FastRTPS Documentation

Release 1.3.1

eProsima

November 23, 2016

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.

This documentation is organized into the following sections:

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

CHAPTER 1

Requirements

eProsima Fast RTPS requires the following packages to work.

1.1 Common Dependencies

1.1.1 Boost Libraries

The Windows installer version provides the necessary boost binaries and headers. If you are using Linux or compiling from source you will need to install version **1.61**

1.1.2 Gtest

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

1.1.3 Java

Java is required to make use of our buil-in code generation tool *fastrtpsgen*.

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 realese 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 preffered 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
```

If you are on Linux, execute:

```
$ cmake ../ -DEPROSIMA_BUILD=ON -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=install
$ make
$ make install
```

If you are on Windows, choose your version of Visual Studio:

```
> cmake ../  -G"Visual Studio 14 2015 Win64" -DEPROSIMA_BUILD=ON -DCMAKE_BUILD_TYPE=Release -DCMAKE_I
> cmake --build . --target install
```

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

CHAPTER 4

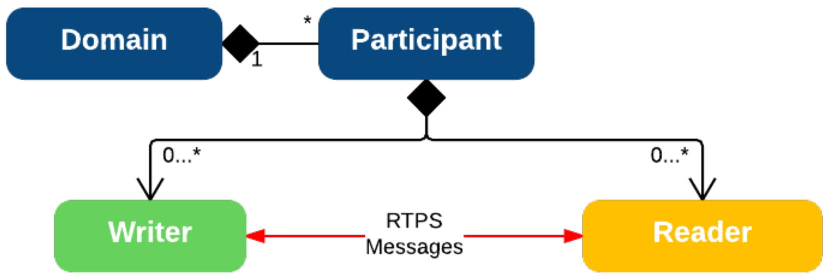
Getting Started

4.1 Brief introduction to the RTPS protocol

At the top of RTPS we find the Domain, which defines a separate plane of communication. 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.

4.2 Building your first application

To build a minimal application, you must first define the topic. 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, 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.message("HelloWorld");
mp_publisher->write((void*)&myHelloWorld);
```

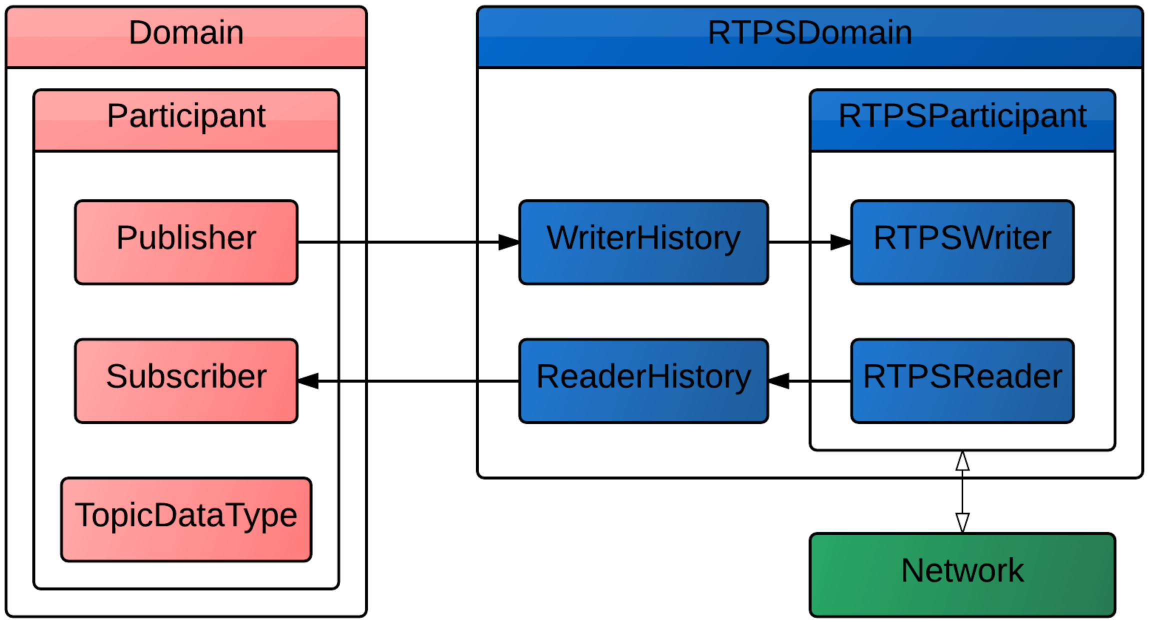
Take a look at the *examples/* folder for ideas on how to improve this basic application through different configuration options, and for examples of advanced Fast RTPS features.

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 for 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 Pparam; //Configuration structure
Participant *mp_participant = Domain::createParticipant(Pparam);
```

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*. Once again, this is done by using the `Domain` class:

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

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

```
PublisherAttributes Wparam; //Configuration structure
PubListener m_listener; //Class that implements callbacks from the publisher
Publisher *mp_publisher = Domain::createPublisher(mp_participant, Wparam, (PublisherListener *)&m_lis
```

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
mp_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 
    }
} m_listener;
```

The `Subscriber` creation and implementation is symmetric.

```
SubscriberAttributes Rparam; //Configuration structure
SubListener m_listener; //Class that implements callbacks from the Subscriber
Subscriber *mp_subscriber = Domain::createSubscriber(mp_participant,Rparam,(SubsciberListener*)&m_lis
```

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

7.2 Participant Configuration

The `Participant` is configured via the `ParticipantAttributes` structure. We will now go over the most common configuration options.

7.2.1 Setting the name and Domain of the Participant

The name of the `Participant`, which forms part of the meta-data of the RTPS protocol, can be changed from within the `ParticipantAttributes`:

```
Pparam.setName("my_participant");
```

Publishers and Subscribers can only talk to each other if their Participants belong to the same DomainId. To set the Domain:

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

7.2.2 Defining Input and Output channels

In the RTPS Standard, input-output channels (sockets) are defined 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.

You can specify a default list of channels for the Publishers and Subscribers by passing lists of Locators to the configuration structure

```
Locator_t loc1.set_IP4_address(127,0,0,1);
loc1.port = 22222;
Pparam.rtps.defaultUnicastLocatorList.push_back(loc1); //Input for Unicast messages
Locator_t loc2.set_IP4_address(127,0,0,1);
loc2.port = 33333;
Pparam.rtps.defaultMulticastLocatorList.push_back(loc2); //Input for Multicast messages
```

```
Locator_t loc3.set_IP4_address(127,0,0,1);
loc3.port = 44444;
Pparam.rtps.defaultOutLocatorList.push_back(loc3); //Output Channels
```

If no Locators are specified by the user *eprosima Fast RTPS* will calculate a minimalistic set to ensure correct behaviour.

7.3 Publisher and Subscriber Configuration

Publishers and Subscribers inherit traits from the configuration of their Participant. You can override these traits by providing different values at their configuration structures. For example, you can specify a different set of Locators for a Publisher to use:

```
Locator_t my_locator;
//Initialize the Locator
SubAttr.unicastLocatorList.push_back(my_locator);
```

7.3.1 Setting the name and data type of the topic

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

```
// Topic name and type for Publisher
Wparam.topic.topicDataType = "HelloWorldType"
Wparam.topic.topicName = "HelloWorldTopic"
// Topic name and type for Subscriber
Rparam.topic.topicName = "HelloWorldTopic"
Rparam.topic.topicDataType = "HelloWorldType"
```

7.3.2 Reliability Kind

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.

You can specify which mode you want to use in the publisher or subscriber parameters:

```
Wparam.qos.m_reliability.kind = RELIABLE_RELIABILITY_QOS; //Set the publisher to Realiable Mode
Rparam.qos.m_reliability.kind = BEST_EFFORT_RELIABILITY_QOS; //Set the subscriber to Best Effort
```

Keep in mind that different reliability mode configurations will make a Publisher and a Subscriber incompatible and unable to talk to each other. Read more about this in the [Built-In Prococols](#built-in-procotols) section.

7.3.3 Configuring sample storage

eProsima Fast RTPS provides two policies for sample storage:

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

The mode is changed the following way:

```
Wparam.topic.historyQos.kind = KEEP_LAST;
```

The depth of the stored samples in keep-last mode is changes in the following way:

```
Wparam.topic.historyQos.depth = 5; //changes the maximum number of stored samples
```

The 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.

To set the durability mode:

```
Rparam.qos.m_durability.kind = VOLATILE_DURABILITY_QOS;
```

It is also possible to control the maximum size of the History:

7.4 Additional Concepts

7.4.1 Built-In Protocols

Before a Publisher and a Subscriber can exchange messages, they must be matched. The matching process is performed by the built-in protocols.

The RTPS Standard defines two built-in protocols that are used by Endpoints to gain information about other elements in the network

- Participant Discovery Protocol (PDP): Used by the Participant to gain knowledge of other Participants in the network.
- Endpoint Discovery Protocol (EDP): Used to gain knowledge of the endpoints (Publishers and Subscribers) a remote Participant has.

When a local and a remote endpoint have the same topic name, data type and have compatible configuration they are matched and data posted by Publisher is delivered to the Subscriber. When a Publisher posts data it is sent to all of its matching Subscribers. This includes the exchange arrival confirmation messages from both parties in the case of Reliable Mode.

As an user, you can actually interact with the way the Built-in protocols behave. To learn more, go to the [Advanced Topics](#Advanced Topics) section.

7.4.2 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
}
```

7.4.3 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,hist);
```

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,hist,&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:

```
CacheChange_t* ch = writer->newCacheChange(ALIVE); //Request a change from the history
ch->serializedPayload->length = sprintf(ch->serializedPayload->data,"My String %d",2); //Write seria
history->add_change(ch); //Insert change back into the history. The Writer takes care of the rest.
```

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)
    {
        printf("%s\n",change->serializedPayload.data); // The incoming message is enclosed within th
        reader->getHistory()->remove_change((CacheChange_t*)change); //Once done, remove the change
    }
}
```

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

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

As in the Publisher-Subscriber Layer, you can specify the input and output channels you want your Writer/Reader to listen to or speak from in the form of Locators. This configuration overrides the one inherited from the `RTPSParticipant`.

```
WriterAttributes Wattr;
Locator_t my_locator;
//Set up your Locator
Wattr.endpoint.OutLocatorList.push_back(my_locator);
```

8.3.2 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:

```
//Publicher-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
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 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.

9.3 Tuning Realiable mode

RTPS protocol uses Heartbeat messages to make matched endpoints exchange meta-data on what pieces of data they hold so the missing ones can be re-sent (on Reliable mode of course). You can modify the frequency of this meta-data 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 = 500; //500 ms
```

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

```
WriterAttributes Wattr;
Wattr.times.heartbeatPeriod.seconds = 0;
Wattr.times.heartbeatPeriod.fraction = 500;
```

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.

9.4 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
WparamSlow.terminalThroughputController = 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.

9.5 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 fo
my_transport->receiveBufferSize = 65536; //Configuration parameters
my_transport->granularMode = false;
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 is in folder *examplesUserDefinedTransportExample*

9.6 Matching endpoints the manual way

By default, when you create a Participant or a RTPS Participant the built-in protocols for automatic discovery of endpoints will be active. You can disable them by configuring the Participant:

```
ParticipantAttributes Pparam;
Pparam.rtps.builtin.use_SIMPLE_EndpointDiscoveryProtocol = false;
Pparam.builtin.use_SIMPLE_RTPSParticipantDiscoveryProtocol = false;
```

If you disable the built-in discovery protocols, you will need to manually match Readers and Writers. To inform a Writer about a remote Reader, you can either provide an XML configuration file or use the :class::*RemoteReaderAttributes* structure:

```
RemoteReaderAttributes ratt;
Locator_t loc; //Add the locator that represents a channel the Reader listens to
loc.set_IP4_address(127,0,0,1);
loc.port = 22222;
ratt.endpoint.unicastLocatorList.push_back(loc)
ratt.guid = c_Guid_Unknown; //GUID_t is left blank, but must be configured when using Reliable Mode.
writer->matched_writer_add(ratt);
```

Registering a remote Writer into a Reliable mode Reader works the same way:

```
RemoteWriterAttributes watt;
//Configure watt
reader->matched_reader_add(watt);
```

If you decide to provide the information via XML, you have to specify the file where you want to load from:

```
participant_attributes.rtps.builtin.use_STATIC_EndpointDiscoveryProtocol = true;
participant_attributes.rtps.builtin.setStaticEndpointXMLFilename("my_xml_configuration.xml");
```

You can use this sample XML as a base for building your configuration files:

```
<staticdiscovery>
    <participant>
        <name>RTPSParticipant</name>
        <reader>
```

```
            <userId>3</userId>
            <entityId>4</entityId>
            <expectsInlineQos>false</expectsInlineQos>
            <topicName>TEST_TOPIC_NAME</topicName>
            <topicDataType>HelloWorldType</topicDataType>
            <topicKind>NO_KEY</topicKind>
            <reliabilityQos>RELIABLE_RELIABILITY_QOS</reliabilityQos>
            <unicastLocator
                address="127.0.0.1"
                port="31377">
            </unicastLocator>
            <multicastLocator
                address="127.0.0.1"
                port="31378">
            </multicastLocator>
            <durabilityQos>TRANSIENT_LOCAL_DURABILITY_QOS</durabilityQos>
        </reader>
    </participant>
</staticdiscovery>
```

9.7 Subscribing to Discovery Topics

As specified in the Built-In protocols 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;
   tempMsg.msg_endian = change->serializedPayload.encapsulation ==
     PL_CDR_BE ? BIGEND : LITTLEEND;
   tempMsg.length = change->serializedPayload.length;
   memcpy(tempMsg.buffer, change->serializedPayload.data, tempMsg.length);
   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

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 topic to a working implementation of the methods needed to create publishers and subscribers of that 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.

10.1 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.

10.2 Compilation

If you are using the binary distribution of *eProsima Fast RTPS*, *fastrtpsgen* is already compiled for you. If you are building from sources, you will need to compile the tool manually using 'gradle'

CHAPTER 11

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 section 3. The rest of the file content is ignored.

eProsima FASTRTPSGEN generated source code uses a relevant component: 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)1 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. This document presents a brief guide of how to use eProsima FASTRTPSGEN. The structure of the document is presented below:

CHAPTER 12

Execution and IDL Definition

12.1 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.

12.2 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
boolean	bool
string	std::string

12.2.1 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

12.2.2 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>

12.2.3 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;
};
```

12.2.4 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;
};
```

12.2.5 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
};
```

12.2.6 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.

CHAPTER 13

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.

13.1 Previous versions

13.1.1 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.

13.1.2 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.