Fast RTPS v1.2.0

User Manual

eProsima
Proyectos y Sistemas de Mantenimiento SL
Ronda del poniente 16 – Bajo K
28760 Tres Cantos Madrid
Tel: + 34 91 804 34 48
info@eProsima.com – www.eProsima.com

Technical Support

- Phone: +34 91 804 34 48
  Email: Support@eProsima.com

Contents

Introduction

What is RTPS?

Real-Time Publish-Subscribe (RTPS) is the wire interoperability protocol defined for the **Data Distribution Service (DDS)** standard by the **Object Management Group (OMG)** consortium. It allows different implementations of the same communication structure to interoperate. This protocol standard is defined by the OMG in the specification document *"The Real-Time Publish-Subscribe Wire Protocol, DDS Interoperability Wire Protocol Specification (DDS-RTPS)"*.

RTPS enables publisher-subscriber communications over unreliable transports such as UDP. It supports *Unicast* and *Multicast* with *best-effort* and *reliable* communication models.

Since RTPS serves as the wire protocol for the DDS (Data Distribution Service) standard, the concepts of RTPS and DDS are closely related.

What is eProsima Fast RTPS?

eProsima Fast RTPS is a standalone C++ implementation of the last release of RTPS standard.

Fast RTPS includes:

- RTPS API to use the protocol and its features directly.
- Simple Publication-Subscription API: A subset of DDS.
- OMG IDL compiler: It generates Type Support for your types and Pub/Sub example code.

Some key features are:

- Configurable communication policies for real-time applications, with different levels of reliability.
- Automatic discovery of new participants by other members of the network.
- Modularity and scalability.
- Network agnosticism: ability to use different transport layers and easily implement your own.
- Fast concurrency through an efficient multithreaded model.

Reference Material

The following documents will be useful as you learn to use Fast RTPS:

- OMG RGPS Specification.
- OMG DDS Specification (To understand examples that operate at this level).
- eProsima Fast RTPS API.
- fastrtpsgen Manual.

Document Organization

The next section of this guide explains how to use Fast RTPS through practical examples. The rest consists of a high level view of the library, supporting the more detailed API reference.

- Section 2 (Getting Started): Introduction to RTPS concepts and a simple application example.
- Section 3 (Library Overview): A first look into the four main Fast RTPS modules.
- Section 4 (Objects and Data Structures): Fast RTPS objects and how they relate to the standard.
- Section 5 (Publisher - Subscriber Interface Layer): Description of the Publisher-Subscriber layer and examples of its use.
- Section 6 (Writer - Reader Layer): For a finer control of the lower communication layers.
- Section 7 (Automatic code generation): Brief look into the **fastrtpsgen** code generation tool

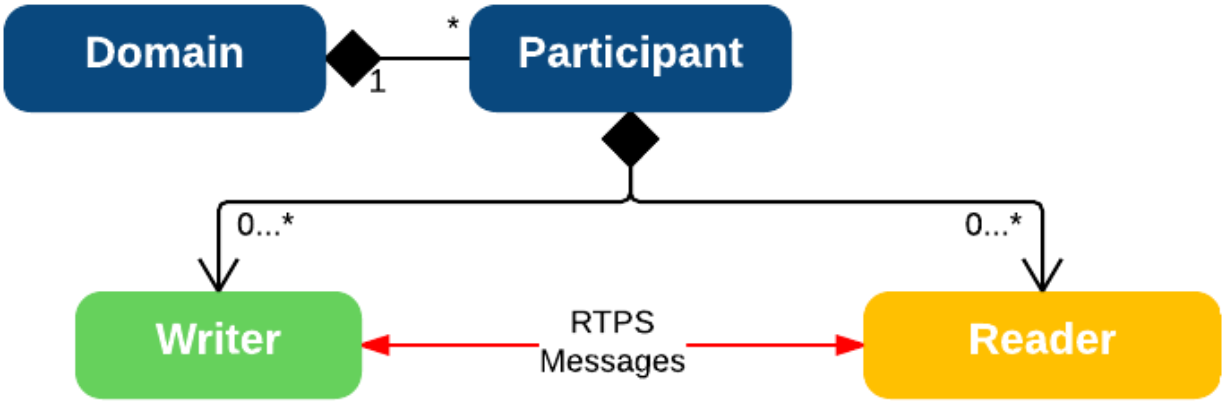
Getting Started

Introduction to the Real Time Publish Subscribe (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.

Building your first Application

To build a minimal application, you must first define the *topic*. You do this by writing an **IDL (Interface Definition Language)** file:

```
// HelloWorld.idl: A simple topic containing a string.

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 for details, but for now the following commands will do:

Linux	`fastrtpsgen -example x64Linux2.6gcc HelloWorld.idl`
Windows	`fastrtpsgen.bat -example x64Win64VS2015 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.

To invoke a subscriber:

Linux	`./HelloWorldPublisherSubscriber subscriber`
Windows	`HelloWorldPublisherSubscriber.exe subscriber`

To invoke a publisher:

Linux	`./HelloWorldPublisherSubscriber publisher`
Windows	`HelloWorldPublisherSubscriber.exe publisher`

Each time you press <Enter> on a *publisher*, a new datagram is generated, sent over the network and receiver by Subscribers currently online. In case you spawned multiple *subscribers*, you can see all of them receive the data.

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.

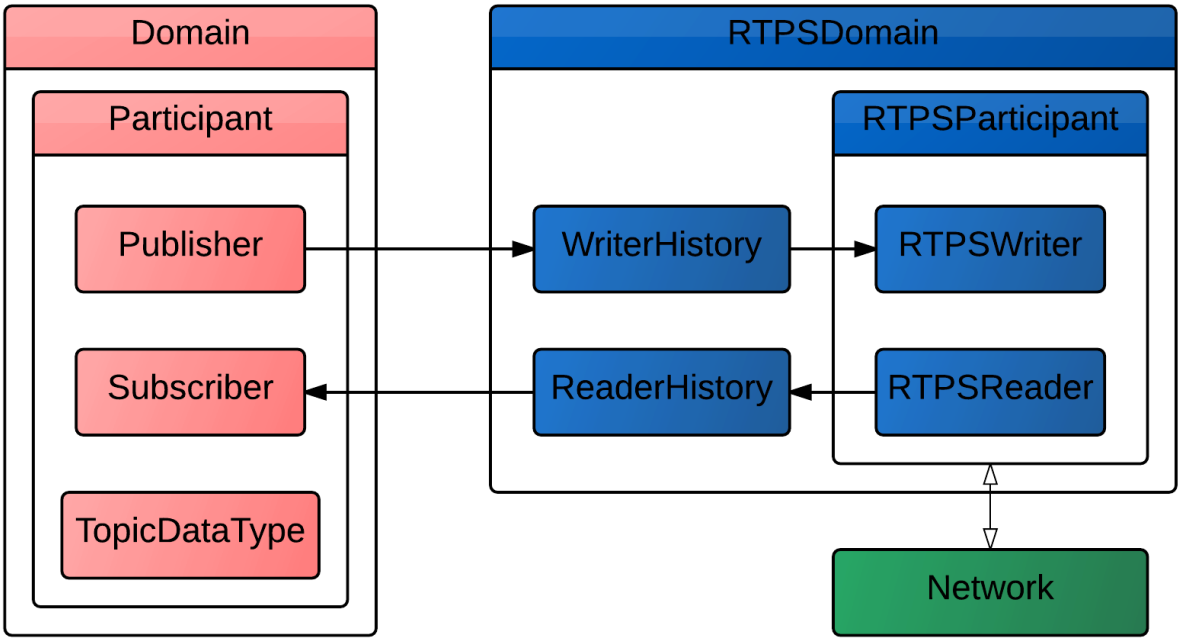
Library Overview

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 section 2 (Getting Started), 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.



Fast RTPS architecture

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).

Events

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

Objects and Data Structures

eProsima Fast RTPS objects are classified by modules. Take a look at the API modules for further details.

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.

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:

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.

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.

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.

RTPS Writer

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

Publisher - Subscriber Layer

High level abstraction over RTPS concepts.

This layer allows you to set up an application quickly and easily, without having to manage RTPS entities directly. You still have the option to interact with the layers below, even if you choose to work at this level.

Domain

The *Domain* class manages creation of *Participants*, *Publishers* and *Subscribers*, and registers topic data.

Participant

A participant stores *Publishers*, *Subscribers* and *Topic Data Types*. It also propagates part of its configuration to its Publishers and Subscribers.

ParticipantAttributes contains all configuration parameters for a participant. There is one you must always define:

- **DomainId**: This attribute is used to calculate the discovery ports and it is important to separate different applications working in the same network.

```
// First, we define the configuration options for our Participant.
ParticipantAttributes PParam;
Pparam.rtps.setName("participant");
Pparam.rtps.builtin.domainId = 80;

// We construct a participant in the domain.
Participant* p = Domain::createParticipant(PParam);

...

// Last, we eliminate the participant
Domain::removeParticipant(p);
```

Constructing and destroying a participant.

Publisher

There are two main things you can do with a publisher:

- Send (publish) data to any subscribers.
- Receive a callback on matching with a subscriber.

Publishers are configured via an instance of the **PublisherAttributes** structure, which is used by the *Domain* during creation. To define a callback, you must derive from the **PublisherListener** class.

Publisher Configuration

The configuration options in *PublisherAttributes* are divided into multiple groups. The most commonly used options under each group are:

- **PublisherAttributes.topic.topicKind: WITH_KEY** or **NO_KEY** as defined in the RTPS standard. It must correspond to the option you chose for your topic data type.
- **PublisherAttributes.topic.topicName*:*** Name of topic.
- **PublisherAttributes.topic.topicDataType:** Data type of the topic it publishes to.
- **PublisherAttributes.topic.historyQos:** Quality of service options for the history cache.
- **PublisherAttributes.Qos.m_publishMode:** Synchronous or asynchronous communication.
- **PublisherAttributes.times.heartbeatPeriod:** Period of **HEARTBEAT** messages as defined in the RTPS standard. Reducing the heartbeat period can increase performance for lossy networks.
- **PublisherAttributes.times.nackResponseDelay:** Delay before replying to ***ACKNACK*** messages as defined in the RTPS standard.
- **PublisherAttributes.unicastLocatorList:** Defines the unicast input channels the *Publisher* listens to.
- ***PublisherAttributes.multicastLocatorList*:** Defines the multicast input channels the *Publisher* listens to.

Note: If you don't specify any unicast or multicast locators, Fast RTPS will automatically provide UDPv4 defaults (a multicast and a unicast address).

Publisher Callback

You can define a callback for your publisher, that will trigger when it matches with a subscriber. To do that, you must create a class derived from PublisherListener and pass it to the Domain when constructing the publisher.

```
class MyListener: public PublisherListener
{
    void onPublicationMatched(Publisher* pub, MatchingInfo& info)
    {
        if(info.status == MATCHED_MATCHING)
       cout << "Publication Matched" << endl;
    }
};

int main()
{
    // ... create a participant and define your Publisher Attributes
    MyListener myListener;
    auto* myPublisher = Domain::createPublisher(myParticipant,
                                                myPublisherAttributes,
                                                &myListener);
}
```

Subscriber

There are two main things you can do with a subscriber:

- Receive a callback when the subscribed topic is updated by a publisher.
- Receive a callback on matching with a publisher.

Subscribers are configured via an instance of the **SubscriberAttributes** structure, which is used by the *Domain* during creation.

Subscriber configuration

Basic configuration options are similar to the ones described in the Publisher Section. Consult the API documentation for details on more advanced configuration.

Subscriber Callbacks

You can define the subscriber callbacks, you must create a class derived from **SubscriberListener** and pass it to the *Domain* when constructing the subscriber.

```
class MyListener: public SubscriberListener
{
    void OnSubscriptionMatched(Subscriber* sub, MatchingInfo& info)
    {
        if(info.status == MATCHED_MATCHING)
       cout << "Subscription Matched" << endl;
    }
    void OnNewDataMessage(Subscriber* sub)
    {
      if(sub->takeNextData((void*)&m_myData, &m_info))
          cout << "Message "<<m_Hello.message()<< " RECEIVED"<<endl;
    }

};
```

SubscriberListener example

Writer - Reader Layer

Use this layer to control RTPS objects directly.

Working with the **Writer - Reader layer** directly is more complex than dealing with the simpler Publisher - Subscriber abstraction, but it allows for finer control, and maps more directly to concepts defined in the RTPS standard.

RTPSDomain

Similar to the Domain class in the Publisher - Subscriber layer, **RTPSDomain** manages creation and destruction of **RTPSParticipants** (the lower level equivalent of the Participant) as well as **RTPSReader** and **RTPSWriter** endpoints.

RTPSParticipant

A participant stores *RTPSWriters*, *RTPSReaders,* and It propagates part of its configuration to its writers and readers.

To create a RTPSParticipant, you must call RTPSDomain::**createRTPSParticipant** and pass it a **RTPSParticipantAttributes** object, which contains all necessary configuration.

```
RTPSParticipantAttributes PParam;
Pparam.setName("participant");
Pparam.builtin.domainId = 80;
Pparam.use_IP6_to_send = false;

RTPSParticipant* participant = RTPSDomain::createRTPSParticipant(PParam);

//To remove:
RTPSDomain::removeParticipant(participant);
```

RTPSParticipant construction with simple parameters

Endpoint registration

You must manually register any Writer or Reader endpoints with your RTPSParticipant:

```
RTPSWriter* writer;
TopicAttributes tA("topicName","topicType",NO_KEY);
WriterQos wqos;

... //Change QoS settings

participant->registerWriter(writer,tA,wqos);
...
participant->updateWriter(writer,wqos);
```

This allows endpoints to be created with with minimal configuration, even less than the one needed for registration with the built-in protocols.

RTPSWriter

Writer History

Before an RTPSWriter sends data to any readers, the *change* to the topic data must be stored on a **history** data structure, that will keep track of the latest changes. When working at the Writer - Reader level, you can interact with the history structures directly.

```
HistoryAttributes hatt;
WriterHistory * history = new WriterHistory(hatt);
WriterAttributes watt;
RTPSWriter* writer = RTPSDomain::createRTPSWriter(rtpsParticipant,watt,hist);
```

Creating a writer while keeping a reference to the history

Writer Attributes

There are several parameters you can set to define the behaviour of a Writer. Here are the most common:

- ***WriterAttributes.endpoint.reliabilityKind***: This parameter can be set to *RELIABLE* (default) or *BEST_EFFORT*. Reliable writers will request feedback from the reader, and attempt to resend any lost data.
- ***WriterAttributes.endpoint.topicKind*:** This parameter can be set to *NO_KEY* (default) and *WITH_KEY.* For *WITH_KEY* writers, you must provide the *KEY* of the CacheChange_t before adding it to the WriterHistory. Keys allow for multiple separate instances of the same topic.
- ***WriterAttributes.endpoint.InputLocatorLists***: There are two locator lists associated with an *RTPSWriter* (unicast and multicast). This list is only used in Reliable mode, when the *RTPSWriter* is interested in listening to the matched Readers.
- ***WriterAttributes.endpoint.OutputLocatorList*:** Defines the output communication channels for the writer.
- ***WriterAttributes.times***: Defines the timing of events in RELIABLE mode. For example, the *heartbeat* period.

If you don't define any input locators, they will be inherited from the parent RTPSParticipant.

Writer History Attributes

You can configure the writer history independently through the **HistoryAttributes** object. Here are the most common parameters you will find in it:

- ***HistoryAttributes.payLoadMaxSize**:* Maximum size for a single change, defaults to 500 bytes.
- ***HistoryAttributes.initialReserverCaches***: Number of initially reserved caches (slots for changes). Defaults to 500.
- ***HistoryAttributes.MaximumReservedCaches*:** Maximum number of caches to allow to be reserved at any point. Defaults to 0, which means unlimited changes.

We recommend you give this last parameter a value, to prevent the cache from growing indefinitely in case of a spike in network traffic.

Sending Data with a RTPSWriter

You can send data by introducing a Change to the *WriterHistory*.

```
// The writer creates an empty change for us.
CacheChange_t* ch = writer->newCacheChange(ALIVE);

// We fill it with our data.
ch->serializedPayload->length = sprintf(ch->serializedPayload->data,"My String
%d",2);

// We place it in the Writer History to get it ready to send.
history->add_change(ch);
```

Matched readers

When a change is added its contents are sent to all **matched** readers. These readers were either discovered via the built-in protocols or added manually by the user.

Note that there are some configurations that make *Writers* and *Readers* incompatible and therefore unable to match: A *Reliable Writer* cannot have a match with a *Best-Effort* readers and Reliable *Reader* can not be a match of a *Best-Effort Writer*.

When matching manually, there is certain information an *RTPSWriter* needs to know about its matched readers. This information is contained in the *RemoteReaderAttributes* structure:

```
RemoteReaderAttributes ratt;
Locator_t loc;
loc.set_IP4_address(127,0,0,1);
loc.port = 22222;
ratt.endpoint.unicastLocatorList.push_back(loc)
ratt.guid = c_Guid_Unknown; //For Realiable Writers, you actually need the
GUID_t
writer->matched_writer_add(ratt);
```

RTPSReader

Reader History

When receiving a change from a writer, the reader stores it in a **ReaderHistory** data structure. When working at the Writer - Reader level, you can interact with the history structures directly.

```
class MyReaderListener:public ReaderListener;
MyReaderListener listen;
HistoryAttributes hatt;
ReaderHistory * history = new ReaderHistory(hatt);
ReaderAttributes ratt;
RTPSReader* reader = RTPSDomain::createRTPSReader(rtpsParticipant,
                                                  watt,
                                                  hist,
                                                  &listen);
```

ReaderAttributes

The ***ReaderAttributes*** structure is equivalent to WriterAttributes.

ReaderHistoryAttributes

The *ReaderHistory* element is configured with the exact same structure we used for the writer.

Reading data from a RTPSReader

To read received changes, you can use the **waitForUnreadMessage**() and **nextUnreadCache**() methods on the reader. Later, **remove_change()** will clear it from the history cache.

```
// Blocks until a message is received
reader->waitForUnreadMessage();

// Reads a change.
CacheChange_t* change;
if(reader->nextUnreadCache(&change))

//Removes change from the history cache.
history->remove_change(change);
```

Receving data with a ReaderListener

Data from a *RTPSReader* can also be received by using a class derived from **RTPS*ReaderListener***. A*n RTPSReaderListener* calls the user back when a *change* is received.

```
class MyReaderListener: public ReaderListener{
   public:
   MyReaderListener(){}
   ~MyReaderListener(){}
   void onNewCacheChangeAdded(RTPSReader* reader,const CacheChange_t* const
change){
        printf("%s\n",change->serializedPayload.data);
        reader->getHistory()->remove_change((CacheChange_t*)change);
   }
}
```

An RTPSReader is limited to having one ReaderListener attached, and therefore supports one callback.

Callbacks on builtin readers

It is possible for the user to attach his own listener to the *Endpoint Discovery Protocol (EDP) RTPSReaders.*

```
CustomReaderListener *my_readerListenerSub = new(CustomReaderListener);
CustomReaderListener *my_readerListenerPub = new(CustomReaderListener);
Std::pair<StatefulReader*,StatefulReader*> EDPReaders =
                                    my_participant->getEDPReaders();
EDPReaders.first()->setListener(my_readerListenerSub);
EDPReaders.second()->setListener(my_readerListenerPub);
```

Transport Layer and Network configuration

Fast RTPS allows you to use different **transport layers** in the same application, even for the same endpoint.

Fast RTPS will load an UDPv4 layer by default. It is possible to enable UDPv6, and also to define your own transport layer by implementing the class **TransportInterface**.

Before configuring a participant, you can supply the following network options:

- **Input channels**: Input points where Readers and Writers receive datagrams from.
- **Output channels:** Output points the Readers and Writers use to send data.
- **Transport layers to use**: Any number of transport layers.

Remember that these input and output channels will be propagated to the Writers and Readers, unless you specify a different set when creating them.

Configuring Transport Layers

If you want to disable the default UDPv4 transport, you can do it through an option in the participant parameters:

```
RTPSParticipantAttributes Pparams;
testTransport->useBuiltinTransports = false;
```

Whether you disable it or not, you may also want your *participant* to use any number of different transport layers. Transports (that is, classes that derive from *TransportInterface*) have a structure associated with them called **TransportDescriptor**, that contains all necessary configuration.

To add an UDPv6 transport to a participant, you would have to store a **UDPv6TransportDescriptor** in the participant attributes:

```
RTPSParticipantAttributes Pparams;
auto myTransport = std::make_shared<UDPv6Transport::TransportDescriptor>();

// We change any UDPv6 defaults to our desired values.
testTransport->receiveBufferSize = 65536;

// By doing this, the participant constructed with these attributes will
support UDPv6.
Pparams.userTransports.push_back(myTransport);
```

examples/C++/UserDefinedTransportExample/ walks through the process of replacing the default layer with a differently configured transport.

Automatic code generation

ePr*osima Fast RTPS* includes a code generation tool, fastrtpsgen*,* which translates an *IDL* specification of a topic to C++ data structures compatible with Fast RTPS.

fastrtpsgen can also generate a sample application using this data type, providing a Makefile to compile it on Linux and a Visual Studio project for Windows.

FASTRTPSGEN use

fastrtpsgen can be invoked by calling *fastrtpsgen* on Linux or *fastrtpsgen.bat* on Windows.

```
fastrtpsgen.bat -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.

FASTRTPSGEN output

fastrtpsgen outputs 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. You must fill the needed methods for your 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.

Executable use

The generated example produces a single executable, which you can run as a Publisher or a Subscriber through the “publisher” and “subscriber” command line arguments. *eProsima Fast RTPS* is known to be flagged as suspicious activity by some popular Windows based antivirus firewalls. We recommend you configure a special firewall rule when working with the library.

Advanced Topics

eProsima Fast RTPS offers several ways to configure all modules covered in this guide. This section offers an overview of the more advanced options you can choose in demanding applications.

User defined QoS policies

The **examples/C++** folder includes a collection of Quality of Service policies implemented on top of Fast RTPS. If you need to offer guarantees at a higher level than what is covered in this guide, these examples can be a good place to start.

Time-Based Filter and Content-Based Filter

This subscriber rejects any data that does not clear two different filters:

- **passTimeFilter**: Accepts a single sample within a given time period. In case the topic is updated more frequently than what the subscriber requires, this allows the subscriber to discard the excess.
- **passContentFilter**: The data must conform to some criteria based on its content. For example, in case of a Radar Track topic, we could use the track coordinates to filter.

Ownership Strength

The Ownership Strength example implements a strength classification for publishers. Subscribers store these strength values associated to the publisher IDs, and reject any incoming changes that don't belong to the strongest Publisher.

When a publisher becomes unmatched, the strength hierarchy is updated by removing the entry corresponding to that publisher, so the next strongest one can get priority.

Deadline

The Deadline QoS policy guarantees a minimum frequency of received messages. If the reception frequency falls below a threshold for a given topic and key, a callback function is called.

Large Data

When defining your custom topic data, the resulting structure may not fit in a single UDP packet, or it may be too big for one or more of the transport layers in your application.

Fast RTPS is able to fragment large data types and reconstruct them at the receiving end. This requires the publisher to be configured in asynchronous mode.

```
// Our data type is large and we require fragmentation, so...

PublisherAttributes Wparam;

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

In principle, you don't need to configure the subscriber to receive fragmented data, it will do it automatically. However, it must be prepared to service the input socket fast enough.

There are two ways to achieve this:

- Apply a throughput controller on the publisher, to reduce bandwidth and allow the subscriber to keep up. This will stop large data bursts from overflowing the receiving socket.
- Enable reliable QoS. This will allow fragments to be re-sent individually even in case the subscriber is not able to keep up.

Flow Control

eProsima Fast RTPS provides a way to control network traffic via Flow Controller objects. In particular, the **Throughput Controller** places a limit on network bandwidth.

Every *Participant* and every *RTPSWriter* contains a built-in throughput controller. This allows you to limit bandwidth at two levels, for example placing a short-term restriction on your *writer* to prevent overflowing the *readers*, and a more relaxed limit for the entire *participant*.

These controllers are inactive by default (i.e. they allow an unlimited amount of traffic). To enable one, you must give it values through a **ThroughputControllerDescriptor**:

```
WriterAttributes Wparams;

// This controller will allow ~300kb per second.
ThroughputControllerDescriptor myThroughputController{300000, 1000};
Wparams.throughputController = myThroughputController;
```

The steps required for the *participant throughput controller* are identical, substituting *WriterAttributes* for *ParticipantAttributes*.

The *ThroughputControllerDescriptor* contains two fields:

- **bytesPerPeriod**: The amount of data allowed through this controller in a period of time.
- **periodMillisecs**: length of time during which no more than *size* bytes will be allowed.

Flow control is done using a sliding window approach. This guarantees that the restriction holds for **any** span of time; the size/period data rate will be kept for any window of ***period*** milliseconds.

Take a look at **examples/C++/FlowControlExample/** to see throughput controllers in action.

Manual Type Definition

We have looked at the *fastrtpsgen* tool, for a simple way to define topic data types and translate them to Fast RTPS compatible classes. However, you have the option to define a type manually, for example to have more control over serialization and deserialization.

Here's how you would define a topic type manually, for a topic encapsulating a single integer value:

```
class MyDataType: public TopicDataType
{
public:
    MyDataType():
       m_myData(0),
       m_typeSize(sizeof(m_myData)),
       m_isGetKeyDefined(false)
    {
        setName("MyType");
    }
    bool serialize(void*data,SerializedPayload_t* payload);
    bool deserialize(SerializedPayload_t* payload,void * data);

    typedef int RawData;
};
```

MyDataType.h

And the implementation:

```
bool MyDataType::serialize(void*data, SerializedPayload_t* payload){
    MyDataType::RawData* myData= (MyDataType::RawData*)data;
    payload->length = sizeof(MyDataType::RawData);
    memcpy(payload->data, myData, sizeof(MyDataType::RawData));
    return true;
}
bool MyDataType::deserialize(SerializedPayload_t* payload, void * data){
     MyDataType::RawData* myData= (MyDataType::RawData*)data;
     memcpy(myData,payload->data, sizeof(MyDataType::RawData);
     return true;
}
```

MyDataType.cpp

Finally, use the Domain static method below to register the type associated with a participant:

```
int main()
{
Participant* part;

// … Construct a participant (see Domain::createParticipant)

TestTypeDataType TestTypeData;
Domain::registerType(part,(TopicDataType*)&TestTypeData);

//...
}
```

Registering a type.