FastRTPS Documentation

Release 1.10.0

eProsima

Apr 03, 2020

Installation manual

1 Requirements **3**
- 1.1 Common Dependencies . . . 3
- 1.2 Windows 7 32-bit and 64-bit . . . 3

2 Installation from Binaries **5**
- 2.1 Windows 7 32-bit and 64-bit . . . 5
- 2.2 Linux . . . 5

3 Installation from Sources **7**
- 3.1 Dependencies . . . 7
- 3.2 Colcon installation . . . 7
- 3.3 Manual installation . . . 8
- 3.4 Fast-RTPS-gen . . . 8
- 3.5 Security . . . 8

4 Getting Started **11**
- 4.1 A brief introduction to the RTPS protocol . . . 11
- 4.2 Building your first application . . . 12

5 Library Overview **15**
- 5.1 Fast RTPS architecture . . . 16

6 Objects and Data Structures **17**
- 6.1 Publisher-Subscriber Module . . . 17
- 6.2 RTPS Module . . . 17

7 Publisher-Subscriber Layer **19**
- 7.1 How to use the Publisher-Subscriber Layer . . . 19
- 7.2 Configuration . . . 21
- 7.3 Additional Concepts . . . 33

8 Writer-Reader Layer **35**
- 8.1 Relation to the Publisher-Subscriber Layer . . . 35
- 8.2 How to use the Writer-Reader Layer . . . 35
- 8.3 Configuring Readers and Writers . . . 37
- 8.4 Configuring the History . . . 38

9 Advanced Functionalities **39**

9.1 Topics and Keys 39
9.2 Partitions 40
9.3 Intra-process delivery 43
9.4 Transports 43
9.5 Flow Controllers 57
9.6 Sending large data 58
9.7 Discovery 60
9.8 Subscribing to Discovery Topics 81
9.9 Tuning 83
9.10 Additional Quality of Service options 84
9.11 Logging 84

10 Security **89**
10.1 Authentication plugins 89
10.2 Access control plugins 89
10.3 Cryptographic plugins 90
10.4 Built-in plugins 90
10.5 Example: configuring the `Participant` 99

11 Real-time behavior **105**
11.1 Tuning allocations 105
11.2 Non-blocking calls 109

12 Dynamic Topic Types **111**
12.1 Concepts 111
12.2 Supported Types 112
12.3 Complex examples 118
12.4 Serialization 121
12.5 Important Notes 122
12.6 Dynamic Types Discovery and Endpoint Matching 122
12.7 XML Dynamic Types 124
12.8 Dynamic HelloWorld Examples 124

13 Persistence **127**
13.1 Configuration 127
13.2 Built-in plugins 128

14 XML profiles **131**
14.1 Making an XML 131
14.2 Library settings 132
14.3 Transport descriptors 133
14.4 XML Dynamic Types 136
14.5 Participant profiles 146
14.6 Publisher profiles 153
14.7 Subscriber profiles 155
14.8 Common 157
14.9 Example 163

15 Code generation using fastrtpsgen **175**
15.1 Output 175
15.2 Where to find *fastrtpsgen* 176

16 Typical Use-Cases **177**
16.1 Fast-RTPS over WIFI 177
16.2 Wide Deployments 179

16.3 Fast-RTPS in ROS 2 192

17 Introduction **195**
17.1 Compile 195

18 Execution and IDL Definition **197**
18.1 Building publisher/subscriber code 197
18.2 Defining a data type via IDL 198

19 Version 1.10.0 **207**
19.1 Previous versions 208

eprosima Fast RTPS is a C++ implementation of the RTPS (Real Time Publish-Subscribe) protocol, which provides publisher-subscriber communications over unreliable transports such as UDP, as defined and maintained by the Object Management Group (OMG) consortium. RTPS is also the wire interoperability protocol defined for the Data Distribution Service (DDS) standard, again by the OMG. *eProsima Fast RTPS* holds the benefit of being standalone and up-to-date, as most vendor solutions either implement RTPS as a tool to implement DDS or use past versions of the specification.

Some of the main features of this library are:

- Configurable best-effort and reliable publish-subscribe communication policies for real-time applications.
- Plug and play connectivity so that any new applications are automatically discovered by any other members of the network.
- Modularity and scalability to allow continuous growth with complex and simple devices in the network.
- Configurable network behavior and interchangeable transport layer: Choose the best protocol and system input/output channel combination for each deployment.
- Two API Layers: a high-level Publisher-Subscriber one focused on usability and a lower-level Writer-Reader one that provides finer access to the inner workings of the RTPS protocol.

eProsima Fast RTPS has been adopted by multiple organizations in many sectors including these important cases:

- Robotics: ROS (Robotic Operating System) as their default middleware for ROS2.
- EU R&D: FIWARE Incubated GE.

This documentation is organized into the following sections:

- *Installation manual*
- *User Manual*
- *FastRTPSGen Manual*
- *Release Notes*

CHAPTER 1

Requirements

eProsima Fast RTPS requires the following packages to work.

1.1 Common Dependencies

1.1.1 Gtest

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

1.1.2 Java & Gradle

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

1.2 Windows 7 32-bit and 64-bit

1.2.1 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.

CHAPTER 2

Installation from Binaries

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

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.

2.1.1 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.

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
```

CHAPTER 3

Installation from Sources

3.1 Dependencies

3.1.1 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.

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
```

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*.

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*.

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.

3.5.1 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
```

3.5.2 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
```

CHAPTER 4

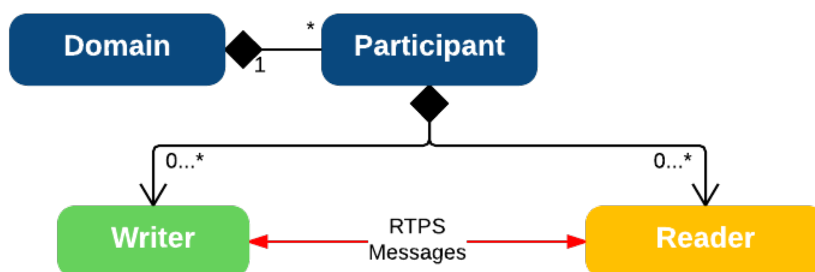
Getting Started

4.1 A brief introduction to the RTPS protocol

At the top of RTPS, we find the Domain, which defines a separate plane of communication. Several domains can coexist at the same time independently. A domain contains any number of Participants, elements capable of sending and receiving data. To do this, the participants use their Endpoints:

- Reader: Endpoint able to receive data.
- Writer: Endpoint able to send data.

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



Communication revolves around Topics, which define the data being exchanged. Topics don't belong to any participant in particular; instead, all interested participants keep track of changes to the topic data and make sure to keep each other up to date. The unit of communication is called a Change, which represents an update to a topic. Endpoints register these changes on their History, a data structure that serves as a cache for recent changes. When you publish a change through a writer endpoint, the following steps happen behind the scenes:

- The change is added to the writer's history cache.
- The writer informs any readers it knows about.
- Any interested (subscribed) readers request the change.

- After receiving data, readers update their history cache with the new change.

By choosing Quality of Service policies, you can affect how these history caches are managed in several ways, but the communication loop remains the same. You can read more information in *Configuration*.

4.2 Building your first application

To build a minimal application, you must first define the topic. To define the data type of the topic Fast-RTPS offers two different approaches, dynamically through *Dynamic Topic Types* and statically through Interface Definition Language (IDL). In this example, we will define the data type statically with IDL, you have more information about IDL in *Introduction*.

Write an IDL file containing the specification you want. In this case, a single string is sufficient.

```
// HelloWorld.idl
struct HelloWorld
{
    string msg;
};
```

Now we need to translate this file to something Fast RTPS understands. For this we have a code generation tool called fastrtpsgen (see *Introduction*), which can do two different things:

- Generate C++ definitions for your custom topic.
- Optionally, generate a working example that uses your topic data.

You may want to check out the fastrtpsgen user manual, which comes with the distribution of the library. But for now, the following commands will do:

On Linux:

```
fastrtpsgen -example CMake HelloWorld.idl
```

On Windows:

```
fastrtpsgen.bat -example CMake HelloWorld.idl
```

The *-example* option creates an example application, and the files needed to build it.

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
```

You may need to set up a special rule in your Firewall for *eprosima Fast RTPS* to work correctly on Windows.

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 modify any values on your custom, IDL-generated data type before sending.

```
HelloWorld sample; //Auto-generated container class for topic data from FastRTPSGen
sample.msg("Hello there!"); // Add contents to the message
publisher->write(&sample); //Publish
```

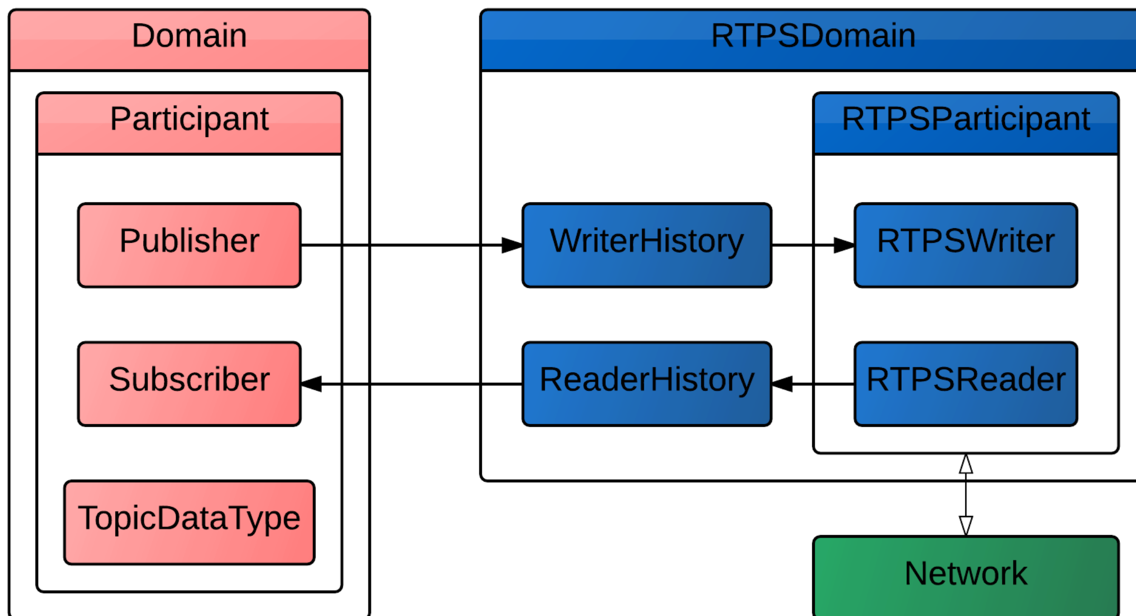
Take a look at the *examples/* folder for ideas on how to improve this basic application through different configuration options, and for examples of advanced Fast RTPS features.

CHAPTER 5

Library Overview

You can interact with Fast RTPS at two different levels:

- Publisher-Subscriber: Simplified abstraction over RTPS.
- Writer-Reader: Direct control over RTPS endpoints.



In red, the Publisher-Subscriber layer offers a convenient abstraction for most use cases. It allows you to define Publishers and Subscribers associated with a topic, and a simple way to transmit topic data. You may remember this from the example we generated in the "Getting Started" section, where we updated our local copy of the topic data, and

called a write() method on it. In blue, the Writer-Reader layer is closer to the concepts defined in the RTPS standard, and allows a finer control, but requires you to interact directly with history caches for each endpoint.

5.1 Fast RTPS architecture

5.1.1 Threads

eProsima Fast RTPS is concurrent and event-based. Each participant 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: the public API is thread safe, so you can fearlessly call any methods on the same participant from different threads. However, it is still useful to know how Fast RTPS schedules work:

- Main thread: Managed by the application.
- Event thread: Each participant owns one of these, and it processes periodic and triggered events.
- Asynchronous writer thread: This thread manages asynchronous writes for all participants. Even for synchronous writers, some forms of communication must be initiated in the background.
- Reception threads: Participants spawn a thread for each reception channel, where the concept of a channel depends on the transport layer (e.g. a UDP port).

5.1.2 Events

There is an event system that enables Fast RTPS to respond to certain conditions, as well as schedule periodic activities. Few of them are visible to the user since most are related to RTPS metadata. However, you can define your own periodic events by inheriting from the TimedEvent class.

CHAPTER 6

Objects and Data Structures

In order to make the most of *eProsima Fast RTPS* it is important to have a grasp of the objects and data structures included in the library. *eProsima Fast RTPS* objects are classified by modules, which are briefly listed and described in this section. For full coverage take a look at the API Reference document that comes with the distribution.

6.1 Publisher-Subscriber Module

This module composes the Publisher-Subscriber abstraction we saw in the Library Overview. The concepts here are higher level than the RTPS standard.

- `Domain` Used to create, manage and destroy high-level Participants.
- `Participant` Contains Publishers and Subscribers, and manages their configuration.
  - `ParticipantAttributes` Configuration parameters used in the creation of a Participant.
  - `ParticipantListener` Allows you to implement callbacks within the scope of the Participant.
- `Publisher` Sends (publishes) data in the form of topic changes.
  - `PublisherAttributes` Configuration parameters for the construction of a Publisher.
  - `PublisherListener` Allows you to implement callbacks within the scope of the Publisher.
- `Subscriber` Receives data for the topics it subscribes to.
  - `SubscriberAttributes` Configuration parameters for the construction of a Subscriber.
  - `SubscriberListener` Allows you to implement callbacks within the scope of the Subscriber.

6.2 RTPS Module

This module directly maps to the ideas defined in the RTPS standard and allows you to interact with RTPS entities directly. It consists of a few sub-modules:

6.2.1 RTPS Common

- `CacheChange_t` Represents a change to a topic, to be stored in a history cache.
- `Data` Payload associated with a cache change. It may be empty depending on the message and change type.
- `Message` Defines the organization of an RTPS Message.
- `Header` Standard header that identifies a message as belonging to the RTPS protocol, and includes the vendor id.
- `Sub-Message Header` Identifier for an RTPS sub-message. An RTPS Message can be composed of several sub-messages.
- `MessageReceiver` Deserializes and processes received RTPS messages.
- `RTPSMessageCreator` Composes RTPS messages.

6.2.2 RTPS Domain

- `RTPSDomain` Use it to create, manage and destroy low-level RTPSParticipants.
- `RTPSParticipant` Contains RTPS Writers and Readers, and manages their configuration.
  - `RTPSParticipantAttributes` Configuration parameters used in the creation of an RTPS Participant.
  - `PDPSimple` Allows the participant to become aware of the other participants within the Network, through the Participant Discovery Protocol.
  - `EDPSimple` Allows the Participant to become aware of the endpoints (RTPS Writers and Readers) present in the other Participants within the network, through the Endpoint Discovery Protocol.
  - `EDPStatic` Reads information about remote endpoints from a user file.
  - `TimedEvent` Base class for periodic or timed events.

6.2.3 RTPS Reader

- `RTPSReader` Base class for the reader endpoint.
  - `ReaderAttributes` Configuration parameters used in the creation of an RTPS Reader.
  - `ReaderHistory` History data structure. Stores recent topic changes.
  - `ReaderListener` Use it to define callbacks in scope of the Reader.

6.2.4 RTPS Writer

- `RTPSWriter` Base class for the writer endpoint.
  - `WriterAttributes` Configuration parameters used in the creation of an RTPS Writer.
  - `WriterHistory` History data structure. Stores outgoing topic changes and schedules them to be sent.

CHAPTER 7

Publisher-Subscriber Layer

eProsima Fast RTPS provides a high-level Publisher-Subscriber Layer, which is a simple to use abstraction over the RTPS protocol. By using this layer, you can code a straight-to-the-point application while letting the library take care of the lower level configuration.

7.1 How to use the Publisher-Subscriber Layer

We are going to use the example built in the previous section to explain how this layer works.

The first step is to create a `Participant` instance, which will act as a container for the Publishers and Subscribers our application needs. For this we use `Domain`, a static class that manages RTPS entities. We also need to pass a configuration structure for the Participant, which can be left in its default configuration for now:

```
ParticipantAttributes participant_attr; //Configuration structure
Participant *participant = Domain::createParticipant(participant_attr);
```

The default configuration provides a basic working set of options with predefined ports for communications. During this tutorial, you will learn to tune *eProsima Fast RTPS*.

In order to use our topic, we have to register it within the `Participant` using the code generated with *fastrtpsgen* (see *Introduction*. Once again, this is done by using the `Domain` class:

```
HelloWorldPubSubType m_type; //Auto-generated type from FastRTPSGen
Domain::registerType(participant, &m_type);
```

Once set up, we instantiate a `Publisher` within our `Participant`:

```
PublisherAttributes publisher_attr; //Configuration structure
PubListener publisher_listener; //Class that implements callbacks from the publisher
Publisher *publisher = Domain::createPublisher(participant, publisher_attr, &
↪publisher_listener);
```

Once the `Publisher` is functional, posting data is a simple process:

```
HelloWorld sample; //Auto-generated container class for topic data from FastRTPSGen
sample.msg("Hello there!"); // Add contents to the message
publisher->write(&sample); //Publish
```

The `Publisher` has a set of optional callback functions that are triggered when events happen. An example is when a `Subscriber` starts listening to our topic.

To implement these callbacks we create the class `PubListener`, which inherits from the base class `PublisherListener`. We pass an instance to this class during the creation of the `Publisher`.

```
class PubListener : public PublisherListener
{
    public:

        PubListener() {}
        ~PubListener() {}

        void onPublicationmatched(Publisher* pub, MatchingInfo& info)
        {
            //Callback implementation. This is called each time the Publisher finds a␣
↪Subscriber on the network that listens to the same topic.
        }
};
```

The `Subscriber` creation and implementation are symmetric.

```
SubscriberAttributes subscriber_attr; //Configuration structure
SubListener subscriber_listener; //Class that implements callbacks from the Subscriber
Subscriber *subscriber = Domain::createSubscriber(participant, subscriber_attr, &
↪subscriber_listener);
```

Incoming messages are processed within the callback that is called when a new message is received:

```
class SubListener: public SubscriberListener
{
    public:

        SubListener() {}

        ~SubListener() {}

        void onNewDataMessage(Subscriber * sub)
        {
            if(sub->takeNextData((void*)&sample, &sample_info))
            {
                if(sample_info.sampleKind == ALIVE)
                {
                    std::cout << "New message: " << sample.msg() << std::endl;
                }
            }
        }

        HelloWorld sample; //Storage for incoming messages

        SampleInfo_t sample_info; //Auxiliary structure with meta-data on the message
};
```

7.2 Configuration

eProsima Fast RTPS entities can be configured through the code or XML profiles. This section will show both alternatives.

7.2.1 Participant configuration

The `Participant` can be configured via the `ParticipantAttributes` structure. `createParticipant` function accepts an instance of this structure.

```
ParticipantAttributes participant_attr;

participant_attr.rtps.setName("my_participant");
participant_attr.rtps.builtin.domainId = 80;

Participant *participant = Domain::createParticipant(participant_attr);
```

Also, it can be configured through an XML profile. `createParticipant` function accepts a name of an XML profile.

```
Participant *participant = Domain::createParticipant("participant_xml_profile");
```

About XML profiles you can learn more in *XML profiles*. This is an example of a participant XML profile.

```
<participant profile_name="participant_xml_conf_profile">
    <rtps>
        <name>my_participant</name>
        <builtin>
            <domainId>80</domainId>
        </builtin>
    </rtps>
</participant>
```

We will now go over the most common configuration options.

- **Participant name:** the name of the `Participant` forms part of the meta-data of the RTPS protocol.

C++

```
participant_attr.rtps.setName("my_participant");
```

XML

```
<participant profile_name="participant_xml_conf_name_profile">
    <rtps>
        <name>my_participant</name>
    </rtps>
</participant>
```

- **DomainId:** Publishers and Subscribers can only talk to each other if their Participants belong to the same DomainId.

C++

```
participant_attr.rtps.builtin.domainId = 80;
```

XML

```
<participant profile_name="participant_xml_conf_domain_profile">
    <rtps>
        <builtin>
            <domainId>80</domainId>
        </builtin>
    </rtps>
</participant>
```

- **Mutation Tries:** The reader's physical port could be already bound. In that case, the Participant uses its *mutation_tries* attribute to determine how many different ports must try before failing. These *mutated* ports will modify the locator's information. By default, its value is *100*.

C++

```
participant_attr.rtps.builtin.mutation_tries = 55;
```

XML

```
<participant profile_name="participant_xml_conf_mutation_tries_profile">
    <rtps>
        <builtin>
            <mutation_tries>55</mutation_tries>
        </builtin>
    </rtps>
</participant>
```

7.2.2 Publisher and Subscriber configuration

The `Publisher` can be configured via the `PublisherAttributes` structure and `createPublisher` function accepts an instance of this structure. The `Subscriber` can be configured via the `SubscriberAttributes` structure and `createSubscriber` function accepts an instance of this structure.

```
PublisherAttributes publisher_attr;
Publisher *publisher = Domain::createPublisher(participant, publisher_attr);

SubscriberAttributes subscriber_attr;
Subscriber *subscriber = Domain::createSubscriber(participant, subscriber_attr);
```

Also, these entities can be configured through an XML profile. `createPublisher` and `createSubscriber` functions accept the name of an XML profile.

```
Publisher *publisher = Domain::createPublisher(participant, "publisher_xml_profile");
Subscriber *subscriber = Domain::createSubscriber(participant, "subscriber_xml_profile
↪");
```

We will now go over the most common configuration options.

Topic information

The topic name and data type are used as meta-data to determine whether Publishers and Subscribers can exchange messages.

C++

```
publisher_attr.topic.topicDataType = "HelloWorldType";
publisher_attr.topic.topicName = "HelloWorldTopic";

subscriber_attr.topic.topicDataType = "HelloWorldType";
subscriber_attr.topic.topicName = "HelloWorldTopic";
```

XML

```
<publisher profile_name="publisher_xml_conf_topic_profile">
    <topic>
        <dataType>HelloWorldType</dataType>
        <name>HelloWorldTopic</name>
    </topic>
</publisher>

<subscriber profile_name="subscriber_xml_conf_topic_profile">
    <topic>
        <dataType>HelloWorldType</dataType>
        <name>HelloWorldTopic</name>
    </topic>
</subscriber>
```

Reliability

The RTPS standard defines two behavior modes for message delivery:

- Best-Effort (default): Messages are sent without arrival confirmation from the receiver (subscriber). It is fast, but messages can be lost.
- Reliable: The sender agent (publisher) expects arrival confirmation from the receiver (subscriber). It is slower but prevents data loss.

C++

```
publisher_attr.qos.m_reliability.kind = RELIABLE_RELIABILITY_QOS;

subscriber_attr.qos.m_reliability.kind = BEST_EFFORT_RELIABILITY_QOS;
```

XML

```
<publisher profile_name="publisher_xml_conf_reliability_profile">
    <qos>
        <reliability>
            <kind>RELIABLE</kind>
        </reliability>
    </qos>
</publisher>

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

Some reliability combinations make a publisher and a subscriber incompatible and unable to talk to each other. Next table shows the incompatibilities.

Publisher \ Subscriber	Best Effort	Reliable
Best Effort	✓	
Reliable	✓	✓

History

There are two policies for sample storage:

- Keep-All: Store all samples in memory.
- Keep-Last (Default): Store samples up to a maximum *depth*. When this limit is reached, they start to become overwritten.

C++

```
publisher_attr.topic.historyQos.kind = KEEP_ALL_HISTORY_QOS;

subscriber_attr.topic.historyQos.kind =   KEEP_LAST_HISTORY_QOS;
subscriber_attr.topic.historyQos.depth = 5;
```

XML

```
<publisher profile_name="publisher_xml_conf_history_profile">
    <topic>
        <historyQos>
            <kind>KEEP_ALL</kind>
        </historyQos>
    </topic>
</publisher>

<subscriber profile_name="subscriber_xml_conf_history_profile">
    <topic>
        <historyQos>
            <kind>KEEP_LAST</kind>
            <depth>5</depth>
        </historyQos>
    </topic>
</subscriber>
```

Durability

Durability configuration of the endpoint defines how it behaves regarding samples that existed on the topic before a subscriber joins

- Volatile: Past samples are ignored, a joining subscriber receives samples generated after the moment it matches.
- Transient Local (Default): When a new subscriber joins, its History is filled with past samples.
- Transient: When a new subscriber joins, its History is filled with past samples, which are stored on persistent storage (see *Persistence*).

C++

```
publisher_attr.qos.m_durability.kind = TRANSIENT_LOCAL_DURABILITY_QOS;

subscriber_attr.qos.m_durability.kind = VOLATILE_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>
```

Deadline

The deadline QoS 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 QoS 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 publishers and subscribers to match, the offered deadline period must be less than or equal to the requested deadline period, otherwise the entities are considered to be incompatible.

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 topic, and use the deadline QoS.

C++

```
publisher_attr.qos.m_deadline.period = 1;

subscriber_attr.qos.m_deadline.period = 1;
```

XML

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

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

Lifespan

Specifies the maximum duration of validity of the data written by the publisher. When the lifespan period expires, data is removed from the history.

C++

```
publisher_attr.qos.m_lifespan.duration = 1;

subscriber_attr.qos.m_lifespan.duration = 1;
```

XML

```
<publisher profile_name="publisher_xml_conf_lifespan_profile">
    <qos>
        <lifespan>
            <duration>
                <sec>1</sec>
            </duration>
        </lifespan>
    </qos>
</publisher>

<subscriber profile_name="subscriber_xml_conf_lifespan_profile">
    <qos>
        <lifespan>
            <duration>
                <sec>1</sec>
            </duration>
        </lifespan>
    </qos>
</subscriber>
```

Liveliness

Liveliness is a quality of service that can be used to ensure that particular entities on the network are "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.

The AUTOMATIC liveliness kind is suitable for applications that only need to detect whether a remote application is still running. Therefore, 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.

The two manual settings require that liveliness is asserted periodically on the publishing side to consider that remote entities are alive. Liveliness can be asserted explicitly by calling the *assert_liveliness* operations on the publisher, or implicitly by writing data. The MANUAL_BY_PARTICIPANT setting only requires that one entity in the publishing side asserts liveliness to deduce that all other entities within that participant are also alive. The MANUAL_BY_TOPIC mode is more restrictive and requires that at least one instance within the publisher is asserted to consider that the publisher is alive.

Besides the liveliness kind, two additional parameters allow defining the application behavior. They are all listed in the table below.

Name	Description	Values	Default
<kind>	Specifies how to manage liveliness.	AUTOMATIC, MANUAL_BY_PARTICIPANT, MANUAL_BY_TOPIC	AUTOMATIC
<lease_duration>	Amount of time to wait since the last message from a writer to consider that it is no longer alive.	*DurationType*	c_TimeInfinite
<announcement_period>	Amount of time between consecutive liveliness messages sent by the publisher. Only used for AUTOMATIC and MANUAL_BY_PARTICIPANT liveliness kinds.	*DurationType*	c_TimeInfinite

C++

```
publisher_attr.qos.m_liveliness.announcement_period = 0.5;
publisher_attr.qos.m_liveliness.lease_duration = 1;
publisher_attr.qos.m_liveliness.kind = AUTOMATIC_LIVELINESS_QOS;

subscriber_attr.qos.m_liveliness.lease_duration = 1;
subscriber_attr.qos.m_liveliness.kind = AUTOMATIC_LIVELINESS_QOS;
```

XML

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

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

Resource limits

Allow controlling the maximum size of the History and other resources.

C++

```
publisher_attr.topic.resourceLimitsQos.max_samples = 200;

subscriber_attr.topic.resourceLimitsQos.max_samples = 200;
```

XML

```
<publisher profile_name="publisher_xml_conf_resource_limits_profile">
    <topic>
        <resourceLimitsQos>
            <max_samples>200</max_samples>
        </resourceLimitsQos>
    </topic>
</publisher>

<subscriber profile_name="subscriber_xml_conf_resource_limits_profile">
    <topic>
        <resourceLimitsQos>
            <max_samples>200</max_samples>
        </resourceLimitsQos>
    </topic>
</subscriber>
```

Disable positive acks

This is an additional QoS that 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 *keep duration* before considering it as acknowledged. A writer and a reader are incompatible (i.e. they will not match) if the latter is using this QoS but the former is not.

C++

```
publisher_attr.qos.m_disablePositiveACKs.enabled = true;
publisher_attr.qos.m_disablePositiveACKs.duration = 1;

subscriber_attr.qos.m_disablePositiveACKs.enabled = true;
```

XML

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

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

Unicast locators

They are network endpoints where the entity will receive data. For more information about the network, see *Transports*. Publishers and subscribers inherit unicast locators from the participant. You can set a different set of locators through this attribute.

C++

```
Locator_t new_locator;
new_locator.port = 7800;

subscriber_attr.unicastLocatorList.push_back(new_locator);

publisher_attr.unicastLocatorList.push_back(new_locator);
```

XML

```
<publisher profile_name="publisher_xml_conf_unicast_locators_profile">
    <unicastLocatorList>
        <locator>
            <udpv4>
                <port>7800</port>
            </udpv4>
        </locator>
    </unicastLocatorList>
</publisher>

<subscriber profile_name="subscriber_xml_conf_unicast_locators_profile">
    <unicastLocatorList>
        <locator>
            <udpv4>
                <port>7800</port>
            </udpv4>
        </locator>
    </unicastLocatorList>
</subscriber>
```

Multicast locators

They are network endpoints where the entity will receive data. For more information about network configuration, see *Transports*. By default publishers and subscribers don't use any multicast locator. This attribute is useful when you have a lot of entities and you want to reduce the network usage.

C++

```cpp
Locator_t new_locator;

IPLocator::setIPv4(new_locator, "239.255.0.4");
new_locator.port = 7900;

subscriber_attr.multicastLocatorList.push_back(new_locator);

publisher_attr.multicastLocatorList.push_back(new_locator);
```

XML

```xml
<publisher profile_name="publisher_xml_conf_multicast_locators_profile">
    <multicastLocatorList>
        <locator>
            <udpv4>
                <address>239.255.0.4</address>
                <port>7900</port>
            </udpv4>
        </locator>
    </multicastLocatorList>
</publisher>

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

7.3 Additional Concepts

7.3.1 Using message meta-data

When a message is taken from the Subscriber, an auxiliary `SampleInfo_t` structure instance is also returned.

Static types

`HelloWorld sample;`
`SampleInfo_t sample_info;`
`subscriber->takeNextData((void*)&sample, &sample_info);`

Dynamic types

`// input_type is an instance of DynamicPubSubType of out current dynamic type`
`DynamicPubSubType *pst = dynamic_cast<DynamicPubSubType*>(input_type);`
`DynamicData *sample = DynamicDataFactory::get_instance()->create_data(pst->`
`↪GetDynamicType());`
`subscriber->takeNextData(sample, &sample_info);`

This `SampleInfo_t` structure contains meta-data on the incoming message:

- *sampleKind*: type of the sample, as defined by the RTPS Standard. Healthy messages from a topic are always ALIVE.
- *WriterGUID*: Signature of the sender (Publisher) the message comes from.
- *OwnershipStrength*: When several senders are writing the same data, this field can be used to determine which data is more reliable.
- *SourceTimestamp*: A timestamp on the sender side that indicates the moment the sample was encapsulated and sent.

This meta-data can be used to implement filters:

```
if( (sample_info.sampleKind == ALIVE) & (sample_info.ownershipStrength > 25) )
{
    //Process data
}
```

7.3.2 Defining callbacks

As we saw in the example, both the `Publisher` and `Subscriber` have a set of callbacks you can use in your application. These callbacks are to be implemented within classes that derive from `SubscriberListener` or `PublisherListener`. The following table gathers information about the possible callbacks that can be implemented in both cases:

Callback	Publisher	Subscriber
onNewDataMessage	N	Y
onSubscriptionMatched	N	Y
onPublicationMatched	Y	N
on_offered_deadline_missed	Y	N
on_requested_deadline_missed	N	Y
on_liveliness_lost	Y	N
on_liveliness_changed	N	Y

CHAPTER 8

Writer-Reader Layer

The lower level Writer-Reader Layer of *eprosima Fast RTPS* provides a raw implementation of the RTPS protocol. It provides more control over the internals of the protocol than the Publisher-Subscriber layer. Advanced users can make use of this layer directly to gain more control over the functionality of the library.

8.1 Relation to the Publisher-Subscriber Layer

Elements of this layer map one-to-one with elements from the Publisher-Subscriber Layer, with a few additions. The following table shows the name correspondence between layers:

Publisher-Subscriber Layer	Writer-Reader Layer
Domain	RTPSDomain
Participant	RTPSParticipant
Publisher	RTPSWriter
Subscriber	RTPSReader

8.2 How to use the Writer-Reader Layer

We will now go over the use of the Writer-Reader Layer like we did with the Publish-Subscriber 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*

8.2.1 Managing the Participant

To create a `RTPSParticipant`, the process is very similar to the one shown in the Publisher-Subscriber layer.

```
RTPSParticipantAttributes participant_attr;
participant_attr.setName("participant");
RTPSParticipant* participant = RTPSDomain::createParticipant(participant_attr);
```

The `RTPSParticipantAttributes` structure is equivalent to the *rtps* member of `ParticipantAttributes` field in the Publisher-Subscriber Layer, so you can configure your `RTPSParticipant` the same way as before:

```
RTPSParticipantAttributes participant_attr;
participant_attr.setName("my_participant");
//etc.
```

8.2.2 Managing the Writers and Readers

As the RTPS standard specifies, Writers and Readers are always associated with a History element. In the Publisher-Subscriber Layer, its creation and management is hidden, but in the Writer-Reader Layer, you have full control over its creation and configuration.

Writers are 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. Note that in this case, you can provide a `ReaderListener` instance that implements your callbacks:

```
class MyReaderListener : public ReaderListener{};
MyReaderListener listener;
HistoryAttributes history_attr;
ReaderHistory* history = new ReaderHistory(history_attr);
ReaderAttributes reader_attr;
RTPSReader* reader = RTPSDomain::createRTPSReader(participant, reader_attr, history, &
↪listener);
```

8.2.3 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 RTPS* 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
```

(continues on next page)

(continued from previous page)

```
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`. *FastRTPSGen* does this for you.

You can receive data from within a `ReaderListener` callback method as we did in the Publisher-Subscriber 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);
        }
};
```

8.3 Configuring Readers and Writers

One of the benefits of using the Writer-Reader layer is that it provides new configuration possibilities while maintaining the options from the Publisher-Subscriber layer (see *Configuration*). 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;
```

8.3.1 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 RTPS* 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 Writer-Reader 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.

8.4 Configuring the History

The History has its own configuration structure, the `HistoryAttributes`.

8.4.1 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
```

8.4.2 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.

CHAPTER 9

Advanced Functionalities

This section covers slightly more advanced, but useful features that enrich your implementation.

9.1 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++

```cpp
// Publisher-Subscriber Layer configuration.
publisher_attr.topic.topicKind = WITH_KEY;
```

XML

```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.

9.2 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.

9.2.1 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**.

9.2.2 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++

```cpp
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

```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>
```

9.3 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>
```

9.4 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, two built-in transports are configured:

- SHM transport will be used for all communications between participants in the same machine.
- UDPv4 will be used for inter machine communications.

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*.

9.4.1 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.

In order to change the default parameters of SHM transport, 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. In both cases `rtps.useBuiltinTransports` must be disabled (see below examples).

C++

```
// Create a descriptor for the new transport.
std::shared_ptr<SharedMemTransportDescriptor> shm_transport = std::make_shared
↪<SharedMemTransportDescriptor>();

// Disable the built-in Transport Layer.
participant_attr.rtps.useBuiltinTransports = false;

// 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>
        <!-- Disable the built-in Transport Layer -->
        <useBuiltinTransports>false</useBuiltinTransports>
    </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.

9.4.2 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++

```cpp
//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

```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++

```
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>
```

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`.

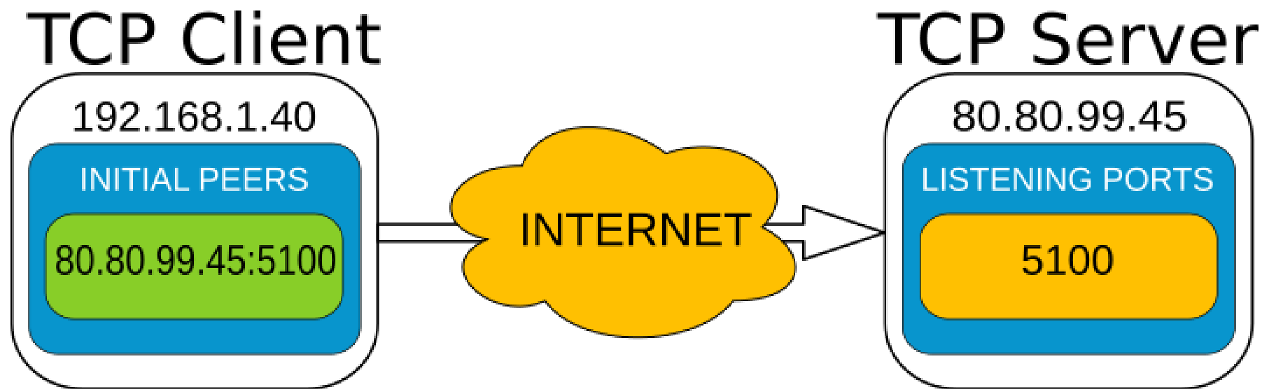
C++

```
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++

```cpp
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

```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*.

9.4.3 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`.

```cpp
// 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.

9.4.4 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>
```

9.4.5 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>
```

9.4.6 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*.

9.5 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

//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.

9.6 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*

9.6.1 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"
```

9.6.2 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.

9.7 Discovery

Fast-RTPS, as a DDS implementation, provides discovery mechanisms that allow for automatically finding and matching publishers and subscribers across participants so they can start sharing data. This discovery is performed, for all the mechanisms, in two phases.

9.7.1 Discovery phases

1. **Participant Discovery Phase (PDP)**: During this phase the participants acknowledge each other's existence. To do that, each participant sends periodic announcement messages, which specify, among other things, unicast addresses (IP and port) where the participant is listening for incoming meta and user data traffic. Two given participants will match when they exist in the same domain. By default, the announcement messages are sent using well-known multicast addresses and ports (calculated using the domain). 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 publishers and subscribers acknowledge each other. To do that, the participants share information about their publishers and subscribers with each other, using the communication channels established during the PDP. This information contains, among other things, the topic and data type. For two endpoints to match, their topic and data type must coincide. Once publisher and subscriber have matched, they are ready for sending/receiving user data traffic.

9.7.2 Discovery mechanisms

Fast-RTPS provides the following discovery mechanisms:

- *Simple Discovery*: This is the default mechanism. It upholds the RTPS standard for both PDP and EDP phases, 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 publishers' and subscribers' addresses and ports, data types, and topics are known beforehand.
- *Server-Client Discovery*: This discovery mechanism uses a centralized discovery architecture, where servers act as a hubs for discovery meta traffic.
- **Manual Discovery**: This mechanism is only compatible with the `RTPSDomain` layer. It disables the PDP discovery phase, letting the user to manually match and unmatch RTPS participants, readers, and writers using whatever, external meta-information channel of its choice.

9.7.3 General discovery settings

Some discovery settings are shared across the different discovery mechanisms. Those 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 participants in the same process, in different processes, or in different hosts	`ParticipantFilteringFlags`	`NO_FILTER`
Lease Duration	Indicates for how much time should a remote participant consider the local participant to be alive.	`Duration_t`	20 s
Announcement Period	The period for the participant 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 participant acts as a hub for discovery traffic, receiving and distributing discovery information.
	`CLIENT`	The participant 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 participant 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` methods.

C++

```
participant_attr.rtps.builtin.discovery_config.discoveryProtocol =
↪DiscoveryProtocol_t::SIMPLE;
```

XML

```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 participant 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 participant's own process is discarded.
FILTER_DIFFERENT_PROCESS | FILTER_SAME_PROCESS	Discovery traffic from participant's own host is discarded.

C++

```
participant_attr.rtps.builtin.discovery_config.ignoreParticipantFlags =
    static_cast<ParticipantFilteringFlags_t>(
        ParticipantFilteringFlags_t::FILTER_DIFFERENT_PROCESS |
        ParticipantFilteringFlags_t::FILTER_SAME_PROCESS);
```

XML

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

Lease Duration

Indicates for how much time should a remote participant consider the local participant to be alive. If the liveliness of the local participant has not being asserted within this time, the remote participant considers the local participant dead and destroys all the information regarding the local participant and all its endpoints.

The local participant's liveliness is asserted on the remote participant any time the remote participant receives any kind of traffic from the local participant.

The lease duration is specified as a time expressed in seconds and nanosecond using a `Duration_t`.

C++

```
participant_attr.rtps.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 participant's PDP announcements. For liveliness' sake it is recommend that the announcement period is shorter than the lease duration, so that the participant'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.

Participant's announcement period is specified as a time expressed in seconds and nanosecond using a `Duration_t`.

C++

```
participant_attr.rtps.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>
```

9.7.4 SIMPLE Discovery Settings

The SIMPLE discovery protocol resolves the establishment of the end-to-end connection between various RTPS entities communicating via the RTPS protocol. Fast-RTPS 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 Participants discover each other in the network; it announces and detects the presence of participants in a domain.
- **Simple Endpoint Discovery Protocol (SEDP):** defines the protocol adopted by the discovered participants for the exchange of information in order to discover the RTPS entities contained in each of them, i.e. the writer and reader Endpoints.

Name	Description
Initial Announcements	It defines the behavior of the RTPSParticipant initial announcements.
Simple EDP Attributes	It defines the use of the SIMPLE protocol as a discovery protocol.
Initial Peers	A list of endpoints to which the SPDP announcements are sent.

Initial Announcements

RTPS standard simple discovery mechanism requires the participant to send announcements. 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++

```
participant_attr.rtps.builtin.discovery_config.initial_announcements.count = 5;
participant_attr.rtps.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 participant may create publishers, subscribers, both or neither.	`bool`	true
Publication writer and Subscription reader	It is intended for participants that implement only one or more publishers, i.e. do not implement subscribers. It allows the creation of only subscriber discovery related EDP endpoints	`bool`	true
Publication reader and Subscription writer	It is intended for participants that implement only one or more subscribers, i.e. do not implement publishers. It allows the creation of only publisher discovery related EDP endpoints.	`bool`	true

C++

```
participant_attr.rtps.builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol␣
↪= true;
participant_attr.rtps.builtin.discovery_config.m_simpleEDP.use_
↪PublicationWriterANDSubscriptionReader = true;
participant_attr.rtps.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

By default, the SPDP protocol uses a well known multicast address for the participant discovery phase. With Fast-RTPS, it is possible to expand the list of endpoints to which the participant announcements are sent by configuring a list of initial peers, as explained in *Initial peers*.

9.7.5 STATIC Endpoints Discovery Settings

Fast-RTPS 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 publishers and subscribers. If all publishers and subscribers, 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 endpoint discovery related settings are:

Name	Description
STATIC EDP	It activates the STATIC endpoint discovery protocol
STATIC EDP XML Files Specification	Specifies an XML file containing a description of the remote endpoints.
Initial Announcements	It defines the behavior of the RTPSParticipant initial announcements (PDP phase).

STATIC EDP

To activate the STATIC EDP, the SEDP must be disabled on the participant attributes. This can be done either by code or using an XML configuration file:

C++

```
participant_attr.rtps.builtin.discovery_config.use_SIMPLE_EndpointDiscoveryProtocol␣
↪= false;
participant_attr.rtps.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 (publishers and subscribers) must be statically specified, which is done using dedicated XML files. A participant may load several of such configuration files so that the information about different endpoints can be contained in one file, or split into different files to keep it more organized. Fast-RTPS provides a Static Endpoint Discovery example that implements this EDP discovery protocol.

The following table describes all the possible attributes 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 endpoint.	uint16_t	0
<entityID>	EntityId of the endpoint.	uint16_t	0
<expectsInlineQos>	It indicates if QOS is expected inline. (reader **only**)	bool	false
<topicName>	Mandatory. The topic of the remote endpoint. Should match with one of the topics of the local participant.	string_255	
<topicDataType>	Mandatory. The data type of the topic.	string_255	
<topicKind>	The kind of topic.	NO_KEY WITH_KEY	NO_KEY
<partitionQos>	The name of a partition of the remote peer. Repeat to configure several partitions.	string	
<unicastLocator>	Unicast locator of the participant. See *Locators definition*.		
<multicastLocator>	Multicast locator of the participant. See *Locators definition*.		
<reliabilityQos>	See the *Reliability* section.	BEST_EFFORT_RELIABILITY_QOS RELIABLE_RELIABILITY_QOS	BEST_EFFORT_RELIABILITY_QOS
<durabilityQos>	See the *Setting the data durability kind* 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 attributes. 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 attributes:

- `kind`: can be one of `SHARED_OWNERSHIP_QOS` or `EXCLUSIVE_OWNERSHIP_QOS`. This attribute is mandatory withing the tag.
- `strength`: an optional `uint32_t` specifying how strongly the remote participant owns the topic. This attribute can be set on writers **only**. If not specified, default value is zero.

Liveliness QoS

The *Liveliness* of the remote peer is configured using `<livelinessQos>` tag. It takes no value, and the configuration is done using tag attributes:

- `kind`: can be any of `AUTOMATIC_LIVELINESS_QOS`, `MANUAL_BY_PARTICIPANT_LIVELINESS_QOS` or `MANUAL_BY_TOPIC_LIVELINESS_QOS`. This attribute 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 participants, a publisher and a subscriber. This configuration **must** agree with the configuration used to create the remote endpoint. Otherwise, communication between endpoints 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 endpoint 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 endpoints **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 entity attributes:

C++

```
SubscriberAttributes sub_attr;
sub_attr.setUserDefinedID(3);

PublisherAttributes pub_attr;
pub_attr.setUserDefinedID(5);
```

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 participant, loading STATIC EDP configuration files is done by:

C++

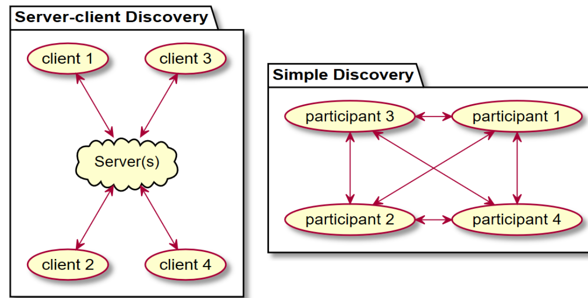
```
participant_attr.rtps.builtin.discovery_config.setStaticEndpointXMLFilename(
↪"RemotePublisher.xml");
participant_attr.rtps.builtin.discovery_config.setStaticEndpointXMLFilename(
↪"RemoteSubscriber.xml");
```

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>
```

9.7.6 Server-Client Discovery

This mechanism is based on a client-server discovery paradigm, i.e. the metatraffic (message exchange among participants to identify each other) is managed by one or several server participants (left figure), as opposed to simple discovery (right figure), where metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol.



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 RTPS writers and readers.
- Discovery server participants 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 publishers and subscribers they create is role-independent.
- All *server* and *client* discovery info will be shared with linked *clients*. will be shared with the *server* or *servers* linked to it. Note that a *server* may act as a *client* for other *servers*.
- *Clients* require a beforehand knowledge of the *servers* they want to link to. Basically it's reduced to the *server* identity (henceforth called `GuidPrefix`) and a list of locators where the *server* is listening. This locators define also the transport protocol (UDP or TCP) the client will use to contact the *server*.
  - The `GuidPrefix` is the RTPS standard participant 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 info the following handshake strategy is followed:

  - *Clients* send hailing messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement.
  - *Servers* receive the hailing messages but they don't start at once to share publishers or subscribers info with the newcomers. They only trigger this process at regular intervals (match period). Tuning this period is possible to bundle the discovery info and deliver it more efficiently.

In order to clarify this discovery setup either on compile time (sources) or runtime (XML files) we are going to split it into two sections: one focusing on the main concepts (*setup by concept*) and the other on the main attribute structures and XML tags (*setup by attribute*).

Server-client setup by concept

Concept	Description
Discovery protocol	how to make a participant a *client* or a *server*.
Server unique id	how to link a *clients* to *servers*.
Seting up transport	how to specify which transport to use and make *servers* reachable.
Pinging period	how to fine tune server-client handshake.
Matching period	how to fine tune server deliver efficiency.

Choosing between client and server

It's set by the *Discovery Protocol* general attribute. 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 attribute and XML tag:

```
ParticipantAttributes.rtps.builtin.discovery_config.discoveryProtocol
```

```
dds>profiles>participant>rtps>builtin>discovery_config>discoveryProtocol
```

C++

```
DiscoverySettings & ds = participant_attr.rtps.builtin.discovery_config;
ds.discoveryProtocol =  DiscoveryProtocol_t::CLIENT;
ds.discoveryProtocol =  DiscoveryProtocol_t::SERVER;
ds.discoveryProtocol =  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 server unique identifier `GuidPrefix`

This belongs to the RTPS specification and univocally identifies each DDS participant. It consists on 12 bytes and is a key in the DDS domain. In the server-client discovery, it has the purpose to link a *server* to its *clients*. Note that there is an auxiliary **ReadguidPrefix** method to populate the `GuidPrefix` using a `string`. It must be mandatorily specified in: *server side* and *client side* setups.

Server side setup

The examples below show how to manage the corresponding enum attribute and XML tag:

```
ParticipantAttributes.rtps.prefix
```

```
dds>profiles>participant>rtps>prefix
```

C++

```
participant_attr.rtps.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");
```

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* it wants to link to. Each single element represents an individual server and a `GuidPrefix` must be provided. The *server* list is the attribute:

```
ParticipantAttributes.rtps.builtin.discovery_config.m_DiscoveryServers
```

and must be populated with `RemoteServerAttributes` objects with a valid `guidPrefix` member. In XML the server list and its elements are simultaneously specified. Note that `prefix` is an attribute of the `RemoteServer` tag.

```
dds>profiles>participant>rtps>builtin>discovery_config>discoveryServerList>
↪RemoteServer@prefix
```

C++

```
RemoteServerAttributes server;
server.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");

DiscoverySettings & ds = participant_attr.rtps.builtin.discovery_config;
ds.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 locator list attribute (note that discovery strategy only deals with metatraffic attributes) and XML tag:

```
ParticipantAttributes.rtps.builtin.
↪(metatrafficMulticastLocatorList|metatrafficUnicastLocatorList)
```

```
dds>profiles>participant>rtps>builtin>
↪(metatrafficMulticastLocatorList|metatrafficUnicastLocatorList)
```

C++

```
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 1 , 133);
locator.port = 64863;

LocatorList_t & ull = participant_attr.rtps.builtin.metatrafficUnicastLocatorList;
ull.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* it wants to link to. Each *server* specifies its own locators. The locator list is the attribute:

```
ParticipantAttributes.rtps.builtin.discovery_config.m_DiscoveryServers
```

and must be populated with `RemoteServerAttributes` objects with a valid `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList` member. In XML the server list and its elements are simultaneously specified. Note the `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList` attributes of the `RemoteServer` tag.

```
dds>profiles>participant>rtps>builtin>discovery_config>discoveryServerList>
↪RemoteServer@metatrafficUnicastLocatorList
dds>profiles>participant>rtps>builtin>discovery_config>discoveryServerList>
↪RemoteServer@metatrafficMulticastLocatorList
```

C++

```
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 1 , 133);
locator.port = 64863;
RemoteServerAttributes server;
server.metatrafficUnicastLocatorList.push_back(locator);

DiscoverySettings & ds = participant_attr.rtps.builtin.discovery_config;
ds.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 hailing messages to the *servers* at regular intervals (ping period) until they receive message reception acknowledgement. This period is specified in the member:

```
ParticipantAttributes.rtps.builtin.discovery_config.discoveryServer_client_syncperiod
```

or the XML tag:

```
dds>profiles>participant>rtps>builtin>discovery_config>clientAnnouncementPeriod
```

C++

```
DiscoverySettings & ds = participant_attr.rtps.builtin.discovery_config;
ds.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* received the hailing messages but they don't start at once to share publishers or subscribers info with the newcomers. They only trigger this process at regular intervals (match period). Note that this member is shared with the *client* setup but its name references solely the *client* functionality. This period is specified in the member:

```
ParticipantAttributes.rtps.builtin.discovery_config.discoveryServer_client_syncperiod
```

or the XML tag:

```
dds>profiles>participant>rtps>builtin>discovery_config>clientAnnouncementPeriod
```

C++

```
DiscoverySettings & ds = participant_attr.rtps.builtin.discovery_config;
ds.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 attribute

The settings related with server-client discovery are:

Name	Description
RTPSParticipantAttributes	Specifies general participant settings. Some of them must be modified in order to properly configure a Server like the `GuidPrefix`.
BuiltinAttributes	It's a member of the above *RTPSParticipantAttributes* structure. Allows to specify some mandatory server discovery settings like the addresses were it listens for clients discovery info.
DiscoverySettings	It's a member of the above *BuiltinAttributes* structure. Allows to specify some mandatory client an optional server settings like the: whether it is a client or a server or the list of servers it is linked to or the client-ping, server-match frequencies.

RTPSParticipantAttributes

A `GuidPrefix_t guidPrefix` member specifies the server's identity. This member has only significance if `discovery_config.discoveryProtocol` is **SERVER** or **BACKUP**. There is a `ReadguidPrefix` method to easily fill in this member from a string formatted like `"4D.49.47.55.45.4c.5f.42.41.52.52.4f"` (note that each byte must be a valid hexadecimal figure).

C++

```
participant_attr.rtps.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");
```

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>
```

BuiltinAttributes

All discovery related info is gathered in a *DiscoverySettings* `discovery_config` member.

In order to receive client metatraffic, `metatrafficUnicastLocatorList` or `metatrafficMulticastLocatorList` must be populated with the addresses that were given to the clients.

C++

```
eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 1 , 133);
locator.port = 64863;

participant_attr.rtps.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

A *discovery_protocol* `discoveryProtocol` member specifies the participant's discovery kind. As was explained before to setup a server-client discovery it may be:

enum value	Description
CLIENT	Generates a client participant, which relies on a server (or servers) to be notified of other clients presence. This participant can create publishers and subscribers of any topic (static or dynamic) as ordinary participants do.
SERVER	Generates a server participant, which receives, manages and spreads its linked client's metatraffic assuring any single one is aware of the others. This participant can create publishers and subscribers of any topic (static or dynamic) as ordinary participants do. Servers can link to other servers in order to share its clients information.
BACKUP	Generates a server participant with additional functionality over **SERVER**. Specifically, it uses a database to backup its client information, so that if for whatever reason it disappears, it can be automatically restored and continue spreading metatraffic to late joiners. A **SERVER** in the same scenario ought to collect client information again, introducing a recovery delay.

A `RemoteServerList_t m_DiscoveryServers` that lists the servers linked to a client participant. 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:

Attribute	Description
`GuidPrefix_t guidPrefix`	Is the RTPS unique identifier of the server participant we want to link to. There is a `ReadguidPrefix` method to easily fill in this member from a string formatted like `"4D.49.47.55.45.4c.5f.42.41.52.52.4f"` (note that each octet must be a valid hexadecimal figure).
`metatrafficUnicastLocatorList` and `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 discoveryServer_client_syncperiod`	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 info with them at regular intervals as a mechanism to bundle discovery info and optimize delivery. The default value is half a second.

C++

```
RemoteServerAttributes server;
server.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");

eprosima::fastrtps::rtps::Locator_t locator;
IPLocator::setIPv4(locator, 192, 168, 1 , 133);
locator.port = 64863;
server.metatrafficUnicastLocatorList.push_back(locator);

DiscoverySettings & ds = participant_attr.rtps.builtin.discovery_config;
ds.discoveryProtocol =  DiscoveryProtocol_t::CLIENT;
ds.m_DiscoveryServers.push_back(server);
ds.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>
```

9.8 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 */
                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 */
                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 */
                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 */
                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.

9.9 Tuning

9.9.1 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 *Multicast locators*.

9.9.2 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 *Reliability* *Fast RTPS* will try to recover lost samples, but with the penalty of retransmission. Using Best-Effort *Reliability* 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
```

9.9.3 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 *History* 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 *History* kind as `KEEP_LAST`.

9.9.4 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*.

9.10 Additional Quality of Service options

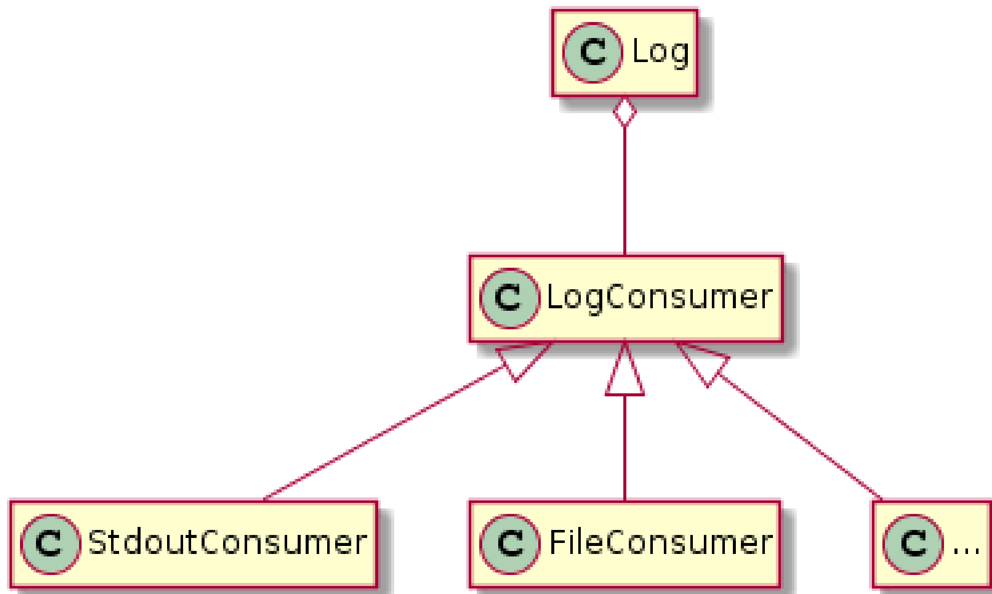
As a user, you can implement your own quality of service (QoS) restrictions in your application. *eProsima Fast RTPS* comes bundled with a set of examples of how to implement common client-wise QoS settings:

- 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.

9.11 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&);
```

9.11.1 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();
```

9.11.2 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>
```

(continues on next page)

(continued from previous page)

```
    <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*.

CHAPTER 10

Security

Fast RTPS can be configured to provide secure communications. For this purpose, Fast RTPS implements pluggable security at three levels: authentication of remote participants, access control of entities and encryption of data.

By default, Fast RTPS doesn't compile security support. You can activate it adding `-DSECURITY=ON` at CMake configuration step. For more information about Fast RTPS compilation, see *Installation from Sources*.

You can activate and configure security plugins through `eprosima::fastrtps::Participant` 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 security plugin.

10.1 Authentication plugins

They provide authentication on the discovery of remote participants. When a remote participant is detected, Fast RTPS tries to authenticate using the activated Authentication plugin. If the authentication process finishes successfully then both participants match and discovery protocol continues. On failure, the remote participant is rejected.

You can activate an Authentication plugin using Participant property `dds.sec.auth.plugin`. Fast RTPS provides a built-in Authentication plugin. More information on *Auth:PKI-DH*.

10.2 Access control plugins

They provide validation of entities' permissions. After a remote participant is authenticated, its permissions need to be validated and enforced.

Access rights that each entity has over a resource are described. Main entity is the Participant and it is used to access or produce information on a Domain; hence the Participant has to be allowed to run in a certain Domain. Also, a Participant is responsible for creating Publishers and Subscribers that communicate over a certain Topic. Hence, a Participant has to have the permissions needed to create a Topic, to publish through its Publishers certain Topics, and to subscribe via its Subscribers to certain Topics. Access control plugin can configure the Cryptographic plugin because its usage is based on the Participant's permissions.

You can activate an Access control plugin using Participant property `dds.sec.access.plugin`. Fast RTPS provides a built-in Access control plugin. More information on *Access:Permissions*.

10.3 Cryptographic plugins

They provide encryption support. Encryption can be applied over three different levels of RTPS protocol. Cryptographic plugins can encrypt whole RTPS messages, RTPS submessages of a particular entity (Writer or Reader) or the payload (user data) of a particular Writer. You can combine them.

You can activate an Cryptographic plugin using Participant property `dds.sec.crypto.plugin`. Fast RTPS provides a built-in Cryptographic plugin. More information on *Crypto:AES-GCM-GMAC*.

The Cryptographic plugin is configured by the Access control plugin. If Access control will not be used, you can configure the Cryptographic plugin manually with the next properties:

Encrypt whole RTPS messages

You can configure a Participant to encrypt all RTPS messages using Participant property `rtps.participant.rtps_protection_kind` with the value `ENCRYPT`.

Encrypt RTPS submessages of a particular entity

You can configure an entity (Writer or Reader) to encrypt its RTPS submessages using Entity property `rtps.endpoint.submessage_protection_kind` with the value `ENCRYPT`.

Encrypt payload of a particular Writer

You can configure a Writer to encrypt its payload using Writer property `rtps.endpoint.payload_protection_kind` with the value `ENCRYPT`.

10.4 Built-in plugins

The current version comes out with three security built-in plugins:

- *Auth:PKI-DH*: this plugin provides authentication using a trusted *Certificate Authority* (CA).
- *Access:Permissions*: this plugin provides access control to Participants at the Domain and Topic level.
- *Crypto:AES-GCM-GMAC*: this plugin provides authenticated encryption using Advanced Encryption Standard (AES) in Galois Counter Mode (AES-GCM).

10.4.1 Auth:PKI-DH

This built-in plugin provides authentication between discovered participants. It is supplied by a trusted *Certificate Authority* (CA) and uses 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 *Crypto:AES-GCM-GMAC*.

You can activate this plugin using Participant property `dds.sec.auth.plugin` with the value `builtin.PKI-DH`. Next tables show you the Participant properties used by this security plugin.

Table 1: **Properties to configure Auth:PKI-DH**

Property name (all properties have "dds.sec.auth.builtin.PKI-DH." prefix)	Property value
identity_ca	URI to the X509 certificate of the Identity CA. Supported URI schemes: file. The **file** schema shall refer to a X.509 v3 certificate in PEM format.
identity_certificate	URI to an X509 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 X509 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.

Generation of x509 certificates

You can generate your own x509 certificates using OpenSSL application. This section teaches you how to do this.

Generate a certificate for the CA

When you want to create your own CA certificate, you first have to write a configuration file with your CA information.

```
# 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.
```

(continues on next page)

(continued from previous page)

```
preserve = no # keep passed DN ordering

policy = policy_match

# For the CA policy
[ policy_match ]
countryName = match
stateOrProvinceName = match
organizationName = match
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName = optional
stateOrProvinceName = optional
localityName = optional
organizationName = optional
organizationalUnitName = optional
commonName = supplied
emailAddress = optional

[ req ]
prompt = no
#default_bits = 1024
#default_keyfile = privkey.pem
distinguished_name= req_distinguished_name
#attributes = req_attributes
#x509_extensions = v3_ca # The extentions to add to the self signed cert
string_mask = utf8only

[ req_distinguished_name ]
countryName = ES
stateOrProvinceName = MA
localityName = Tres Cantos
0.organizationName = eProsima
commonName = eProsima Main Test CA
emailAddress = mainca@eprosima.com
```

After writing the configuration file, next commands generate the certificate using ECDSA.

```
openssl ecparam -name prime256v1 > ecdsaparam

openssl req -nodes -x509 -days 3650 -newkey ec:ecdsaparam -keyout maincakey.pem -out␣
↪maincacert.pem -config maincaconf.cnf
```

Generate a certificate for the Participant

When you want to create your own certificate for your Participant, you first have to write a configuration file.

```
# File: appconf.cnf

prompt = no
string_mask = utf8only
```

(continues on next page)

(continued from previous page)

```
distinguished_name = req_distinguished_name

[ req_distinguished_name ]
countryName = ES
stateOrProvinceName = MA
localityName = Tres Cantos
organizationName = eProsima
emailAddress = example@eprosima.com
commonName = AppName
```

After writing the configuration file, next commands generate the certificate, using ECDSA, for your Participant.

```
openssl ecparam -name prime256v1 > ecdsaparam

openssl req -nodes -new -newkey ec:ecdsaparam -config appconf.cnf -keyout appkey.pem -
↪out appreq.pem

openssl ca -batch -create_serial -config maincaconf.cnf -days 3650 -in appreq.pem -
↪out appcert.pem
```

10.4.2 Access:Permissions

This built-in plugin provides access control using a permissions document signed by a shared *Certificate Authority*. It is configured with three documents:

You can activate this plugin using Participant property `dds.sec.access.plugin` with the value `builtin.Access-Permissions`. Next table shows the Participant properties used by this security plugin.

Table 2: **Properties to configure Access:Permissions**

Property name (all properties have "dds.sec.access.builtin.Access-Permissions." prefix)	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.

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 and Domain Permissions documents.

Domain Governance Document

Domain Governance document is an XML document that specifies how the domain should be secured. It shall be signed by the Permissions CA in S/MIME format.

The format of this document is defined in this Governance XSD file. You can also find a generic Governance XML example.

Domain Rules

Each domain rule is delimited by the `<domain_rule>` XML element tag. Each domain rule contains the following elements and sections:

- Domains element
- Allow Unauthenticated Participants element
- Enable Join Access Control element
- Discovery Protection Kind element
- Liveliness Protection Kind element
- RTPS Protection Kind element
- Topic Access Rules section

The domain rules are evaluated in the same order as they appear in the document. A rule only applies to a particular Participant if the domain section matches the domain to which the Participant belongs. If multiple rules match, the first rule that matches is the only one that applies.

Domains element

This element is delimited by the XML element `<domains>`. The value in this element identifies the collection of Domains values to which the rule applies.

The `<domains>` element can contain a single domain identifier, for example:

```
<domains>
    <id>1</id>
</domains>
```

Or it can contain a range of domain identifiers, for example:

```
<domains>
    <id_range>
        <min>1</min>
        <max>10</max>
    </id_range>
</domains>
```

Or it can contain both, a list of domain identifiers and ranges of domain identifiers.

Allow Unauthenticated Participants element

This element is delimited by the XML element `<allow_unauthenticated_participants>`. Indicates whether the matching of the Participant with a remote Participant requires authentication. If the value is `false`, the Participant shall enforce the authentication of remote Participants and disallow matching those that cannot be successfully authenticated. If the value is `true`, the Participant shall allow matching other Participants (event if the remote Participant cannot authenticate) as long as there is not an already valid authentication with the same Participant's GUID.

Enable Join Access Control element

This element is delimited by the XML element `<enable_join_access_control>`. Indicates whether the matching of the participant with a remote Participant requires authorization by the Access control plugin. If the value is `false`, the Participant shall not check the permissions of the authenticated remote Participant. If the value is `true`, the Participant shall check the permissions of the authenticated remote Participant.

Discovery Protection Kind element

This element is delimited by the XML element `<discovery_protection_kind>`. Indicates whether the secure channel of the endpoint discovery phase needs to be encrypted. If the value is `SIGN` or `ENCRYPT`, the secure channel shall be encrypted. If the value is `NONE`, it shall not.

Liveliness Protection Kind element

This element is delimited by the XML element `<liveliness_protection_kind>`. Indicates whether the secure channel of the liveliness mechanism needs to be encrypted. If the value is `SIGN` or `ENCRYPT`, the secure channel shall be encrypted. If the value is `NONE`, it shall not.

RTPS Protection Kind element

This element is delimited by the XML element `<rtps_protection_kind>`. Indicates whether the whole RTPS Message needs to be encrypted. If the value is `SIGN` or `ENCRYPT`, whole RTPS Messages shall be encrypted. If the value is `NONE`, it shall not.

Topic Rule Section

This element is delimited by the XML element `<topic_rule>` and appears within the Topic Access Rules Section whose XML element is `<topic_access_rules>`.

Each one contains the following elements:

- Topic expression
- Enable Discovery protection
- Enable Liveliness protection
- Enable Read Access Control element
- Enable Write Access Control element
- Metadata protection Kind
- Data protection Kind

The topic expression selects a set of Topic names. The rule applies to any Publisher or Subscriber 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 element

This element is delimited by the XML element `<topic_expression>`. The value in this element identifies the set of Topic names to which the rule applies. The rule will apply to any Publisher and Subscriber 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 *POSIX 1003.2-1992, Section B.6*.

Enable Discovery protection element

This element is delimited by the XML element `<enable_discovery_protection>`. Indicates whether the entity related discovery information shall go through the secure channel of endpoint discovery phase. If the value is `false`, the entity discovery information shall be sent by an unsecured channel of discovery. If the value is `true`, the information shall be sent by the secure channel.

Enable Liveliness Protection element

This element is delimited by the XML element `<enable_liveliness_protection>`. Indicates whether the entity related liveliness information shall go through the secure channel of liveliness mechanism. If the value is `false`, the entity liveliness information shall be sent by an unsecured channel of liveliness. If the value is `true`, the information shall be sent by the secure channel.

Enable Read Access Control element

This element is delimited by the XML element `<enable_read_access_control>`. Indicates whether read access to the Topic is protected. If the value is `false`, then local Subscriber creation and remote Subscriber matching can proceed without further access-control mechanisms imposed. If the value is `true`, they shall be checked using Access control plugin.

Enable Write Access Control element

This element is delimited by the XML element `<enable_write_access_control>`. Indicates whether write access to the Topic is protected. If the value is `false`, then local Publisher creation and remote Publisher matching can proceed without further access-control mechanisms imposed. If the value is `true`, they shall be checked using Access control plugin.

Metadata Protection Kind element

This element is delimited by the XML element `<metadata_protection_kind>`. Indicates whether the entity's RTPS submessages shall be encrypted by the Cryptographic plugin. If the value is `true`, the RTPS submessages shall be encrypted. If the value is `false`, they shall not.

Data Protection Kind element

This element is delimited by the XML element `<data_protection_kind>`. Indicates whether the data payload shall be encrypted by the Cryptographic plugin. If the value is `true`, the data payload shall be encrypted. If the value is `false`, the data payload shall not.

Participant permissions document

The permissions document is an XML document containing the permissions of the Participant and binding them to its distinguished name. The permissions document shall be signed by the Permissions CA in S/MIME format.

The format of this document is defined in this Permissions XSD file. You can also find a generic Permissions XML example.

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 Participant 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 for the Participant as is given in the Authorization 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 Participant. The rules are applied in the same order that appears in the document. If the criteria for the rule matched the Domain join and/or 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 Domain values to which the rule applies. The syntax is the same as for the *Domains element* of the Governance document.

Format of the Allowed/Denied Actions sections

The sections for each of the three action kinds have a similar format. The only difference is the name of the XML element used to delimit the action:

- The Allow/Deny Publish Action is delimited by the `<publish>` XML element.
- The Allow/Deny Subscribe Action is delimited by the `<subscribe>` XML element.
- The Allow/Deny Relay Action is delimited by the `<relay>` XML element.

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 topic name of 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 *POSIX 1003.1-1992, Section B.6*.

```
<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 *POSIX 1003.2-1992, Section B.6*. 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>
```

Signing documents using x509 certificate

Governance document and Permissions document have to be signed by an X509 certificate. Generation of an X509 certificate is explained in *Generation of x509 certificates*. Next commands sign the necessary documents for 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
```

(continues on next page)

(continued from previous page)

10.4.3 Crypto:AES-GCM-GMAC

This built-in plugin provides authenticated encryption using AES in Galois Counter Mode (AES-GCM). It also provides additional reader-specific message authentication codes (MACs) using Galois MAC (AES-GMAC). This plugin needs the activation of the security plugin *Auth:PKI-DH*.

You can activate this plugin using Participant property `dds.sec.crypto.plugin` with the value `builtin.AES-GCM-GMAC`.

10.5 Example: configuring the `Participant`

This example show you how to configure a Participant to activate and configure *Auth:PKI-DH*, *Access:Permissions* and *Crypto:AES-GCM-GMAC* plugins.

Participant attributes

C++

```
eprosima::fastrtps::ParticipantAttributes part_attr;

// Activate Auth:PKI-DH plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.plugin", "builtin.
↪PKI-DH");

// Configure Auth:PKI-DH plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪identity_ca", "file://maincacert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪identity_certificate", "file://appcert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪private_key", "file://appkey.pem");

// Activate Access:Permissions plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.access.plugin",
↪"builtin.Access-Permissions");

// Configure Access:Permissions plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.access.builtin.Access-
↪Permissions.permissions_ca",
    "file://maincacet.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.access.builtin.Access-
↪Permissions.governance",
    "file://governance.smime");
part_attr.rtps.properties.properties().emplace_back("dds.sec.access.builtin.Access-
↪Permissions.permissions",
    "file://permissions.smime");

// Activate Crypto:AES-GCM-GMAC plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.crypto.plugin",
↪"builtin.AES-GCM-GMAC");
```

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>

                <!-- Configure Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_ca</name>
                    <value>file://maincacert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_certificate</name>
                    <value>file://appcert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.private_key</name>
                    <value>file://appkey.pem</value>
                </property>

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

This example shows you how to configure a Participant to activate and configure *Auth:PKI-DH* and *Crypto:AES-GCM-GMAC* plugins, without and Access control plugin. It also configures Participant to encrypt its RTPS messages, Writer and Reader to encrypt their RTPS submessages and a writer to encrypt the payload (user data).

Participant attributes

C++

```
eprosima::fastrtps::ParticipantAttributes part_attr;

// Activate Auth:PKI-DH plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.plugin", "builtin.
↪PKI-DH");

// Configure Auth:PKI-DH plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪identity_ca", "file://maincacert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪identity_certificate", "file://appcert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪private_key", "file://appkey.pem");

// Activate Crypto:AES-GCM-GMAC plugin
part_attr.rtps.properties.properties().emplace_back("dds.sec.crypto.plugin",
↪"builtin.AES-GCM-GMAC");

// Encrypt all RTPS submessages
part_attr.rtps.properties.properties().emplace_back("rtps.participant.rtps_
↪protection_kind", "ENCRYPT");
```

XML

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

                <!-- Configure Auth:PKI-DH plugin -->
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_ca</name>
                    <value>file://maincacert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.identity_certificate</name>
                    <value>file://appcert.pem</value>
                </property>
                <property>
                    <name>dds.sec.auth.builtin.PKI-DH.private_key</name>
                    <value>file://appkey.pem</value>
                </property>

                <!-- Activate Crypto:AES-GCM-GMAC plugin -->
                <property>
                    <name>dds.sec.crypto.plugin</name>
                    <value>builtin.AES-GCM-GMAC</value>
                </property>

                <!-- Encrypt all RTPS submessages -->
                <property>
                    <name>rtps.participant.rtps_protection_kind</name>
                    <value>ENCRYPT</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
```

Publisher attributes

C++

```
eprosima::fastrtps::PublisherAttributes pub_attr;

// Encrypt RTPS submessages
pub_attr.properties.properties().emplace_back("rtps.endpoint.submessage_protection_
↪kind", "ENCRYPT");

// Encrypt payload
pub_attr.properties.properties().emplace_back("rtps.endpoint.payload_protection_kind
↪", "ENCRYPT");
```

XML

```
<publisher profile_name="secure_publisher_xml_profile">
    <propertiesPolicy>
        <properties>
            <!-- Encrypt RTPS submessages -->
            <property>
                <name>rtps.endpoint.submessage_protection_kind</name>
                <value>ENCRYPT</value>
            </property>

            <!-- Encrypt payload -->
            <property>
                <name>rtps.endpoint.payload_protection_kind</name>
                <value>ENCRYPT</value>
            </property>
        </properties>
    </propertiesPolicy>
</publisher>
```

Subscriber attributes

C++

```
eprosima::fastrtps::SubscriberAttributes sub_attr;

// Encrypt RTPS submessages
sub_attr.properties.properties().emplace_back("rtps.endpoint.submessage_protection_
↪kind", "ENCRYPT");
```

XML

```
<subscriber profile_name="secure_publisher_xml_profile">
   <propertiesPolicy>
      <properties>
         <!-- Encrypt RTPS submessages -->
         <property>
            <name>rtps.endpoint.submessage_protection_kind</name>
            <value>ENCRYPT</value>
         </property>
      </properties>
   </propertiesPolicy>
</subscriber>
```

CHAPTER 11

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.

11.1 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.

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 Full example

Given a system with the following topology:

Table 1: **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>
            <max_user_data>256</max_user_data>
            <max_properties>512</max_properties>
        </allocation>
    </rtps>
</participant>
```

11.2 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 2: **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.

CHAPTER 12

Dynamic Topic Types

eProsima Fast RTPS 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 document (DDS-XTypes V1.2) in this link.

The dynamic topic types offer the possibility to work over RTPS without the restrictions related to the IDLs. Using them the users can declare the different types that they need and manage the information directly, avoiding the additional step of updating the IDL file and the generation of *C++* classes.

The management of dynamic types is split into two main groups. The first one manages the declaration of the types, building and setting the configuration of every type and the second one is in charge of the data instances and their information.

12.1 Concepts

Type Descriptor

Stores the information about one type with its relationships and restrictions. It's the minimum class needed to generate a Dynamic type and in case of the complex ones, it stores information about its children or its parent types.

Member Descriptor

Several complex types need member descriptors to declare the relationship between types. This class stores information about that members like their name, their unique ID, the type that is going to be created and the default value after the creation. Union types have special fields to identify each member by labels.

Dynamic Type Builder Factory

Singleton class that is in charge of the creation and the management of every `DynamicTypes` and `DynamicTypeBuilders`. It declares methods to create each kind of supported types, making easier the management of the descriptors. Every object created by the factory must be deleted calling the `delete_type` method.

Dynamic Type Builder

Intermediate class used to configure and create `DynamicTypes`. By design Dynamic types can't be modified, so the previous step to create a new one is to create a builder and apply the settings that the user needs. Users can create

several types using the same builder, but the changes applied to the builder don't affect to the types created previously. Every object created by a builder must be deleted calling the `delete_type` method of the Dynamic Type builder Factory.

Dynamic Type

Base class in the declaration of Dynamic types, it stores the information about its type and every Member that is related to it. It creates a copy of the descriptor on its creation and cannot be changed to keep the consistency.

Dynamic Type Member

A class that creates the relationship between a member descriptor with its parent type. Dynamic Types have a one Dynamic type member for every child member added to it.

Dynamic Data Factory

Singleton class that is in charge of the creation and the management of every `DynamicData`. It creates them using the given `DynamicType` with its settings. Every data object created by the factory must be deleted calling the `delete_type` method. Allows creating a `TypeIdentifier` and a (Minimal and Complete) `TypeObject` from a `TypeDescriptor`.

Dynamic Data

A class that manages the data of the Dynamic Types. It stores the information that is sent and received. There are two ways to work with DynamicDatas, the first one is the most secured, activating the macro `DYNAMIC_TYPES_CHECKING`, it creates a variable for each primitive kind to help the debug process. The second one reduces the size of the `DynamicData` class using only the minimum values and making the code harder to debug.

Dynamic PubSubType

A class that inherits from `TopicDataType` and works as an intermediary between RTPS Domain and the Dynamic Types. It implements the methods needed to create, serialize, deserialize and delete `DynamicData` instances when the participants need to convert the received information from any transport to the registered dynamic type.

12.2 Supported Types

12.2.1 Primitive Types

This section includes every simple kind:

BOOLEAN	INT64
BYTE	UINT16
CHAR8	UINT32
CHAR16	UINT64
INT16	FLOAT32
INT32	FLOAT64
FLOAT128	

Primitive types don't need a specific configuration to create the type. Because of that `DynamicTypeBuilderFactory` has got exposed several methods to allow users to create the Dynamic Types avoiding the `DynamicTypeBuilder` step. The example below shows the two ways to create dynamic data of a primitive type. The `DynamicData` class has a specific `get` and `set` Methods for each primitive type of the list.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_int32_builder();
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicData* data = DynamicDataFactory::get_instance()->create_data(created_type);
data->set_int32_value(1);

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_int32_
↪type();
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pType);
data2->set_int32_value(1);
```

12.2.2 String and WString

Strings are pretty similar to primitive types with one exception, they need to set the size of the `buffer` that they can manage. To do that, `DynamicTypeBuilderFactory` exposes the methods `create_string_type` and `create_wstring_type`. By default, its size is set to 255 characters.

```
// 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");
```

12.2.3 Alias

Alias types have been implemented to rename an existing type, keeping the rest of properties of the given type. `DynamicTypeBuilderFactory` exposes the method `create_alias_type` to create alias types taking the base type and the new name that the alias is going to set. After the creation of the `DynamicData`, users can access its information like they were working with the base type.

```
// Using Builders
DynamicTypeBuilder_ptr base_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_string_builder(100);
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(base_builder.get());
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪alias_builder(created_type, "alias");
DynamicData* data = DynamicDataFactory::get_instance()->create_data(builder.get());
data->set_string_value("Dynamic Alias String");

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_string_
↪type(100);
```

(continues on next page)

(continued from previous page)

```
DynamicType_ptr pAliasType = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪type(pType, "alias");
DynamicData* data2 = DynamicDataFactory::get_instance()->create_data(pAliasType);
data2->set_string_value("Dynamic Alias String");
```

12.2.4 Enumeration

The enum type is managed as complex in Dynamic Types because it allows adding members to set the different values that the enum is going to manage. Internally, it works with a `UINT32` to store what value is selected.

To use enumerations users must create a Dynamic Type builder calling to `create_enum_type` and after that, they can call to `add_member` given the index and the name of the different values that the enum is going to support.

The *DynamicData* class has got methods `get_enum_value` and `set_enum_value` to work with `UINT32` or with strings using the names of the members added to the builder.

```
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");
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_
↪type(builder.get());
DynamicData* data = DynamicDataFactory::get_instance()->create_data(pType);

std::string sValue = "SECOND";
data->set_enum_value(sValue);
uint32_t uValue = 2;
data->set_enum_value(uValue);
```

12.2.5 Bitset

Bitset types are similar to *structure* types but their members are only *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
DynamicTypeBuilder_ptr base_type_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_byte_builder();
```

(continues on next page)

(continued from previous page)

```
auto base_type = base_type_builder->build();

DynamicTypeBuilder_ptr base_type_builder2 = DynamicTypeBuilderFactory::get_instance()-
↪>create_uint32_builder();
auto base_type2 = base_type_builder2->build();

DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪bitset_builder();
builder->add_member(0, "int2", base_type);
builder->add_member(1, "int20", base_type2);
// Apply members' annotations
builder->apply_annotation_to_member(0, ANNOTATION_BIT_BOUND_ID, "value", "2");
builder->apply_annotation_to_member(0, ANNOTATION_POSITION_ID, "value", "0");
builder->apply_annotation_to_member(1, ANNOTATION_BIT_BOUND_ID, "value", "20");
builder->apply_annotation_to_member(1, ANNOTATION_POSITION_ID, "value", "10"); // 8␣
↪bits empty
DynamicType_ptr pType(DynamicTypeBuilderFactory::get_instance()->create_type(builder.
↪get()));
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(pType));
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 method 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());
```

12.2.6 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 methods `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"

// Using Builders
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪bitmask_builder(limit);
builder->add_empty_member(0, "FIRST");
builder->add_empty_member(1, "SECOND");
DynamicType_ptr pType(DynamicTypeBuilderFactory::get_instance()->create_type(builder.
↪get()));
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(pType));
data->set_bool_value(true, 2);
bool bValue;
```

(continues on next page)

(continued from previous page)

```
data->get_bool_value(bValue, 0);
uint64_t fullValue;
data->get_bitmask_value(fullValue);
```

12.2.7 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` methods 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 method `get_member_id_by_name`.

```
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());
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(struct_type));

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 method 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());
```

12.2.8 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_type` method. After the creation of the Dynamic 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.

```
DynamicType_ptr discriminator = DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type();
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪union_builder(discriminator);

builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type(), "", { 0 }, true);
```

(continues on next page)

(continued from previous page)

```
builder->add_member(0, "second", DynamicTypeBuilderFactory::get_instance()->create_
↪int64_type(), "", { 1 }, false);
DynamicType_ptr union_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(union_type));

data->set_int32_value(9, 0);
data->set_int64_value(13, 1);
uint64_t unionLabel;
data->get_union_label(unionLabel);
```

12.2.9 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 methods: - `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 refresh the ids to keep the consistency of the list. - `clear_data`: Removes all the elements of the list.

```
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);
DynamicType_ptr sequence_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(sequence_type));

MemberId newId, newId2;
data->insert_int32_value(10, newId);
data->insert_int32_value(12, newId2);
data->remove_sequence_data(newId);
```

12.2.10 Array

Arrays are pretty similar to sequences with two main differences. The first one is that they can have multiple dimensions and the other one is that they don't need that the elements are stored consecutively. The method to create arrays needs a vector of sizes to set how many dimensions are going to be managed, if users don't want to set a limit can set the value as zero on each dimension and it applies the default value (`100`). To ease the management of arrays every `set` method in `DynamicData` class creates the item if there isn't any in the given `Id`. Arrays also have methods to handle the creation and deletion of elements like sequences, they are `insert_array_data`, `remove_array_data` and `clear_data`. Additionally, there is a special method `get_array_index` that returns the position id giving a vector of indexes on every dimension that the arrays support, that is useful in multidimensional arrays.

```
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);
DynamicType_ptr array_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(array_type));
```

(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);
```

12.2.11 Map

Maps contain a list of pairs 'key-value' 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 add a new element to the map, `DynamicData` has the method `insert_map_data` that returns the ids of the key and the value elements inside the map. To remove an element of the map there is the method `remove_map_data` that uses the given id to find the key element and removes the key and the value elements from the map. The method `clear_data` removes all the elements from the map.

```
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);
DynamicType_ptr map_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(map_type));

DynamicData_ptr key(DynamicDataFactory::get_instance()->create_data(base));
MemberId keyId;
MemberId valueId;
data->insert_map_data(key.get(), keyId, valueId);
MemberId keyId2;
MemberId valueId2;
key->set_int32_value(2);
data->insert_map_data(key.get(), keyId2, valueId2);

data->set_int32_value(53, valueId2);

data->remove_map_data(keyId);
data->remove_map_data(keyId2);
```

12.3 Complex examples

12.3.1 Nested structures

Structures allow to add other structures inside them, but users must take care that to access to these members they need to call `loan_value` to get a pointer to the data and release it calling `return_loaned_value`. `DynamicDatas` manages the counter of loaned values and users can't loan a value that has been loaned previously without calling `return_loaned_value` before.

```
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();

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()));

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);
```

12.3.2 Structures inheritance

Structures can inherit from other structures. To do that `DynamicTypeBuilderFactory` has the method `create_child_struct_type` that relates the given struct type with the new one. The resultant type contains the members of the base class and the ones that users have added to it.

Structures support several levels of inheritance, creating recursively the members of all the types in the hierarchy of the struct.

```
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());

DynamicTypeBuilder_ptr child_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_child_struct_builder(builder.get());
builder->add_member(2, "third", DynamicTypeBuilderFactory::get_instance()->create_
↪uint64_type());

DynamicType_ptr struct_type = child_builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(struct_type));

data->set_int32_value(5, 0);
data->set_uint64_value(13, 1);
data->set_uint64_value(47, 2);
```

12.3.3 Alias of an alias

Alias types support recursion, so if users need to create an alias of another alias, it can be done calling `create_alias_type` method giving the alias as a base type.

```
// Using Builders
DynamicTypeBuilder_ptr created_builder = DynamicTypeBuilderFactory::get_instance()->
↪create_string_builder(100);
DynamicType_ptr created_type = DynamicTypeBuilderFactory::get_instance()->create_
↪type(created_builder.get());
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪alias_builder(created_builder.get(), "alias");
DynamicTypeBuilder_ptr builder2 = DynamicTypeBuilderFactory::get_instance()->create_
↪alias_builder(builder.get(), "alias2");
DynamicData* data(DynamicDataFactory::get_instance()->create_data(builder2->build()));
data->set_string_value("Dynamic Alias 2 String");

// Creating directly the Dynamic Type
DynamicType_ptr pType = DynamicTypeBuilderFactory::get_instance()->create_string_
↪type(100);
DynamicType_ptr pAliasType = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪type(pType, "alias");
DynamicType_ptr pAliasType2 = DynamicTypeBuilderFactory::get_instance()->create_alias_
↪type(pAliasType, "alias2");
DynamicData* data2(DynamicDataFactory::get_instance()->create_data(pAliasType));
data2->set_string_value("Dynamic Alias 2 String");
```

12.3.4 Unions with complex types

Unions support complex types, the available interface to access to them is calling `loan_value` to get a pointer to the data and set this field as the active one and release it calling `return_loaned_value`.

```
DynamicType_ptr discriminator = DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type();
DynamicTypeBuilder_ptr builder = DynamicTypeBuilderFactory::get_instance()->create_
↪union_builder(discriminator);
builder->add_member(0, "first", DynamicTypeBuilderFactory::get_instance()->create_
↪int32_type(), "", { 0 }, true);

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);

DynamicType_ptr union_type = builder->build();
DynamicData_ptr data(DynamicDataFactory::get_instance()->create_data(union_type));

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);
```

12.3.5 Annotations

DynamicTypeBuilder allows applying an annotation to both current type and inner members with the methods:

- `apply_annotation`

- `apply_annotation_to_member`

`apply_annotation_to_member` receives the `MemberId` to apply plus the same parameters than `apply_annotation`. The common parameters are the name of the annotation, the key and the value.

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.

12.4 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(data)());
```

(continues on next page)

(continued from previous page)

```
SerializedPayload_t payload(payloadSize);
pubsubType.serialize(data, &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`.

12.5 Important Notes

The most important part of Dynamic Types is memory management because every dynamic type and dynamic data are managed with pointers. Every object stored inside of other dynamic object is managed by its owner, so users only must take care of the objects that they have created calling to the factories. These two factories in charge to manage these objects, and they must create and delete every object.

```
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 incorporates a special kind of shared pointers to call to the factories to delete the object directly (`DynamicTypeBuilder_ptr` and `DynamicData_ptr`). The only restriction on using this kind of pointers are the methods `loan_value` and `return_loaned_value`, because they return a pointer to an object that is already managed by the library and using a `DynamicData_ptr` with them will cause a crash. `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));
```

12.6 Dynamic Types Discovery and Endpoint Matching

When using Dynamic Types support, *Fast RTPS* make use of an optional `TypeObjectV1` and `TypeIdV1`. At its current state, the matching will only verify that both endpoints are using the same topic type, but will not negotiate about it.

This verification is done by `TypeIdentifier`, then `MinimalTypeObject`, and finally `CompleteTypeObject`.

If one endpoints uses a `CompleteTypeObject` instead, it makes possible *Discovery-Time Data Typing*.

12.6.1 TypeObject (TypeObjectV1)

There are two kinds of `TypeObject`: `MinimalTypeObject` and `CompleteTypeObject`.

- `MinimalTypeObject` is used to check compatibility between types.
- `CompleteTypeObject` fully describes the type.

Both are defined in the annexes of DDS-XTypes V1.2 document so its details will not be covered in this document.

- `TypeObject` is an IDL union with both representation, *Minimal* and *Complete*.

12.6.2 TypeIdentifier (TypeIdV1)

`TypeIdentifier` is described too in the annexes of *DDS-XTypes V1.2 document*. It represents a full description of basic types and has an `EquivalenceKind` for complex ones. An `EquivalenceKind` is a hash code of 14 octets, as described by the *DDS-XTypes V1.2 document*.

12.6.3 TypeObjectFactory

Singleton class that manages the creation and access for all registered `TypeObjects` and `TypeIdentifiers`. From a basic `TypeIdentifier` (in other words, a `TypeIdentifier` whose discriminator isn't `EK_MINIMAL` or `EK_COMPLETE`) can generate a full `DynamicType`.

12.6.4 Fastrtpsgen

FastRTPSGen has been upgraded to generate `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 very easy the use of static types with dynamic types.

12.6.5 Discovery-Time Data Typing

When using fastdds API, if a participant discovers an endpoint which sends a complete TypeObject or a simple TypeIdentifier describing a type that the participant doesn't know, *Fast RTPS* will call to the participant listener's method `on_type_discovery` with the TypeObject and TypeIdentifier provided, and, when possible, a `DynamicType_ptr` 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, there exists an attribute in `TopicAttributes` named `auto_fill_type_object`. If set to true, on discovery time, the local participant will try to fill `type` and `type_id` fields in the correspond `ReaderProxyData` or `WriterProxyData` to be sent to the remote endpoint.

12.6.6 TypeLookup Service

When using fastdds API, if a participant discovers an endpoint which sends a TypeInformation describing a type that the participant doesn't know, *Fast RTPS* will call to the participant listener's method `on_type_information_received` with the TypeInformation provided. Then the user can 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 Participant's RTPS builtin attributes:

```
participant_attr.rtps.builtin.typelookup_config.use_client = true;
participant_attr.rtps.builtin.typelookup_config.use_server = true;
```

A participant can be enabled to act as a TypeLookup server, client, or both.

To ease the recovery and registration of a remote type when received its TypeInformation from the remote endpoint, the DomainParticipant provides the `register_remote_type` function that internally uses the TypeLookup Service.

On success, this function will call a function received as parameter with the signature:

```
void(std::string& type_name, const DynamicType_ptr type)
```

- type_name: Is the given type's name to the `register_remote_type` function, to allow multiple calls using the same
  function.
- type: When possible, the function will be called with an already built DynamicType. If wasn't possible, it will be
  `nullptr`. In that case, the user can try to build the type by himself using the factories, but it is very likely that the build process will fail.

To ease the sharing of the TypeInformation used by the TypeLookup Service, there exists an attribute in `TopicAttributes` named `auto_fill_type_information`. If set to true, on discovery time, the local participant will try to fill `type_information` field in the correspond `ReaderProxyData` or `WriterProxyData` to be sent to the remote endpoint.

12.7 XML Dynamic Types

XML Dynamic Types allows *eProsima Fast RTPS* to create Dynamic Types directly defining them through XML. This allows any application to change `TopicDataTypes` without the need to change its source code.

12.8 Dynamic HelloWorld Examples

12.8.1 DynamicHelloWorldExample

Using some of the functionality described in this document, there exists an example at the `examples/C++/DynamicHelloWorldExample` folder named `DynamicHelloWorldExample` that uses DynamicType generation to provide the TopicDataType.

This example is compatible with classic HelloWorldExample.

As a quick reference, it is shown 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());
```

(continues on next page)

(continued from previous page)

```
// 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);
```

12.8.2 DDSDynamicHelloWorldExample

Another example located in the `examples/C++/DDS/DynamicHelloWorldExample` folder shows a publisher that shares a type loaded from an XML file, and a subscriber that discovers the type using *discovery-time-data-typing*, showing the received data after introspecting it.

12.8.3 TypeLookupService

Finally, an example making use of TypeLookup Service can be found int he `examples/C++/DDS/TypeLookupService` folder. It's very similar to DDSDynamicHelloWorldExample, but the shared type is complex enough to make require the usage of the TypeLookup Service due to the dependency of inner struct types.

CHAPTER 13

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.

13.1 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.

13.2 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.

13.2.1 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 1: **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.

13.2.2 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");

// 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;
```

CHAPTER 14

XML profiles

The *Configuration* section shows how to configure entity attributes using XML profiles, but this section goes deeper on it, 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*.

14.1 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>
```

(continues on next page)

(continued from previous page)

```
</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.

14.1.1 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*.

14.2 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.

14.3 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_send>`	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*.

14.3.1 TLS Configuration

Fast-RTPS allows configuring TLS parameters through the `<tls>` tag of its Transport Descriptor. The full list of options is listed here:

```xml
<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_path>	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

14.4 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.

14.4.1 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 -->
```

(continues on next page)

(continued from previous page)

```
    </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.

14.4.2 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

```xml
<typedef name="MyAliasEnum" type="nonBasic" nonBasicTypeName="MyEnum"/>

<typedef name="MyAliasArray" type="int32" arrayDimension="2,2"/>
```

C++

```cpp
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

```xml
<struct name="MyStruct">
    <member name="first" type="int32"/>
    <member name="second" type="int64"/>
</struct>
```

C++

```cpp
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

```xml
<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>
```

C++

```cpp
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:

<table>
<tr><th>XML</th><th>C++</th></tr>
<tr><td>

```xml
<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>
```

</td><td>

```cpp
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());
```

</td></tr>
</table>

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++

<pre><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></pre>
	<pre>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");</pre>

Bitmask

The `<bitmask>` type is defined by its `name` and inner *bit_values*.

Example:

<table>
<tr><th>XML</th><th>C++</th></tr>
<tr><td>

```
<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>
```

</td><td>

```
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");
```

</td></tr>
</table>

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.

14.4.3 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:

<table>
<tr><th>XML</th><th>C++</th></tr>
<tr><td>

```
<member name="my_long" type="int64"/>
```

</td><td>

```
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());
```

</td></tr>
</table>

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>
```

14.4.4 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");
// Create a "MyStruct" instance
DynamicData* data = DynamicDataFactory::get_instance()->create_data(pbType->
↪GetDynamicType());
```

14.5 Participant profiles

Participant profiles allow declaring *Participant configuration* from an XML file. All the configuration options for the participant 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">
    <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 -->
```

(continues on next page)

(continued from previous page)

```
        </throughputController>

        <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>
```

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*.

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	
<defaultUnicastLocatorList>	List of default input unicast locators. It expects a *LocatorListType*.	LocatorListType	
<defaultMulticastLocatorList>	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*	

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

14.5.1 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>
```

(continues on next page)

(continued from previous page)

```
    <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`	

14.5.2 Built-in parameters

This section of the `Participant's rtps` configuration allows defining built-in parameters.

```
<builtin>
    <discovery_config>
```

(continues on next page)

(continued from previous page)

```
        <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>
            <!-- 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 -
↪->

    <domainId>4</domainId> <!-- uint32 -->

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

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

    <initialPeersList>
        <!-- LOCATOR_LIST -->
```

(continues on next page)

(continued from previous page)

```
        <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_WriterLivelinessProtocol>`	Indicates to use the WriterLiveliness protocol.	`Boolean`	`true`
`<domainId>`	DomainId to be used by the RTPSParticipant.	`UInt32`	0
`<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	5
<period>	The period for the RTPSParticipant to send its first <count> Discovery Messages.	*DurationType*	100 ms

14.6 Publisher profiles

Publisher profiles allow declaring *Publisher configuration* 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.

```xml
<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>
```

(continues on next page)

(continued from previous page)

```
        <!-- LOCATOR_LIST -->
        <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

14.7 Subscriber profiles

Subscriber profiles allow declaring *Subscriber configuration* 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*.

```xml
<subscriber profile_name="sub_profile_name">
    <topic>
        <!-- TOPIC_TYPE -->
    </topic>

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

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

(continues on next page)

(continued from previous page)

```
        <heartbeatResponseDelay> <!-- DURATION -->
            <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

14.8 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.

14.8.1 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>
```

(continues on next page)

(continued from previous page)

```
            <!-- Both physical and logical (port), useful in TCP transports -->
            <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`

14.8.2 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 -->
```

(continues on next page)

(continued from previous page)

```
            <value>Property2Value</value> <!-- string -->
            <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

14.8.3 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

14.8.4 Topic Type

The topic name and data type are used as meta-data to determine whether Publishers and Subscribers can exchange messages. There is a deeper explanation of the "topic" field here: *Topic information*.

```
<topic>
    <kind>NO_KEY</kind> <!-- string -->
    <name>TopicName</name> <!-- string -->
```

(continues on next page)

(continued from previous page)

```
    <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

14.8.5 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.	*Liveliness*	
<reliability>	It is defined in *Reliability* 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 *Deadline* section.	Deadline period as a *DurationType*	c_TimeInfinite
<lifespan>	It is defined in *Lifespan* section.	Lifespan duration as a *DurationType*	c_TimeInfinite
<disablePositiveAcks>	It is defined in section *Disable positive acks*		It is disabled by default and duration is set to c_TimeInfinite

14.8.6 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

14.8.7 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

14.8.8 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.

14.8.9 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.

14.9 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.

```xml
<profiles>
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>ExampleTransportId1</transport_id>
            <type>TCPv4</type>
            <sendBufferSize>8192</sendBufferSize>
            <receiveBufferSize>8192</receiveBufferSize>
            <TTL>250</TTL>
            <maxMessageSize>16384</maxMessageSize>
            <maxInitialPeersRange>100</maxInitialPeersRange>
            <interfaceWhiteList>
                <address>192.168.1.41</address>
                <address>127.0.0.1</address>
            </interfaceWhiteList>
            <wan_addr>80.80.55.44</wan_addr>
            <keep_alive_frequency_ms>5000</keep_alive_frequency_ms>
            <keep_alive_timeout_ms>25000</keep_alive_timeout_ms>
            <max_logical_port>200</max_logical_port>
            <logical_port_range>20</logical_port_range>
            <logical_port_increment>2</logical_port_increment>
            <listening_ports>
                <port>5100</port>
```

(continues on next page)

(continued from previous page)

```
                <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>
                    <caseDiscriminator value="0"/>
                    <caseDiscriminator value="1"/>
                    <member name="first" type="int32"/>
                </case>
                <case>
                    <caseDiscriminator value="2"/>
```

(continues on next page)

(continued from previous page)

```
                    <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">
        <rtps>
            <name>Participant Name</name> <!-- String -->

            <defaultUnicastLocatorList>
                <locator>
                    <udpv4>
```

(continues on next page)

(continued from previous page)

```xml
                <!-- 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>

    <listenSocketBufferSize>8192</listenSocketBufferSize>

    <builtin>
        <discovery_config>

            <discoveryProtocol>NONE</discoveryProtocol>
```

(continues on next page)

(continued from previous page)

```
                    <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>

                <domainId>4</domainId>

                <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>

                <metatrafficMulticastLocatorList>
                    <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>
            </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>

        <port>
            <portBase>7400</portBase>
            <domainIDGain>200</domainIDGain>
            <participantIDGain>10</participantIDGain>
```

(continues on next page)

(continued from previous page)

```
                <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>
        </topic>

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

(continues on next page)

(continued from previous page)

```
            </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>
            </nackSupressionDuration>
        </times>

        <unicastLocatorList>
            <locator>
```

(continues on next page)

(continued from previous page)

```
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </unicastLocatorList>

        <multicastLocatorList>
            <locator>
                <udpv4>
                    <!-- Access as physical, like UDP -->
                    <port>7400</port>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
                    <!-- Both physical and logical (port), like TCP -->
                    <physical_port>5100</physical_port>
                    <port>7400</port>
                    <unique_lan_id>192.168.1.1.1.1.2.55</unique_lan_id>
                    <wan_address>80.80.99.45</wan_address>
                    <address>192.168.1.55</address>
                </tcpv4>
            </locator>
            <locator>
                <udpv6>
                    <port>8844</port>
                    <address>::1</address>
                </udpv6>
            </locator>
        </multicastLocatorList>

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

        <historyMemoryPolicy>DYNAMIC</historyMemoryPolicy>
```

(continues on next page)

(continued from previous page)

```xml
        <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>
            <reliability>
                <kind>BEST_EFFORT</kind>
                <max_blocking_time>
                    <sec>1</sec>
                    <nanosec>856000</nanosec>
```

(continues on next page)

(continued from previous page)

```
                </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>
                    <address>192.168.1.41</address>
                </udpv4>
            </locator>
            <locator>
                <tcpv4>
```

(continues on next page)

(continued from previous page)

```
                <!-- 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>
```

CHAPTER 15

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*.

15.1 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.

15.2 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*.

CHAPTER 16

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.

16.1 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.

16.1.1 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>
```

16.1.2 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>
```

16.2 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.

16.2.1 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, Fast-RTPS 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++

```cpp
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

```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++

```cpp
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

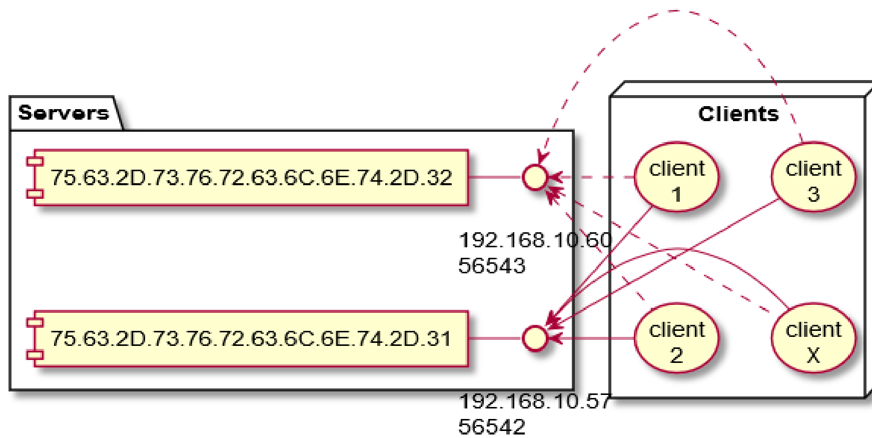
```xml
<participant profile_name="UDP CLIENT" is_default_profile="true">
    <rtps>
        <builtin>
            <discovery_config>
                <discoveryProtocol>CLIENT</discoveryProtocol>
                <discoveryServersList>
                    <RemoteServer prefix="72.61.73.70.66.61.72.6d.74.65.73.74">
                        <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.31	192.168.10.57:56542
75.63.2D.73.76.72.63.6C.6E.74.2D.32	192.168.10.60:56543



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++

```cpp
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);
```

CLIENT

C++

```cpp
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);
```

SERVER

XML

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

CLIENT

XML

```xml
<participant profile_name="UDP CLIENT">
    <rtps>
        <builtin>
            <discovery_config>
                <discoveryProtocol>CLIENT</discoveryProtocol>
                <discoveryServersList>
                    <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.31">
                        <metatrafficUnicastLocatorList>
                            <locator>
                                <udpv4>
                                    <address>192.168.10.57</address>
                                    <port>56542</port>
                                </udpv4>
                            </locator>
                        </metatrafficUnicastLocatorList>
                    </RemoteServer>
                    <RemoteServer prefix="75.63.2D.73.76.72.63.6C.6E.74.2D.32">
```

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.

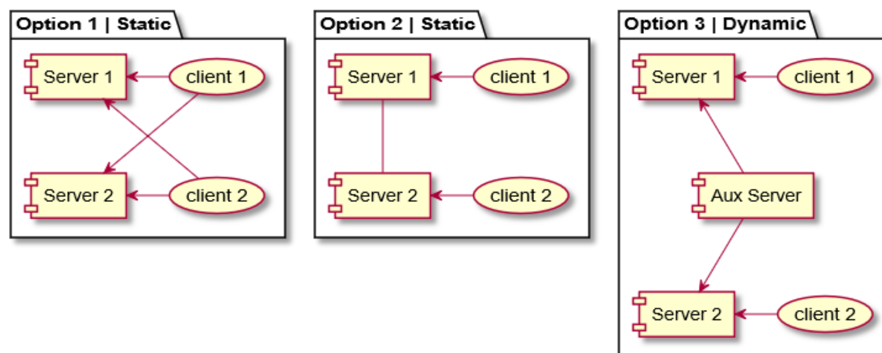
In order to start afresh, that is without deserialize any discovery info, the old backup file must be removed or renamed before launching the server.

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.73.76.72.63.6C.6E.74.2D.41	192.168.10.60:56543
B	75.63.2D.73.76.72.63.6C.6E.74.2D.42	192.168.10.57:56542

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.73.76.72.63.6C.6E.74.2D.31	192.168.10.60:56543
B	75.63.2D.73.76.72.63.6C.6E.74.2D.32	192.168.10.57:56542
C	75.63.2D.73.76.72.63.6C.6E.74.2D.33	192.168.10.54:56541

In order to communicate both networks we can setup server C to act as client of servers A and B as follows:

C++

```
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>
```

16.2.2 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 Endpoints 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

<table>
<tr><th>HelloWorldPublisher.xml</th><th>HelloWorldSubscriber.xml</th></tr>
<tr><td>

```xml
<staticdiscovery>
    <participant>
        <name>HelloWorldPublisher</
→name>
        <writer>
            <userId>1</userId>
            <entityId>2</entityId>
            <topicName>
→HelloWorldTopic</topicName>
            <topicDataType>
→HelloWorld</topicDataType>
        </writer>
    </participant>
</staticdiscovery>
```

</td><td>

```xml
<staticdiscovery>
    <participant>
        <name>HelloWorldSubscriber</
→name>
        <reader>
            <userId>3</userId>
            <entityId>4</entityId>
            <topicName>
→HelloWorldTopic</topicName>
            <topicDataType>
→HelloWorld</topicDataType>
        </reader>
    </participant>
</staticdiscovery>
```

</td></tr>
</table>

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++**

```cpp
// 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);
```

SUBSCRIBER — **C++**

```cpp
// 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);
```

PUBLISHER — **XML**

```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>
<!-->STATIC_DISCOVERY_USE_CASE_PUB<-->
<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_
```

SUBSCRIBER — **XML**

```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>
```

16.3 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>
```

16.3.1 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

```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>
```

16.3.2 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

```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
```

CHAPTER 17

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.

17.1 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.

CHAPTER 18

Execution and IDL Definition

18.1 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*.

18.2 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

18.2.1 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

18.2.2 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>

18.2.3 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>

18.2.4 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;
```

(continues on next page)

(continued from previous page)

```
    string string_value;
};
```

Would be converted to:

```
class Structure
{
public:
   Structure();
   ~Structure();
   Structure(const Structure &x);
   Structure(Structure &&x);
   Structure& operator=( const Structure &x);
   Structure& operator=(Structure &&x);

   void octet_value(uint8_t _octet_value);
   uint8_t octet_value() const;
   uint8_t& octet_value();
   void long_value(int64_t _long_value);
   int64_t long_value() const;
   int64_t& long_value();
   void string_value(const std::string
      &_string_value);
   void string_value(std::string &&_string_value);
   const std::string& string_value() const;
   std::string& string_value();

private:
   uint8_t m_octet_value;
   int64_t m_long_value;
   std::string m_string_value;
};
```

Structures can inherit from other structures, extending their member set.

```
struct ParentStruct
{
    octet parent_member;
};

struct ChildStruct : ParentStruct
{
    long child_member;
};
```

In this case, the resulting C++ code will be:

```
class ParentStruct
{
    ...
};

class ChildStruct : public ParentStruct
{
    ...
};
```

18.2.5 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);
   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;
};
```

18.2.6 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
```

(continues on next page)

(continued from previous page)

```
{
    ...
};
```

Note that in this case, ChildBitset will have two `std::bitset` members, one belonging to ParentBitset and the other belonging to ChildBitset.

18.2.7 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
};
```

18.2.8 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,
```

(continues on next page)

(continued from previous page)

```
    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.

18.2.9 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.

18.2.10 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`.

18.2.11 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.

18.2.12 IDL 4.2 aliases

IDL 4.2 allows using the following names for primitive types:

int8
uint8
int16
uint16
int32
uint32
int64
uint64

18.2.13 Forward declaration

The application allows forward declarations:

```
struct ForwardStruct;

union ForwardUnion;
```

(continues on next page)

(continued from previous page)

```
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.

CHAPTER 19

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*.

19.1 Previous versions

19.1.1 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*.

19.1.2 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*.

19.1.3 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*.

19.1.4 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*.

19.1.5 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*.

19.1.6 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*

19.1.7 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*

19.1.8 Version 1.8.1

This release includes the following features:

- Implementation of *Liveliness* 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*.

19.1.9 Version 1.8.0

This release included the following features:

- Implementation of IDL 4.2.
- Implementation of *Deadline* QoS.
- Implementation of *Lifespan* QoS.
- Implementation of *Disable positive acks* 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 (see *Participant configuration*).
- 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*.

19.1.10 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*.

19.1.11 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*.

19.1.12 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*.

19.1.13 Version 1.6.0

This release included the following features:

- *Persistence*.
- Security access control plugin API and builtin *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*.

19.1.14 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*.

19.1.15 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*.

19.1.16 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.

19.1.17 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.

19.1.18 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.