeSupport::register_type (C++ function), 426
eprosima::fastdds::dds::TypeSupport::serialize (C++ function), 427
eprosima::fastdds::dds::TypeSupport::TypeSupport (C++ function), 426
eprosima::fastdds::dds::UserDataQosPolicy (C++ class), 355
eprosima::fastdds::dds::WireProtocolConfigQos (C++ class), 355
eprosima::fastdds::dds::WireProtocolConfigQos::~Wi (C++ function), 355
eprosima::fastdds::dds::WireProtocolConfigQos::buil (C++ member), 355
eprosima::fastdds::dds::WireProtocolConfigQos::clea (C++ function), 355
eprosima::fastdds::dds::WireProtocolConfigQos::defa (C++ member), 355
eprosima::fastdds::dds::WireProtocolConfigQos::defa (C++ member), 355
eprosima::fastdds::dds::WireProtocolConfigQos::part (C++ member), 355
eprosima::fastdds::dds::WireProtocolConfigQos::port (C++ member), 355
eprosima::fastdds::dds::WireProtocolConfigQos::pre (C++ member), 355
eprosima::fastdds::dds::WireProtocolConfigQos::thro (C++ member), 355
eprosima::fastdds::dds::WireProtocolConfigQos::Wire (C++ function), 355
eprosima::fastdds::dds::WriterDataLifecycleQosPolic (C++ class), 356
eprosima::fastdds::dds::WriterDataLifecycleQosPolic (C++ function), 356
eprosima::fastdds::dds::WriterDataLifecycleQosPolic (C++ member), 356
eprosima::fastdds::dds::WriterDataLifecycleQosPolic (C++ function), 356
eprosima::fastdds::dds::WriterResourceLimitsQos (C++ class), 356
eprosima::fastdds::dds::WriterResourceLimitsQos::~W (C++ function), 356
eprosima::fastdds::dds::WriterResourceLimitsQos::ma (C++ member), 356
eprosima::fastdds::dds::WriterResourceLimitsQos::Wr (C++ function), 356
eprosima::fastdds::dds::xtypes::TypeInformation (C++ class), 424

eprosima::fastdds::dds::xtypes::TypeInformation::~TypeInformation
(*C++ function*), 424
eprosima::fastdds::dds::xtypes::TypeInformation::assigned
(*C++ function*), 424
eprosima::fastdds::dds::xtypes::TypeInformation::clear
(*C++ function*), 424
eprosima::fastdds::dds::xtypes::TypeInformation::type_information
(*C++ member*), 425
eprosima::fastdds::dds::xtypes::TypeInformation::TypeInformation
(*C++ function*), 424

F

FASTDDS_STATUS_COUNT (*C macro*), 364