

---

# **FastRTPS Documentation**

*Release 1.5.0*

**eProsima**

**Jun 18, 2018**



<b>1</b>	<b>Requirements</b>	<b>3</b>
1.1	Common Dependencies . . . . .	3
1.2	Windows 7 32-bit and 64-bit . . . . .	3
<b>2</b>	<b>Installation from Binaries</b>	<b>5</b>
2.1	Windows 7 32-bit and 64-bit . . . . .	5
2.2	Linux . . . . .	5
<b>3</b>	<b>Installation from Sources</b>	<b>7</b>
3.1	Security . . . . .	7
<b>4</b>	<b>Getting Started</b>	<b>9</b>
4.1	Brief introduction to the RTPS protocol . . . . .	9
4.2	Building your first application . . . . .	10
<b>5</b>	<b>Library Overview</b>	<b>13</b>
5.1	Fast RTPS architecture . . . . .	14
<b>6</b>	<b>Objects and Data Structures</b>	<b>15</b>
6.1	Publisher Subscriber Module . . . . .	15
6.2	RTPS Module . . . . .	15
<b>7</b>	<b>Publisher-Subscriber Layer</b>	<b>17</b>
7.1	How to use the Publisher-Subscriber Layer . . . . .	17
7.2	Configuration . . . . .	18
7.3	XML profiles . . . . .	28
7.4	Additional Concepts . . . . .	29
<b>8</b>	<b>Writer-Reader Layer</b>	<b>31</b>
8.1	Relation to the Publisher-Subscriber Layer . . . . .	31
8.2	How to use the Writer-Reader Layer . . . . .	31
8.3	Configuring Readers and Writers . . . . .	33
8.4	Configuring the History . . . . .	34
<b>9</b>	<b>Advanced Functionalities</b>	<b>35</b>
9.1	Topics and Keys . . . . .	35
9.2	Flow Controllers . . . . .	36
9.3	Sending large data . . . . .	36

9.4	Transport Layer . . . . .	37
9.5	Discovery . . . . .	38
9.6	Subscribing to Discovery Topics . . . . .	40
9.7	Tuning . . . . .	40
9.8	Additional Quality of Service options . . . . .	42
<b>10</b>	<b>Security</b>	<b>43</b>
10.1	Authentication plugins . . . . .	43
10.2	Access control plugins . . . . .	43
10.3	Cryptographic plugins . . . . .	44
10.4	Built-in plugins . . . . .	44
<b>11</b>	<b>Persistence</b>	<b>55</b>
11.1	Configuration . . . . .	55
11.2	Built-in plugins . . . . .	55
<b>12</b>	<b>Code generation using <i>fastrtps</i>gen</b>	<b>57</b>
12.1	Output . . . . .	57
12.2	Where to find <i>fastrtps</i> gen . . . . .	58
<b>13</b>	<b>Introduction</b>	<b>59</b>
13.1	Compile . . . . .	59
<b>14</b>	<b>Execution and IDL Definition</b>	<b>61</b>
14.1	Building publisher/subscriber code . . . . .	61
14.2	Defining a data type via IDL . . . . .	61
<b>15</b>	<b>Version 1.5.0</b>	<b>67</b>
15.1	Previous versions . . . . .	67



*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*



*eProxima 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++ 2013 or 2015 Redistributable Package

*eProxima 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.



---

## Installation from Binaries

---

You can always download the latest binary release of *eProxima 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

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

- **FASTRTPSHOME:** Root folder where *eProxima 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 *eProxima Fast RTPS* and its required package *eProxima Fast CDR*. You will have 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
```

---

## Installation from Sources

---

Clone the project from Github:

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

If you are on Linux, execute:

```
$ cmake -DTHIRDPARTY=ON ..
$ make
$ sudo make install
```

If you are on Windows, choose your version of Visual Studio using CMake option `-G`:

```
> cmake -G "Visual Studio 14 2015 Win64" -DTHIRDPARTY=ON ..
> cmake --build . --target install
```

If you want to compile *fastrtps* java application, you will need to add the argument `-DBUILD_JAVA=ON` when calling CMake (see [Compile](#)).

If you want to compile the examples, you will need to add the argument `-DCOMPLETE_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 *fastrtps* please see [Compile](#).

### 3.1 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 two built-in security plugins. Both have the dependency of OpenSSL library.

### 3.1.1 OpenSSL installation on Linux

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

```
sudo yum install openssl-devel
```

### 3.1.2 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 the installer that fits your requirements and install it. After installing, add the environment variable `OPENSSL_ROOT_DIR` pointing to the installation root directory. For example:

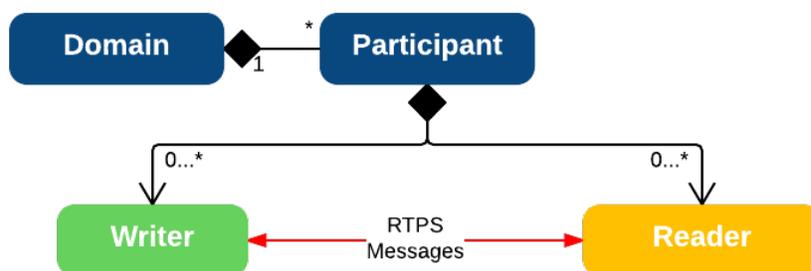
```
OPENSSL_ROOT_DIR=C:\OpenSSL-Win64
```

## 4.1 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. Interface Definition Language (IDL) is used to define the data type of the topic and 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 Windows:

```
fastrtpsgen.bat -example x64Win64VS2015 HelloWorld.idl
```

On Linux:

```
fastrtpsgen -example x64Linux2.6gcc HelloWorld.idl
```

The `-example` option creates an example application, which you can use to spawn any number of publishers and a subscribers associated with your topic.

```
./HelloWorldPublisherSubscriber publisher
./HelloWorldPublisherSubscriber subscriber
```

On Windows:

```
HelloWorldPublisherSubscriber.exe publisher
HelloWorldPublisherSubscriber.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 myHelloWorld;
myHelloWorld.msg("HelloWorld");
mp_publisher->write((void*)&myHelloWorld);
```

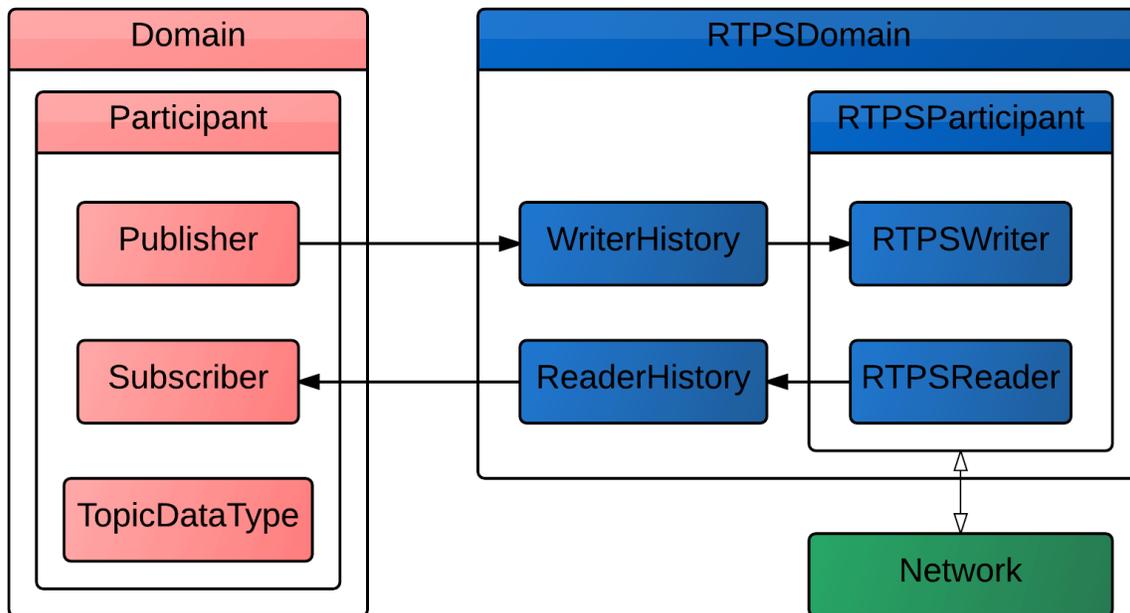
Take a look at the *examples/* folder for ideas on how to improve this basic application through different configuration options, and for examples of advanced Fast RTPS features.



## 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 to 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 channel depends on the transport layer (e.g. an 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.

---

## 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 that conform 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 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 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 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 to a cache change. May be empty depending on the message and change type.
- `Message` Defines the organization of a 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.

---

## 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 m_listener; //Class that implements callbacks from the publisher
Publisher *publisher = Domain::createPublisher(participant, publisher_attr,
↳ (PublisherListener *) &m_listener);
```

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

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

The Subscriber creation and implementation is symmetric.

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

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

## 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.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_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++	XML
<pre>participant_attr.setName("my_ ↔participant");</pre>	<pre>&lt;profiles&gt;   &lt;participant profile_name= ↔"participant_xml_profile"&gt;     &lt;rtps&gt;       &lt;name&gt;my_participant&lt;/ ↔name&gt;       &lt;/rtps&gt;     &lt;/participant&gt;   &lt;/profiles&gt;</pre>

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

C++	XML
<pre>participant_attr.rtps.builtin. ↔domainId = 80;</pre>	<pre>&lt;profiles&gt;   &lt;participant profile_name= ↔"participant_xml_profile"&gt;     &lt;rtps&gt;       &lt;builtin&gt;         &lt;domainId&gt;80&lt;/ ↔domainId&gt;       &lt;/builtin&gt;     &lt;/rtps&gt;   &lt;/participant&gt; &lt;/profiles&gt;</pre>

## 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);
```

(continues on next page)

(continued from previous page)

```
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 a 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++	XML
<pre>publisher_attr.topic.topicDataType_ ↳= "HelloWorldType"; publisher_attr.topic.topicName = ↳"HelloWorldTopic";  subscriber_attr.topic.topicDataType_ ↳= "HelloWorldType"; subscriber_attr.topic.topicName = ↳"HelloWorldTopic";</pre>	<pre>&lt;profiles&gt;   &lt;publisher profile_name= ↳"publisher_xml_profile"&gt;     &lt;topic&gt;       &lt;dataType&gt;HelloWorldType&lt;/ ↳dataType&gt;       &lt;name&gt;HelloWorldTopic&lt;/ ↳name&gt;     &lt;/topic&gt;   &lt;/publisher&gt;    &lt;subscriber profile_name= ↳"subscriber_xml_profile"&gt;     &lt;topic&gt;       &lt;dataType&gt;HelloWorldType&lt;/ ↳dataType&gt;       &lt;name&gt;HelloWorldTopic&lt;/ ↳name&gt;     &lt;/topic&gt;   &lt;/subscriber&gt; &lt;/profiles&gt;</pre>

## Reliability

The RTPS standard defines two behaviour 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++	XML
<pre> publisher_attr.qos.m_reliability. ↪kind =     RELIABLE_RELIABILITY_QOS;  subscriber_attr.qos.m_reliability. ↪kind =     BEST_EFFORT_RELIABILITY_QOS; </pre>	<pre> &lt;profiles&gt;   &lt;publisher profile_name= ↪"publisher_xml_profile"&gt;     &lt;qos&gt;       &lt;reliability&gt;         &lt;kind&gt;RELIABLE&lt;/kind&gt;       &lt;/reliability&gt;     &lt;/qos&gt;   &lt;/publisher&gt;    &lt;subscriber profile_name= ↪"subscriber_xml_profile"&gt;     &lt;qos&gt;       &lt;reliability&gt;         &lt;kind&gt;BEST_EFFORT&lt;/kind&gt;       &lt;/reliability&gt;     &lt;/qos&gt;   &lt;/subscriber&gt; &lt;/profiles&gt; </pre>

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++	XML
<pre> publisher_attr.topic.historyQos. ↪kind =     KEEP_ALL_HISTORY_QOS;  subscriber_attr.topic.historyQos. ↪kind =     KEEP_LAST_HISTORY_QOS; subscriber_attr.topic.historyQos. ↪depth = 5 </pre>	<pre> &lt;profiles&gt;   &lt;publisher profile_name= ↪"publisher_xml_profile"&gt;     &lt;topic&gt;       &lt;historyQos&gt;         &lt;kind&gt;KEEP_ALL&lt;/kind&gt;       &lt;/historyQos&gt;     &lt;/topic&gt;   &lt;/publisher&gt;    &lt;subscriber profile_name= ↪"subscriber_xml_profile"&gt;     &lt;topic&gt;       &lt;historyQos&gt;         &lt;kind&gt;KEEP_LAST&lt;/kind&gt;         &lt;depth&gt;5&lt;/depth&gt;       &lt;/historyQos&gt;     &lt;/topic&gt;   &lt;/subscriber&gt; &lt;/profiles&gt; </pre>

## 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++	XML
<pre> publisher_attr.qos.m_durability. ↪kind =   TRANSIENT_LOCAL_DURABILITY_QOS;  subscriber_attr.qos.m_durability. ↪kind =   VOLATILE_DURABILITY_QOS; </pre>	<pre> &lt;profiles&gt;   &lt;publisher profile_name= ↪"publisher_xml_profile"&gt;     &lt;qos&gt;       &lt;durability&gt;         &lt;kind&gt;TRANSIENT_LOCAL&lt;/ ↪kind&gt;       &lt;/durability&gt;     &lt;/qos&gt;   &lt;/publisher&gt;    &lt;subscriber profile_name= ↪"subscriber_xml_profile"&gt;     &lt;qos&gt;       &lt;durability&gt;         &lt;kind&gt;VOLATILE&lt;/kind&gt;       &lt;/durability&gt;     &lt;/qos&gt;   &lt;/subscriber&gt; &lt;/profiles&gt; </pre>

## Resource limits

Allow to control the maximum size of the History and other resources.

C++	XML
<pre> publisher_attr.topic. ↪resourceLimitsQos.max_samples = ↵ ↪200;  subscriber_attr.topic. ↪resourceLimitsQos.max_samples = ↵ ↪200; </pre>	<pre> &lt;profiles&gt;   &lt;publisher profile_name= ↪"publisher_xml_profile"&gt;     &lt;topic&gt;       &lt;resourceLimitsQos&gt;         &lt;max_samples&gt;200&lt;/max_ ↪samples&gt;       &lt;/resourceLimitsQos&gt;     &lt;/topic&gt;   &lt;/publisher&gt;    &lt;subscriber profile_name= ↪"subscriber_xml_profile"&gt;     &lt;topic&gt;       &lt;resourceLimitsQos&gt;         &lt;max_samples&gt;200&lt;/max_ ↪samples&gt;       &lt;/resourceLimitsQos&gt;     &lt;/topic&gt;   &lt;/subscriber&gt; &lt;/profiles&gt; </pre>

## Unicast locators

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

C++	XML
<pre>Locator_t new_locator; new_locator.port = 7800;  subscriber_attr.unicastLocatorList. ↳push_back(new_locator);  publisher_attr.unicastLocatorList. ↳push_back(new_locator);</pre>	<pre>&lt;profiles&gt;   &lt;publisher profile_name= ↳"publisher_xml_profile"&gt;     &lt;unicastLocatorList&gt;       &lt;locator&gt;         &lt;port&gt;7800&lt;/port&gt;       &lt;/locator&gt;     &lt;/unicastLocatorList&gt;   &lt;/publisher&gt;    &lt;subscriber profile_name= ↳"subscriber_xml_profile"&gt;     &lt;unicastLocatorList&gt;       &lt;locator&gt;         &lt;port&gt;7800&lt;/port&gt;       &lt;/locator&gt;     &lt;/unicastLocatorList&gt;   &lt;/subscriber&gt; &lt;/profiles&gt;</pre>

## Multicast locators

They are network endpoints where the entity will receive data. For more information about network, see *Setting up network configuration*. 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++	XML
<pre>Locator_t new_locator;  new_locator.set_IP4_address("239. ↪255.0.4"); new_locator.port = 7900;  subscriber_attr. ↪multicastLocatorList.push_ ↪back(new_locator);  publisher_attr.multicastLocatorList. ↪push_back(new_locator);</pre>	<pre>&lt;profiles&gt;   &lt;publisher profile_name= ↪"publisher_xml_profile"&gt;     &lt;multicastLocatorList&gt;       &lt;locator&gt;         &lt;address&gt;239.255.0.4&lt;/ ↪address&gt;         &lt;port&gt;7900&lt;/port&gt;       &lt;/locator&gt;     &lt;/multicastLocatorList&gt;   &lt;/publisher&gt;    &lt;subscriber profile_name= ↪"subscriber_xml_profile"&gt;     &lt;multicastLocatorList&gt;       &lt;locator&gt;         &lt;address&gt;239.255.0.4&lt;/ ↪address&gt;         &lt;port&gt;7900&lt;/port&gt;       &lt;/locator&gt;     &lt;/multicastLocatorList&gt;   &lt;/subscriber&gt; &lt;/profiles&gt;</pre>

## 7.2.3 Advanced configuration

### Setting up network configuration

*eProsima Fast RTPS* implements an architecture of pluggable network transports. Current version implements two network transports: UDPv4 and UDPv6. By default, when a Participant is created, one built-in UDPv4 network transport is configured.

You can add custom transport using the attribute `rtps.userTransports`.

```
//Creation of the participant
eprosima::fastrtps::ParticipantAttributes part_attr;

auto customTransport = std::make_shared<UDPv4TransportDescriptor>();
customTransport->sendBufferSize = 9216;
customTransport->receiveBufferSize = 9216;

part_attr.rtps.userTransports.push_back(customTransport);
```

Also you can disable built-in UDPv4 network transport using the attribute `rtps.useBuiltinTransports`.

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

part_attr.rtps.useBuiltinTransports = false;
```

Network endpoints are defined by *eProsima Fast RTPS* as locators. Locators in *eProsima Fast RTPS* are enclosed as type `Locator_t`, which has the following fields:

- `kind`: Defines the protocol. *eProxima Fast RTPS* currently supports UDPv4 or UDPv6
- `port`: Port as an UDP/IP port.
- `address`: Maps to IP address

### Listening locators

*eProxima Fast RTPS* divides listening locators in 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 built-in endpoints. You can set your own locators using attribute `rtps.builtin.metatrafficMulticastLocatorList`.

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

// 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.set_IP4_address(239, 255, 0 , 1);
locator.port = 22222;

part_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 built-in endpoints. You can set your own locators using attribute `rtps.builtin.metatrafficUnicastLocatorList`.

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

// 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.set_IP4_address(192, 168, 0 , 1);
locator.port = 22223;

part_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`.

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

// 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.set_IP4_address(239, 255, 0 , 1);
locator.port = 22224;

part_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`.

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

// 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.set_IP4_address(192, 168, 0 , 1);
locator.port = 22225;
```

(continues on next page)

(continued from previous page)

```
part_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 * <i>domainId</i> + offsetd0
Metatraffic unicast	PB + DG * <i>domainId</i> + offsetd1 + PG * <i>participantId</i>
User multicast	PB + DG * <i>domainId</i> + offsetd2
User unicast	PB + DG * <i>domainId</i> + offsetd3 + PG * <i>participantId</i>

These predefined rules use some values explained here:

- **DG:** DomainId Gain. You can set this value using attribute `rtps.port.domainIDGain`. Default value is 250.
- **PG:** ParticipantId Gain. You can set this value using attribute `rtps.port.participantIDGain`. Default value is 2.
- **PB:** Port Base number. You can set this value using attribute `rtps.port.portBase`. 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`.

A UDPv4 unicast locator supports to have a null address. In that case *eProsima Fast RTPS* understands to get local network addresses and use them.

A UDPv4 locator support to have a zero port. In that case *eProsima Fast RTPS* understands to calculate well-known port for that type of traffic.

## Sending locators

These locators are used to create network endpoints to send all network messages. You can set your own locators using the attribute `rtps.defaultOutLocatorList`.

```
eProsima::fastrtps::ParticipantAttributes part_attr;

// This locator will create a socket to send network message on UDPv4 port 34000 over
↪network interface 192.168.0.1
Locator_t locator.set_IP4_address(192.168.0.1);
locator.port = 34000;

part_attr.rtps.defaultOutLocatorList.push_back(locator);
```

By default *eProsima Fast RTPS* sends network messages using a random UDPv4 port over all interface networks.

A UDPv4 unicast locator supports to have a null address. In that case *eProsima Fast RTPS* understands to get local network addresses and use them.

A UDPv4 locator support to have a zero port. In that case *eProsima Fast RTPS* understands to get a random UDPv4 port.

## Initial peers

These locators are used to know where to send initial discovery network messages. You can set your own locators using attribute `rtps.builtin.initialPeersList`. By default *eProsima Fast RTPS* uses as initial peers the Metatraffic Multicast Locators.

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

// This locator configures as initial peer the UDPv4 address 192.168.0.2:7600.
// Initial discovery network messages will send to this UDPv4 address.
Locator_t locator.set_IP4_address(192.168.0.2);
locator.port = 7600;

part_attr.rtps.builtin.initialPeersList.push_back(locator);
```

## Tips

### Disabling all multicast traffic

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

// 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;
initial_peer.set_IP4_address(192, 168, 0, 1);
participant_attr.rtps.builtin.initialPeersList.push_back(initial_peer);
```

## 7.3 XML profiles

In the *Configuration* section you could see how to configure entity attributes using XML profiles, but this section goes deeper into it.

XML profiles are loaded from XML files. *eProsima Fast RTPS* permits to load as much XML files as you want. An XML file can contain several XML profiles. An XML profile is defined by a unique name that is used to reference the XML profile when you create a Fast RTPS entity. *eProsima Fast RTPS* will also try to find in current execution path and load an XML file with the name `DEFAULT_FASTRTPS_PROFILES.xml`. If this file exists, it is loaded at the library initialization.

### 7.3.1 Making an XML

An XML file can contain several XML profiles. They can be divided in participant, publisher and subscriber profiles.

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles>
  <participant profile_name="participant_profile">
    ....
  </participant>

  <publisher profile_name="publisher_profile">
    ....
  </publisher>

  <subscriber profile_name="subscriber_profile">
    ....
  </subscriber>
</profiles>
```

The entire list of supported attributes can be checked in this [XSD file](#).

### 7.3.2 Loading and applying profiles

Before creating any entity, you can load XML files using `Domain::loadXMLProfilesFile` function. `createParticipant`, `createPublisher` and `createSubscriber` have a version that expects the profile name as 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
↪");
```

## 7.4 Additional Concepts

### 7.4.1 Using message meta-data

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

```
HelloWorld m>Hello;
SampleInfo_t m_info;
sub->takeNextData((void*)&m>Hello, &m_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((m_info->sampleKind == ALIVE) & (m_info->OwnershipStrength > 25 ){
    //Process data
}
```

### 7.4.2 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
<code>onNewDataMessage</code>	N	Y
<code>onSubscriptionMatched</code>	N	Y
<code>onPublicationMatched</code>	Y	N

---

## 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 Pparam;
Pparam.setName("participant");
RTPSParticipant* p = RTPSDomain::createRTPSParticipant(Pparam);
```

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 Pparam;
Pparam.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 hatt;
WriterHistory * history = new WriterHistory(hatt);
WriterAttributes watt;
RTPSWriter* writer = RTPSDomain::createRTPSWriter(rtpsParticipant, watt, 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 listen;
HistoryAttributes hatt;
ReaderHistory * history = new ReaderHistory(hatt);
ReaderAttributes ratt;
RTPSReader* reader = RTPSDomain::createRTPSReader(rtpsParticipant, watt, history, &
↪listen);
```

## 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 an 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* ch = writer->new_change([]() -> uint32_t { return 255;}, ALIVE);
//Write serialized data into the change
```

(continues on next page)

(continued from previous page)

```

ch->serializedPayload.length = sprintf((char*) ch->serializedPayload.data, "My_
↳example string %d", 2)+1;
//Insert change back into the history. The Writer takes care of the rest.
history->add_change(ch);

```

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);
    }
}

```

Additionally you can read an incoming message directly by interacting with the History:

```

//Blocking method
reader->waitForUnreadMessage();
CacheChange_t* change;
//Take the first unread change present in the History
if(reader->nextUnreadCache(&change))
{
    /* use data */
}
//Once done, remove the change
history->remove_change(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:

```

Wattr.endpoint.reliabilityKind = BEST_EFFORT;

```

### 8.3.1 Setting the data durability kind

The Durability parameter defines the behaviour 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:

```
WriterAttributes Wparams;  
Wparams.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:

```
HistoryAttributes.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 initial amount of allocated changes:

```
HistoryAttributes.initialReservedCaches = 250; //Defaults to 500  
HistoryAttributes.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.

---

## Advanced Functionalities

---

This section covers slightly more advanced, but useful features that enriches 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 two 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.

Publishers and Subscribers using topics with keys must be configured to use them, otherwise they will have no effect:

```
//Publisher-Subscriber Layer configuration  
PubAttributes.topic.topicKind = WITH_KEY
```

The RTPS Layer requires you to call the `getKey()` method manually within your callbacks.

You can tweak the History to accomodate data from multiples keys based on your current configuration. This consist on defining a maximum number of data sinks and a maximum size for each sink:

```
Rparam.topic.resourceLimitsQos.max_instances = 3; //Set the subscriber to remember_  
↪and store up to 3 different keys  
Rparam.topic.resourceLimitsQos.max_samples_per_instance = 20; //Hold a maximum of 20_  
↪samples per key
```

Note that your History must be big enough to accomodate the maximum number of samples for each key. eProsima Fast RTPS will notify you if your History is too small.

## 9.2 Flow Controllers

*eProsima Fast RTPS* supports user configurable flow controllers on a Publisher and Participant level. These controllers can be used to limit the amount of data to be sent under certain conditions depending on the kind of controller implemented.

The current release implement throughput controllers, which can be used to limit the total message throughput to be sent over the network per time measurement unit. In order to use them, a descriptor must be passed into the Participant or Publisher Attributes.

```
PublisherAttributes WparamSlow;
ThroughputControllerDescriptor slowPublisherThroughputController{300000, 1000}; //
↪Limit to 300kb per second
WparamSlow.throughputController = slowPublisherThroughputController;
```

In the Writer-Reader layer, the throughput controllers is built-in and the descriptor defaults to infinite throughput. To change the values:

```
WriterAttributes WParams;
WParams.throughputController.size = 300000; //300kb
WParams.throughputController.timeMS = 1000; //1000ms
```

Note that specifying a throughput controller with a size smaller than the socket size can cause messages to never become sent.

## 9.3 Sending large data

The default message size *eProsima Fast RTPS* uses is a conservative value of 65kb. If your topic data is bigger, it must be fragmented.

Fragmented messages are sent over multiple packets, as understood by the particular transport layer. To make this possible, you must configure the Publisher to work in asynchronous mode.

```
PublisherAttributes Wparam;
Wparam.qos.m_publishMode.kind = ASYNCHRONOUS_PUBLISH_MODE; // Allows fragmentation
```

In the Writer-Subscriber layer, you have to configure the Writer:

```
WriterAttributes Wparam;
Wparam.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*

### 9.3.1 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 have 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, 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 reliable mode.

The default message size of this fragments using the UDPv4 transport has a value of 65kb (which includes the space reserved to 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 a 5% of the total bandwidth. Anyway, this values are highly dependant of the specific application and its desired behaviour.

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"
```

### 9.3.2 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, asynchronous mode has to be activated in the publisher parameters.

In audio or video transmissions, sometimes is better to have an 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.

## 9.4 Transport Layer

Unless you specify other configuration, *eProsima Fast RTPS* will use its built-in UDPv4 Transport Layer with a default configuration. You can change this default configuration or switch to UDPv6 by providing an alternative configuration when you create the Participant.

```
RTPSParticipantAttributes Pparams;
auto my_transport = std::make_shared<UDPv6Transport::TransportDescriptor>(); //Create_
↳a descriptor for the new transport
Pparams.useBuiltinTransport = false; //Disable the built-in Transport Layer
Pparams.userTransports.push_back(my_transport); //Link the Transport Layer to the_
↳Participant
```

Note that unless you manually disable the built-in transport layer, the Participant will use your custom transport configuration along the built-in one.

This distribution comes with an example of how to change the configuration of the transport layer. It can be found [here](#).

## 9.5 Discovery

Fast RTPS provides a discovery mechanism that allows to match automatically publishers and subscribers. The discovery mechanism is divided in two phases: Participant Discovery Phase and Endpoints Discovery Phase.

- **Participant Discovery Phase (PDP)** Before discovering any entity of a remote participant, both participants have to met between them. Participant Discovery Phase provides this step and is responsible for sending periodic information about itself. To know how to configure where to send this periodic information, see *Initial peers*. When both participants are met, is the turn of Endpoints Discovery Phase.
- **Endpoints Discovery Phase (EDP)** This phase is responsible for sending entities information to the remote participant. Also it has to process the entities information of the remote participant and check which entities can match between them.

By default the discovery mechanism is enabled, but you can disable it through participant attributes.

```
ParticipantAttributes participant_attr;
participant_attr.rtps.builtin.use_SIMPLE_RTPSParticipantDiscoveryProtocol = false;
```

### 9.5.1 Static Endpoints Discovery

Endpoints Discovery Phase can be replaced by a static version that doesn't send any information. It is useful when you have a limited network bandwidth and a well-known schema of publishers and subscribers. Instead of receiving entities information for matching, this information is loaded from a XML file.

First of all, you have to disable the Endpoints Discovery Phase and enable the Static Endpoints Discovery. This can be done from the participant attributes.

```
ParticipantAttributes participant_attr;
participant_attr.rtps.builtin.use_SIMPLE_EndpointDiscoveryProtocol = false;
participant_attr.rtps.builtin.use_STATIC_EndpointDiscoveryProtocol = true;
```

Then, you will need to load the XML file containing the configuration of the remote participant. So, for example, if there is a remote participant with a subscriber which is waiting to receive samples from your publisher, you will need to load the configuration of this remote participant.

```
participant_attr.rtps.builtin.setStaticEndpointXMLFilename (
↪ "ParticipantWithASubscriber.xml");
```

A basic XML configuration file for this remote participant would contain information like the name of the remote participant, the topic name and data type of the subscriber, and its entity and user defined ID. All these values have to exactly match the parameter values used to configure the remote participant (through the class ParticipantAttributes) and its subscriber (through the class SubscriberAttributes). Missing elements will acquire default values. For example:

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

The XML that configures the participant on the other side (in this case, a subscriber) could look like this:

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

You can find an example that uses [Static Endpoint Discovery](#).

The full list of fields for readers and writers includes the following parameters:

- **userId**: numeric value.
- **entityID**: numeric value.
- **expectsInlineQos**: *true* or *false*. (only valid for readers)
- **topicName**: text value.
- **topicDataType**: text value.
- **topicKind**: *NO\_KEY* or *WITH\_KEY*.
- **reliabilityQos**: *BEST\_EFFORT\_RELIABILITY\_QOS* or *RELIABLE\_RELIABILITY\_QOS*.
- **unicastLocator**
  - address: text value.
  - port: numeric value.
- **multicastLocator**
  - address: text value.
  - port: numeric value.
- **topic**
  - name: text value.
  - data type: text value.
  - kind: text value.
- **durabilityQos**: *VOLATILE\_DURABILITY\_QOS*, *TRANSIENT\_LOCAL\_DURABILITY\_QOS* or *TRANSIENT\_DURABILITY\_QOS*.
- **ownershipQos**
  - kind: *SHARED\_OWNERSHIP\_QOS* or *EXCLUSIVE\_OWNERSHIP\_QOS*.
- **partitionQos**: text value.
- **livelinessQos**
  - kind: *AUTOMATIC\_LIVELINESS\_QOS*, *MANUAL\_BY\_PARTICIPANT\_LIVELINESS\_QOS* or *MANUAL\_BY\_TOPIC\_LIVELINESS\_QOS*.
  - leaseDuration\_ms: numeric value.

## 9.6 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. It is possible to create a custom listener that listens to the Endpoint Discovery Protocol meta-data. This allows you to create your own network analysis tools.

```

/* Create Custom user ReaderListeners */
CustomReaderListener *my_readerListenerSub = new(CustomReaderListener);
CustomReaderListener *my_readerListenerPub = new(CustomReaderListener);
/* Get access to the EDP endpoints */
std::pair<StatefulReader*,StatefulReader*> EDPReaders = my_participant->
↳ getEDPReaders();
/* Install the listeners for Subscribers and Publishers Discovery Data*/
EDPReaders.first()->setListener(my_readerListenerSub);
EDPReaders.second()->setListener(my_readerListenerPub);
/* ... */
/* Custom Reader Listener onNewCacheChangeAdded*/
void onNewCacheChangeAdded(RTPSReader * reader, const CacheChange_t * const change)
{
    (void) reader;
    if (change->kind == ALIVE) {
        WriterProxyData proxyData;

        CDRMessage_t tempMsg(0);
        tempMsg.wraps = true;
        tempMsg.msg_endian = change_in->serializedPayload.encapsulation == PL_CDR_BE ?
↳ BIGEND : LITTLEEND;
        tempMsg.length = change_in->serializedPayload.length;
        tempMsg.max_size = change_in->serializedPayload.max_size;
        tempMsg.buffer = change_in->serializedPayload.data;

        if (proxyData.readFromCDRMessage(&tempMsg)) {
            cout << proxyData.topicName();
            cout << proxyData.typeName();
        }
    }
}

```

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.

## 9.7 Tuning

### 9.7.1 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*.

### 9.7.2 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 increasing the socket buffer used to send data. `listenSocketBufferSize` attribute helps increasing the socket buffer used to read data.

C++	XML
<pre>part_attr.rtps.sendSocketBufferSize_ ↪= 1048576; part_attr.rtps. ↪listenSocketBufferSize = 4194304;</pre>	<pre>&lt;profiles&gt;   &lt;participant profile_name= ↪"participant_xml_profile"&gt;     &lt;rtps&gt;       &lt;sendSocketBufferSize&gt; ↪1048576&lt;/sendSocketBufferSize&gt;       &lt;listenSocketBufferSize&gt; ↪4194304&lt;/listenSocketBufferSize&gt;     &lt;/rtps&gt;   &lt;/participant&gt; &lt;/profiles&gt;</pre>

### 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.

For getting 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
```

### 9.7.3 Tuning Reliable mode

RTPS protocol can maintain a 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`:

```
PublisherAttributes pubAttr;
pubAttr.times.heartbeatPeriod.seconds = 0;
pubAttr.times.heartbeatPeriod.fraction = 4294967 * 500; //500 ms
```

In the Writer-Reader layer, this belong to the `WriterAttributes`:

```
WriterAttributes Wattr;  
Wattr.times.heartbeatPeriod.seconds = 0;  
Wattr.times.heartbeatPeriod.fraction = 4294967 * 500; //500 ms
```

A smaller heartbeat period increases the amount 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`, determinates 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`.

### 9.7.4 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*.

## 9.8 Additional Quality of Service options

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

- **Deadline:** Rise an alarm when the frequency of message arrival for a topic falls below a certain threshold.
- **Ownership Srength:** 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.

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](mailto:support@eprosima.com).

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 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 Cryptographic plugin manually with 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

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 <b>file</b> schema shall refer to a X.509 v3 certificate in PEM format.
identity_certificate	URI to a X509 certificate signed by the Identity CA in PEM format containing the signed public key for the Participant. Supported URI schemes: file.
identity_crl ( <i>optional</i> )	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 ( <i>optional</i> )	A password used to decrypt the private_key.

## Generation of x509 certificates

You can generate you 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 extensions 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
<code>permissions_ca</code>	URI to the X509 certificate of the Permissions CA. Supported URI schemes: file. The <b>file</b> schema shall refer to a X.509 v3 certificate in PEM format.
<code>governance</code>	URI to shared Governance Document signed by the Permissions CA in S/MIME format. Supported URI schemes: file.
<code>permissions</code>	URI to the Participant permissions document signed by the Permissions CA in S/MIME format. Supported URI schemes: file.

### Permissions CA Certificate

This is a 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 unsecure 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 unsecure 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 entity's RTPS submessages shall be encrypted by 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 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 appear 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 contains 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 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 a X509 certificate. Generation of a 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 maincert.
↳pem -inkey maincakey.pem

# Permissions document: permissions.xml
openssl smime -sign -in permissions.xml -text -out permissions.smime -signer_
↳maincert.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`.

### 10.4.4 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

```
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", "maincacert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪identity_certificate", "appcert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪private_key", "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",
    "maincacet.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.access.builtin.Access-
↪Permissions.governance",
    "governance.xml");
part_attr.rtps.properties.properties().emplace_back("dds.sec.access.builtin.Access-
↪Permissions.permissions",
    "permissions.xml");

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

This example show 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 Writer to encrypt the payload (user data).

**Participant attributes**

```
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", "maincert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪identity_certificate", "appcert.pem");
part_attr.rtps.properties.properties().emplace_back("dds.sec.auth.builtin.PKI-DH.
↪private_key", "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");
```

**Writer attributes**

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

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

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

**Reader attributes**

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

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

Fast RTPS can be configured to provide persistence to the history of a writer and the highest sequence number notified by a reader.

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*

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.

## 11.1 Configuration

In order for the persistence service 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`

## 11.2 Built-in plugins

Current version comes out with one persistence built-in plugin:

- **SQLITE3**: this plugin provides persistence on a local file using SQLite3 API.

## 11.2.1 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: <i>persistence.db</i>

## 11.2.2 Example

This example shows you how to configure a 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;
```

---

## 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 <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.

### 12.1 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`: Main file of the example application in case it is generated.
- Makefiles or Visual studio project files.

## 12.2 Where to find *fastrtpsgen*

If you are using the binary distribution of *eProxima 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*.

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 a first implementation of a publisher and a subscriber using eProsima RTPS library.

## 13.1 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 generate *fastrtpsgen* you will need to add the argument `-DBUILD_JAVA=ON` when calling CMake.



## 14.1 Building publisher/subscriber code

This section guides you through the usage of this Java application and briefly describes the generated files.

The Java application can be executed using the following scripts depending on if you are on Windows or Linux:

```
> fastrtpsgen.bat
$ fastrtpsgen
```

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>	Output directory where the generated files are created.
-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 whether they exist.
-ppDisable	Disables the preprocessor.
-ppPath	Specifies the preprocessor path.

## 14.2 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 long	int64_t
unsigned long long	uint64_t
float	float
double	double
long double	long double
boolean	bool
string	std::string

### 14.2.1 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 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

### 14.2.2 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 long>	std::vector<int64_t>
sequence<unsigned long long>	std::vector<uint64_t>
sequence<float>	std::vector<float>
sequence<double>	std::vector<double>

### 14.2.3 Structures

You can define an IDL structure with a set of members with multiple types. It will be converted into a C++ class with each member mapped as an attribute plus methods to *get* and *set* each member.

The following IDL structure:

```

struct Structure
{
  octet octet_value;
      long long_value;
  string string_value;
};

```

Would be converted to:

```

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

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

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

```

## 14.2.4 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;
};
```

## 14.2.5 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
};
```

## 14.2.6 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;
};
```

*fastrtps*gen automatically detects these tags and correctly generates the serialization methods for the key generation function in TopicDataType (getKey). This function will obtain the 128-bit MD5 digest of the big-endian serialization of the Key Members.

## 14.2.7 Including other IDL files

You can include another IDL files in yours in order to use data types defined in them. *fastrtps*gen 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 *fastrtps*gen 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 preprocessor, you could use the parameter `-ppDisable`.



This release includes 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 *fastrtps-gen*

## 15.1 Previous versions

### 15.1.1 Version 1.4.0

This release includes 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 *fastrtps-gen*

### 15.1.2 Version 1.3.1

This release includes the following:

- New examples that illustrate how to tweak Fast RTPS towards different applications.
- Improved support for embedded Linux.

- Bug fixing.

### 15.1.3 Version 1.3.0

This release introduces 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.

### 15.1.4 Version 1.2.0

This release introduces 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.