FastRTPS Documentation

Release 1.5.0

eProsima

Feb 08, 2018

Installation manual

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:

CHAPTER 1

Requirements

eProsima Fast RTPS requires the following packages to work.

1.1 Common Dependencies

1.1.1 Gtest

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

1.1.2 Java & Gradle

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

1.2 Windows 7 32-bit and 64-bit

1.2.1 Visual C++ 2013 or 2015 Redistributable Package

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

CHAPTER 2

Installation from Binaries

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

2.1 Windows 7 32-bit and 64-bit

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

2.1.1 Environmental Variables

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

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

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

2.2 Linux

Extract the contents of the package. It will containt both *eProsima Fast RTPS* and its required package *eProsima 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
```

CHAPTER 3

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 *fastrtpsgen* 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 `-DCOMPILE_EXAMPLES=ON` when calling CMake.

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

For generate *fastrtpsgen* please see *Compile*.

3.1 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-tin 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
```

CHAPTER 4

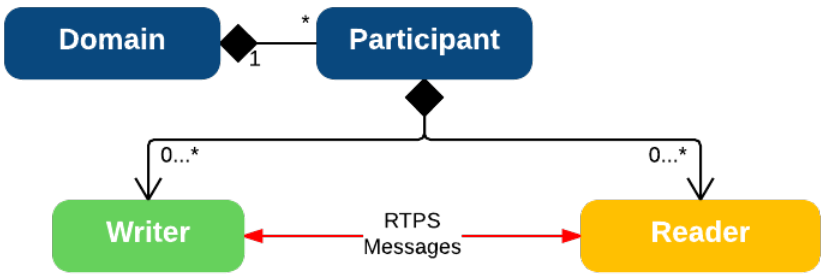
Getting Started

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.i

```
./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);
```

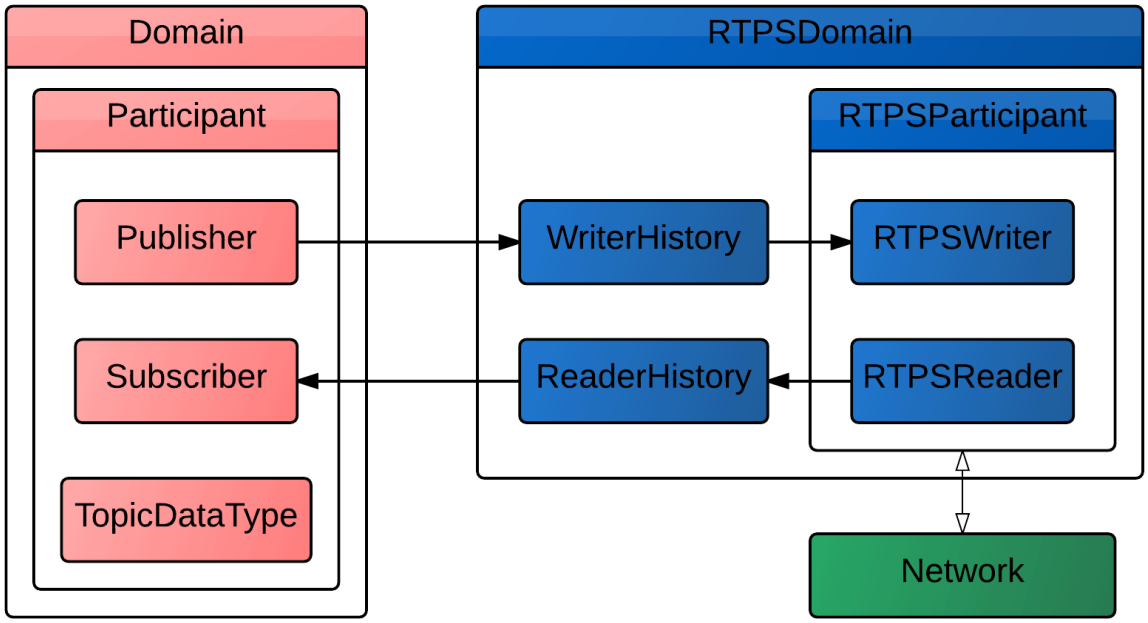
Take a look at the *examples/* folder for ideas on how to improve this basic application through different configuration options, and for examples of advanced Fast RTPS features.

CHAPTER 5

Library Overview

You can interact with Fast RTPS at two different levels:

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



In red, the Publisher-Subscriber layer offers a convenient abstraction for most use cases. It allows you to define Publishers and Subscribers associated 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.

CHAPTER 6

Objects and Data Structures

In order to make the most of *eProsima Fast RTPS* it is important to have a grasp of the objects and data structures that conform the library. *eProsima Fast RTPS* objects are classified by modules, which are briefly listed and descripted 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.

CHAPTER 7

Publisher-Subscriber Layer

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

7.1 How to use the Publisher-Subscriber Layer

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

The first step 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,
↪(SubsciberListener*)&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

`participant_attr.setName("my_participant");`	`<profiles>`
	`  <participant profile_name="participant_xml_profile">`
	`  <rtps>`
	`  <name>my_participant</name>`
	`  </rtps>`
	`  </participant>`
	`</profiles>`

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

C++	XML

`participant_attr.rtps.builtin.domainId = 80;`	`<profiles>`
	`  <participant profile_name="participant_xml_profile">`
	`  <rtps>`
	`  <builtin>`
	`  <domainId>80</domainId>`
	`  </builtin>`
	`  </rtps>`
	`  </participant>`
	`</profiles>`

7.2.2 Publisher and Subscriber configuration

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

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

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

Also these entities can be configured through an XML profile. `createPublisher` and `createSubscriber` functions accept 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++

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

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

XML

```
<profiles>
   <publisher profile_name=
↪"publisher_xml_profile">
      <topic>
         <dataType>HelloWorldType
↪</dataType>
         <name>HelloWorldTopic</
↪name>
      </topic>
   </publisher>

   <subscriber profile_name=
↪"subscriber_xml_profile">
      <topic>
         <dataType>HelloWorldType
↪</dataType>
         <name>HelloWorldTopic</
↪name>
      </topic>
   </subscriber>
</profiles>
```

- **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++

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

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

XML

```
<profiles>
   <publisher profile_name=
↪"publisher_xml_profile">
      <qos>
         <reliability>
            <kind>RELIABLE</kind>
         </reliability>
      </qos>
   </publisher>

   <subscriber profile_name=
↪"subscriber_xml_profile">
      <qos>
         <reliability>
            <kind>BEST_EFFORT</
↪kind>
         </reliability>
      </qos>
   </subscriber>
</profiles>
```

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

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

- **History:** there are two policies for sample storage:
  - Keep-All: Store all samples in memory.
  - Keep-Last (Default): Store samples up to a maximum *depth*. When this limit is reached, they start to become overwritten.

C++

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

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

XML

```
<profiles>
   <publisher profile_name=
↪"publisher_xml_profile">
      <topic>
         <historyQos>
            <kind>KEEP_ALL</kind>
         </historyQos>
      </topic>
   </publisher>

   <subscriber profile_name=
↪"subscriber_xml_profile">
      <topic>
         <historyQos>
            <kind>KEEP_LAST</kind>
            <depth>5</depth>
         </historyQos>
      </topic>
   </subscriber>
</profiles>
```

- **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.

C++

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

subscriber_attr.qos.m_durability.
↪kind =
   VOLATILE_DURABILITY_QOS;
```

XML

```
<profiles>
   <publisher profile_name=
↪"publisher_xml_profile">
      <qos>
         <durability>
            <kind>TRANSIENT_LOCAL
↪</kind>
         </durability>
      </qos>
   </publisher>

   <subscriber profile_name=
↪"subscriber_xml_profile">
      <qos>
         <durability>
            <kind>VOLATILE</kind>
         </durability>
      </qos>
   </subscriber>
</profiles>
```

- **Resource limits:** allow to control the maximum size of the History and other resources.

C++

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

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

XML

```
<profiles>
   <publisher profile_name=
↪"publisher_xml_profile">
      <topic>
         <resourceLimitsQos>
            <max_samples>200</max_
↪samples>
         </resourceLimitsQos>
      </topic>
   </publisher>

   <subscriber profile_name=
↪"subscriber_xml_profile">
      <topic>
         <resourceLimitsQos>
            <max_samples>200</max_
↪samples>
         </resourceLimitsQos>
      </topic>
   </subscriber>
</profiles>
```

- **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 locators through this attribute.

C++

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

subscriber_attr.
↪unicastLocatorList.push_
↪back(new_locator);

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

XML

```
<profiles>
   <publisher profile_name=
↪"publisher_xml_profile">
      <unicastLocatorList>
         <locator>
            <port>7800</port>
         </locator>
      </unicastLocatorList>
   </publisher>

   <subscriber profile_name=
↪"subscriber_xml_profile">
      <unicastLocatorList>
         <locator>
            <port>7800</port>
         </locator>
      </unicastLocatorList>
   </subscriber>
</profiles>
```

- **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++

```
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);
```

XML

```
<profiles>
   <publisher profile_name=
↪"publisher_xml_profile">
      <multicastLocatorList>
         <locator>
            <address>239.255.0.4</
↪address>
            <port>7900</port>
         </locator>
      </multicastLocatorList>
   </publisher>

   <subscriber profile_name=
↪"subscriber_xml_profile">
      <multicastLocatorList>
         <locator>
            <address>239.255.0.4</
↪address>
            <port>7900</port>
         </locator>
      </multicastLocatorList>
   </subscriber>
</profiles>
```

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. *eProsima Fast RTPS* currently supports UDPv4 or UDPv6
- `port`: Port as an UDP/IP port.
- `address`: Maps to IP address

Listening locators

eProsima 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. The 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.metatrafficUniicastLocatorList.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;

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 next predefined rules:

Table 7.1: Ports used

Traffic type	Well-known port expression
Metatraffic multicast	PB + DG * *domainId* + offsetd0
Metatraffic unicast	PB + DG * *domainId* + offsetd1 + PG * *participantId*
User multicast	PB + DG * *domainId* + offsetd2
User unicast	PB + DG * *domainId* + offsetd3 + PG * *participantId*

These predefined rules use some values explained here:

- DG: DomainId Gain. You can set this value using attribute `rtps.port.domainIDGain`. 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 to listen network messages.

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 *Configuration* section you could see how 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 contains 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* 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
onNewDataMessage	N	Y
onSubscriptionMatched	N	Y
onPublicationMatched	Y	N

CHAPTER 8

Writer-Reader Layer

The lower level Writer-Reader Layer of *eprosima Fast RTPS* provides a raw implementation of th 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:

```
//Request a change from the history
CacheChange_t* ch = writer->newCacheChange(ALIVE);
//Write serialized data into the change
ch->serializedPayload->length = sprintf(ch->serializedPayload->data, "My String %d",␣
↪2);
```

```
//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 two 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 lask *k* messages it has sent. If a new reader matches after message *n*, it will start receiving from message *n-k*

To choose you preferred option:

```
WriterAttributes Wparams;
Wparams.endpoint.durabilityKind = TRANSIENT_LOCAL;
```

Because in the Writer-Reader layer you have control over the History, in TRANSIENT_LOCAL mode the Writer send 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 they 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; //Dedaults to 0 = Unlimited Changes
```

When the initial amount of reserved changes is lower than the maximum, the History will allocate more changes as they are needed until it reaches the maximum size.

CHAPTER 9

Advanced Functionalities

This section covers slightly more advanced, but useful features that 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 consinst 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 Tuning Realiable 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 exchange 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.

9.3 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.

9.4 Sending large data

The default size *eProsima Fast RTPS* uses to create sockets 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. In the other hand, in reliable mode, the existence of a lot of data fragments could decrease the frecuency in which messages are received. If this happens, it can be resolved setting a lower Heartbeat period, as stated in *Tuning Realiable 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*

9.4.1 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 bandwith 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 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 bandwith. A reasonable flow controller for this application could be a limit of 5MB/s, which represents only a 5% of the total bandwith. 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"
```

9.4.2 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 appropiate to configure the reliability mode to best-effort.

9.5 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
my_transport->receiveBufferSize = 65536; //Configuration parameters
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.

9.6 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;
```

9.6.1 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</userId>
            <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</userId>
            <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 writes 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* or *TRANSIENT_LOCAL_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.

9.7 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.

9.8 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 bundles 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 it *support@eprosima.com*.

CHAPTER 10

Security

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

By default Fast RTPS doesn't compile security support. You can activate it adding `-DSECURITY=ON` at CMake configuration step. 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 matches 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 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*.

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 `ÈNCRYPT`.

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`.

10.3 Built-in plugins

Current version comes out with two security built-in plugins:

- **Auth:PKI-DH**: this plugin provides authentication using a trusted *Certificate Authority* (CA).
- **Crypto:AES-GCM-GMAC**: this plugin provides authenticated encryption using Advanced Encryption Standard (AES) in Galois Counter Mode (AES-GCM).

10.3.1 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 10.1: **Properties to configure Auth::PKI-DH**

Property name (all properties have "dds.sec.auth.builtin.PKI-DH." prefix)	Property value
identity_ca	URI to the X509 certificate of the Identity CA. Supported URI schemes: file. The **file** schema shall refer to a X.509 v3 certificate in PEM format.
identity_certificate	URI to 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 *(optional)*	URI to a X509 Certificate Revocation List (CRL). Supported URI schemes: file.
private_key	URI to access the private Private Key for the Participant. Supported URI schemes: file.
password *(optional)*	A password used to decrypt the private_key.

Generation of x509 certificates

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

Generate a certificate for the CA

Wether you want to create your own CA certificate, first you 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.
preserve = no # keep passed DN ordering

policy = policy_match

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

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

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

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

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

```
openssl ecparam -name prime256v1 > ecdsaparam

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

Generate a certificate for the Participant

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

```
# File: appconf.cnf

prompt = no
string_mask = utf8only
distinguished_name = req_distinguished_name

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

10.3.2 Crypto:AES-GCM-GMAC

This built-in plugin provides authenticated encryption using AES in Galois Counter Mode (AES-GCM). It also provide 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`.

10.3.3 Example

This example show you how to configure a Participant to activate and configure *Auth:PKI-DH* and *Crypto:AES-GCM-GMAC* plugins. Also it 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", "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 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.properties.properties().emplace_back("rtps.endpoint.submessage_protection_
↪kind", "ENCRYPT");

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

Reader attributes

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

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

CHAPTER 11

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.

11.1 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.

11.2 Where to find *fastrtpsgen*

If you are using the binary distribution of *eProsima Fast RTPS*, *fastrtpsgen* is already provided for you. If you are building from sources, you have to compile *fastrtpsgen*. You can find instruction in section.

CHAPTER 12

Introduction

eProsima FASTRTPSGEN is a Java application that generates source code using the data types defined in an IDL file. This generated source code can be used in your applications in order to publish and subscribe to a topic of your defined type.

To declare your structured data, you have to use IDL (Interface Definition Language) format. IDL is a specification language, made by OMG (Object Management Group), which describes an interface in a language-independent way, enabling communication between software components that do not share the same language.

eProsima FASTRTPSGEN is a tool that reads IDL files and parses a subset of the OMG IDL specification to generate serialization source code. This subset includes the data type descriptions included in *Defining a data type via IDL*. The rest of the file content is ignored.

eProsima FASTRTPSGEN generated source code uses Fast CDR: a C++11 library that provides a serialization mechanism. In this case, as indicated by the RTPS specification document, the serialization mechanism used in 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.

12.1 Compile

In order to compile *fastrtpsgen* we first need already have installed gradle and java JDK (please, check the JDK recommended version for the gradle version you have installed).

To generate *fastrtpsgen* we will need to add the argument `-DBUILD_JAVA=ON` when calling CMake.

CHAPTER 13

Execution and IDL Definition

13.1 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.

13.2 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

13.2.1 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

13.2.2 Sequences

fastrtpsgen fupports 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>

13.2.3 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 attributes plus method 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;
};
```

13.2.4 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;
};
```

13.2.5 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
};
```

13.2.6 Keyed Types

In order to use keyed topics the user should define some key members inside the structure. This is achieved by writting "@Key" before the members of the structure you want to use as keys. For example in the following IDL file the *id* and *type* field would be the keys:

```
struct MyType
{
    @Key long id;
    @Key string type;
    long positionX;
    long positionY;
};
```

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

13.2.7 Including other IDL files

You can include another IDL files in yours in order to use data types defined in them. *fastrtpsgen* uses a C/C++ preprocessor for this purpose, and you can use `#include` directive to include an IDL file.

```
#include "OtherFile.idl"
#include <AnotherFile.idl>
```

If *fastrtpsgen* doesn't find a C/C++ preprocessor in default system paths, you could specify the preprocessor path using parameter `-ppPath`. If you want to disable the usage of preprocessor, you could use the parameter `-ppDisable`.

CHAPTER 14

Version 1.5.0

This release include the following features:

- Configuration of Fast RTPS entities through XML profiles.
- Added heartbeat piggyback support.

Also bug fixing.

Note: If you are upgrading from an older version than 1.4.0, it is advisable to regenerate generated source from IDL files using *fastrtpsgen*

14.1 Previous versions

14.1.1 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 *fastrtpsgen*

14.1.2 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.

14.1.3 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.

14.1.4 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.