MicroXRCE-DDS Documentation

Release 2.1.1

eProsima

Sep 28, 2022

INSTALLATION MANUAL

1 Main Features 3

2 Installation 5

3 User manual 7

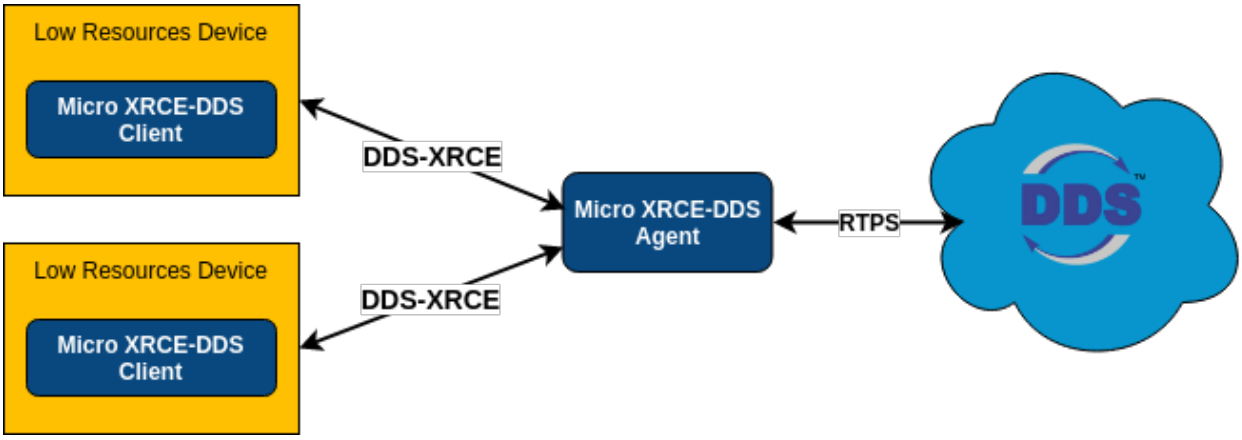
4 eProsima Micro XRCE-DDS Gen 9

5 Structure of the documentation 11

eProsima Micro XRCE-DDS is a software solution that allows communicating eXtremely Resource Constrained Environments (XRCEs) with an existing DDS network. This implementation complies with the specification of the eXtremely Resource Constrained Environments DDS (DDS-XRCE) protocol as defined and maintained by the Object Management Group (OMG) consortium.

The *eprosima Micro XRCE-DDS* library implements a client-server protocol that enables resource-constrained devices (clients) to take part in DDS communications. The *eProsima Micro XRCE-DDS Agent* (server) acts as a bridge to make this communication possible. It acts on behalf of the *Micro XRCE-DDS Clients* by enabling them to take part to the DDS Global Data Space as DDS publishers and/or subscribers. It also allows for Remote Procedure Calls, as defined by the DDS-RPC standard, which implement a request/reply communication pattern.

eProsima Micro XRCE-DDS provides both a plug and play *eProsima Micro XRCE-DDS Agent* and an API layer which allows the user to implement the *eProsimaMicro XRCE-DDS Clients*.



CHAPTER

ONE

MAIN FEATURES

High performance. The *eProsima Micro XRCE-DDS Client* uses a static low-level serialization library (eProsima Micro CDR) that serializes in XCDR.

Low resources. The *Client* library is dynamic and static memory free, as the only memory footprint is due to the stack growth. It can manage a simple publisher/subscriber with less of 2 kB of RAM. Besides, the *Client* is built according to a *profiles* concept, allowing to add or remove functionalities to/from the library at the same time as changing the binary size.

Multi-platform. The OS dependencies are treated as pluggable modules, so that users can easily implement their platform-specific modules for the *eProsima Micro XRCE-DDS Client* library. By default, the project can run both over the standard Operating Systems *Linux* and *Windows*, and on the Real-Time Operating Systems *Nuttx*, *FreeRTOS* and *Zephyr*.

Compiler dependencies free. The *Client* library uses pure C99 standard. No *C* compiler extensions are used.

Free and Open Source. The *Client* library, the *Agent* executable, the generator tool and other internal dependencies as *eProsima Micro CDR* or *eProsima Fast DDS* are all free and open-source.

Easy to use. The project provides several examples. This documentation guides the user step-by-step through some of them, namely how to create a *publisher/subscriber*, a *requester/replier* and a *Peer-to-Peer publiser/subscriber Client* example. An interactive demo can also be found to interact with the ShapesDemo application, useful to understand the DDS-XRCE protocol and to make tests. The *Client API* is thoroughly explained.

Implementation of the DDS-XRCE standard. DDS-XRCE is a standard communication protocol by the OMG consortium focused on communicating eXtremely Resource Constrained Environments with the DDS world.

Best effort and reliable communication. *eProsima Micro XRCE-DDS* supports both *best-effort* and *reliable* communication modes. The first implements a fast and light communication, while the second ensures reliability independently of the transport layer used underneath.

Pluggable transport layer. *eProsima Micro XRCE-DDS* is not built over a specific transport protocol as *Serial* or *UDP*. It is agnostic about the transport used, and give the user the possibility of implementing the needed transport easily. By default, *UDP*, *TCP*, and *Serial* transports are provided. Also, an easy way to implement *Custom* transport is offered.

Commercial support Available at *support@eprosima.com*

CHAPTER

TWO

INSTALLATION

To install *eProsima Micro XRCE-DDS*, follow the instructions provided in the *Installation* page.

CHAPTER

THREE

USER MANUAL

To test *eProsima Micro XRCE-DDS*, follow the *Quick start* instructions. This page shows how to create simple *publisher/subscriber* and *requester/replier* applications.

Additionally, there is an interactive example called *Shapes Demo* allowing users to create entities and to send/receive topics by instructions given by the command line. This example is useful to understand how the DDS-XRCE protocol interfaces with the DDS World.

To learn how to handle all the ingredients needed to create a *Client*, carefully read the *Getting started* page. This page describes how to use the *eProsima Micro XRCE-DDS* API in order to set up and run a *Client* application.

A generic introduction to the library can be found in the *Overview* page. To know more about *Clients* and *Agents*, find detailed information in the *eProsima Micro XRCE-DDS Client* and *Client API* pages, and in the *eProsima Micro XRCE-DDS Agent* and *Agent API* pages respectively.

CHAPTER

FOUR

EPROSIMA MICRO XRCE-DDS GEN

To create a serialization/deserialization topic code for the *eProsima Micro XRCE-DDS Client*, there is a tool called `microxrceddsgen`. Information about this tool can be found in the *eProsima Micro XRCE-DDS Gen* page.

CHAPTER

FIVE

STRUCTURE OF THE DOCUMENTATION

5.1 External dependencies

In this section we list the dependencies of the libraries composing the *eProsima Micro XRCE-DDS* suite.

5.1.1 eProsima Micro XRCE-DDS Client

The *eProsima Micro XRCE-DDS Client* has no external dependencies.

5.1.2 eProsima Micro XRCE-DDS Agent

eProsima Fast DDS The *eProsima Micro XRCE-DDS Agent* requires *eProsima Fast DDS* to work. If *eProsima Fast DDS* is already installed in the system, the *Agent* will look for it and use it. Otherwise, it will be automatically downloaded together with the *Agent* application.

If willing to install *eProsima Fast DDS*, follow the instructions provided in the installation guide of the eProsima Fast DDS documentation.

5.1.3 eProsima Micro XRCE-DDS Gen

In order to compile the code generation tool *Micro XRCE-DDS Gen*, the following packages need to be installed in the system.

Java JDK The JDK is a development environment for building applications and components using the Java language. Download and install it following the steps provided in the Oracle website.

Gradle Gradle is an open-source build automation tool, **version 7.0 or higher is required**. Download and install it following the steps provided in the Gradle website.

5.1.4 Windows

Microsoft Visual C++ 2017 or greater *eProsima Micro XRCE-DDS* is supported on Windows over the Microsoft Visual C++ 2017 or greater frameworks.

5.2 Installation

In this section, instructions are provided to install the following packages:

- *Using pre-installed docker images*
- *Installing the Agent standalone*
- *Installing the Client standalone*
- *Installing the Micro XRCE-DDS Gen tool*
- *Installing Agent and Client*

The user can decide wether to install the *Agent* and *Client* as stand-alone packages, or together. If the first approach is chosen, refer to the *Release Notes* section in order to match versions that are compatible.

5.2.1 Using pre-installed docker images

Download here the Micro XRCE-DDS and Fast-DDS Suite docker image that contains a pre-installed client and agent as well as some compiled examples. More information about this Docker image can be found *here*.

5.2.2 Installing the Agent standalone

Clone the project from GitHub:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
$ cd Micro-XRCE-DDS-Agent
$ mkdir build && cd build
```

On Linux, inside of the `build` folder, execute the following commands:

```
$ cmake ..
$ make
$ sudo make install
```

On Windows first select the Visual Studio version:

```
$ cmake -G "Visual Studio 15 2017 Win64" ..
$ cmake --build .
$ cmake --build . --target install
```

Note: The *eProsima Micro XRCE-DDS Agent* can be configured at compile-time via several CMake definitions. Find them listed in the *Configuration* section of the *eProsima Micro XRCE-DDS Agent* page.

Now the the executable *eProsima Micro XRCE-DDS Agent* is installed in the system. Before running it, add `/usr/local/lib` to the dynamic loader-linker directories.

```
sudo ldconfig /usr/local/lib/
```

Important: The *eProsima Micro XRCE-DDS Agent* executable comes with a rich CLI. Find out all the options offered by this CLI when running the *Agent* in the *Agent CLI* section of the *eProsima Micro XRCE-DDS Agent* page.

Installation from Snap package

The *eProsima Micro XRCE-DDS Agent* can also be installed as a Snap package..

To this aim, simply execute `sudo snap install micro-xrce-dds-agent` in the console. This will download the Snap package corresponding to the `stable` version, that is, the *master* branch on GitHub.

For downloading the Snap image corresponding to the *develop* branch, add the `--edge` flag to the installation command.

Note: The Snap package is only available for Linux.

Using the provided Docker image

The *eProsima Micro XRCE-DDS Agent* can also be launched directly from its dedicated Docker image..

Pull the image by executing `docker pull eprosima/micro-xrce-dds-agent:<tag> <<args>>`, with *tag* being one of the following options:

- `stable`: Micro XRCE-DDS Agent master branch.
- `latest`: Micro XRCE-DDS Agent develop branch)
- `vM.m.p`: Micro XRCE-DDS Agent tagged versions, with the *Major*, *minor*, *patch* structure.

The accepted arguments for `<<args>>` are exactly the same which are listed in the *Agent CLI* section.

5.2.3 Installing the Client standalone

Clone the project from GitHub:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Client.git
$ cd Micro-XRCE-DDS-Client
$ mkdir build && cd build
```

On Linux, inside of `build` folder, execute the following commands:

```
$ cmake ..
$ make
$ sudo make install
```

Now the the executable *eProsima Micro XRCE-DDS Client* is installed in the system. Before running it, add `/usr/local/lib` to the dynamic loader-linker directories.

```
sudo ldconfig /usr/local/lib/
```

On Windows first select the Visual Studio version:

```
$ cmake -G "Visual Studio 15 2017 Win64" ..
$ cmake --build .
$ cmake --build . --target install
```

Note: In order to install the *eProsima Micro XRCE-DDS Client* examples, add `-DUCLIENT_BUILD_EXAMPLES=ON` to the `cmake ..` command-line options. This flag will enable the compilation of the examples. In addition to this flag, there are several other CMake definitions for configuring the

building of the client library at compile-time. Find them in the *Profiles* and `configurations` sections of the *eProsima Micro XRCE-DDS Client* page.

For building a Client app in the host machine, compile against the following libs:

```
gcc <main.c> -lmicrocdr -lmicroxrcedds_client
```

Using the provided Docker image

The *eProsima Micro XRCE-DDS Client* comes with a Docker image where the library is installed together with the provided examples, so they can easily be executed by the users.

Pull the image by executing `docker pull eprosima/micro-xrce-dds-client:<tag> <<args>>`, with *tag* being one of the following options:

- `stable`: Micro XRCE-DDS Client master branch.
- `latest`: Micro XRCE-DDS Client develop branch
- `vM.m.p`: Micro XRCE-DDS Client tagged versions, with the *Major*, *minor*, *patch* structure.

The accepted arguments for `<<args>>` are the examples' executable names, followed by the arguments required for each example to work. Find a list of all the available examples here. Note that they may differ between *master* and *develop* branches and the tagged versions.

5.2.4 Installing the Micro XRCE-DDS Gen tool

Install dependencies, clone the project from GitHub and build:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Gen.git
$ cd Micro-XRCE-DDS-Gen
$ git submodule init
$ git submodule update
$ gradle build -Dbranch=v1.2.5
```

The *Micro XRCE-DDS-Gen* tool will be available as:

```
$ ./scripts/microxrceddsgen -help
```

5.2.5 Installing Agent and Client

Clone the project from GitHub:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS.git
$ cd Micro-XRCE-DDS
$ mkdir build && cd build
```

On Linux, inside of the `build` folder, execute the following commands:

```
$ cmake ..
$ make
$ sudo make install
```

On Windows choose the Visual Studio version using the CMake option *-G*, for example:

```
$ cmake -G "Visual Studio 15 2017 Win64" ..
$ cmake --build . --target install
```

Now both the *eProsima Micro XRCE-DDS Agent* and the *eProsima Micro XRCE-DDS Client* are installed in the system.

Note: In order to install the *eProsima Micro XRCE-DDS Gen* tool as well, add `-DUXRCE_ENABLE_GEN=ON` to the `cmake ..` command-line options. This flag will enable the downloading and compilation of the code generating tool.

Note: In order to install the *eProsima Micro XRCE-DDS* examples, add `-DUXRCE_BUILD_EXAMPLES=ON` to the `cmake ..` command-line options. This flag will enable the compilation of the examples.

Using the provided Docker image

eProsima Micro XRCE-DDS is also available as a whole package in a Docker image.

Within this Docker image, the *Micro XRCE-DDS Agent* standalone application and library are installed, as well as the *Micro XRCE-DDS Client* library and built-in examples.

Pull the image by executing `docker pull eprosima/micro-xrce-dds:<tag> <<args>>`, with *tag* being one of the following options:

- `stable`: Micro XRCE-DDS master branch.
- `latest`: Micro XRCE-DDS develop branch
- `vM.m.p`: Micro XRCE-DDS tagged versions, with the *Major*, *minor*, *patch* structure.

The accepted arguments for `<<args>>` are:

- To launch the *Micro XRCE-DDS Agent*: `MicroXRCEAgent <<agent_args`, being `<<agent_args>>` the ones described in the *Agent CLI* section.
- The *Micro XRCE-DDS Client* examples' executable names, as explained *above.*

5.3 Overview

This section provides an overview of the *Micro XRCE-DDS* library. It is organized as follows:

- *DDS-XRCE protocol*
- *DDS standard and Fast DDS*
- *Operations and entities*

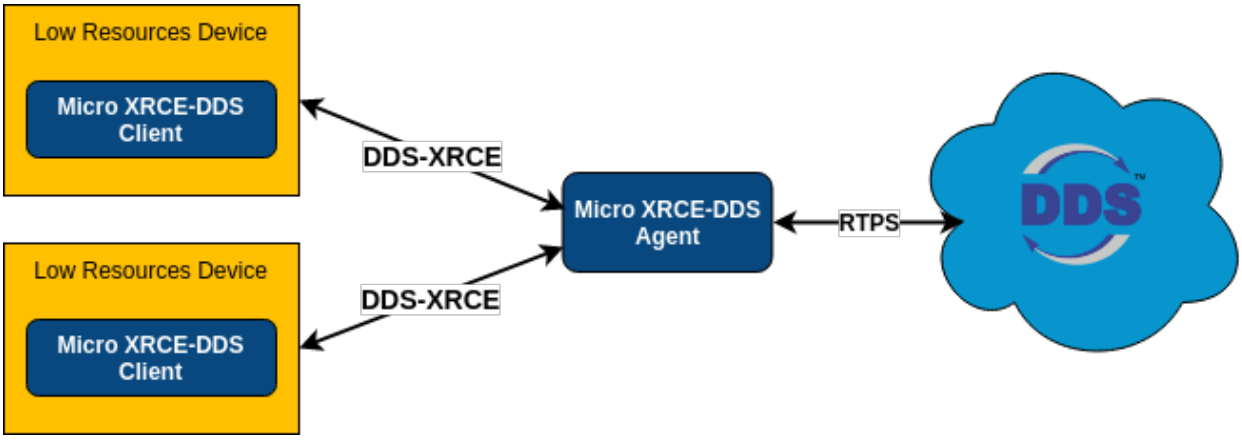
5.3.1 DDS-XRCE protocol

eProsima Micro XRCE-DDS implements the DDS-XRCE protocol specified in the *DDS for eXtremely Resource Constrained Environments* standard as defined and maintained by the *Object Management Group (OMG)* consortium.

This protocol allows resource-constrained devices to communicate with a larger *DDS (Data Distribution Service)* network. This communication is based on a client-server architecture, where the server (XRCE Agent) acts as an intermediary between the clients (XRCE Clients) and the *DDS Global Data Space*.

The *DDS-XRCE protocol* defines the wire protocol between XRCE Agents and XRCE Clients. The messages exchanged revolve around operations and their responses. The XRCE Clients request the XRCE Agents to run operations, and the XRCE Agents reply with the result of the requested operations. Making use of these operations, the XRCE Clients can create the DDS entities' hierarchy necessary to publish and/or receive data from DDS. The DDS entities are created and stored on the XRCE Agent's side so the XRCE Clients can reuse them at will with subsequent operations requests.

eProsima Micro XRCE-DDS implements the DDS-XRCE protocol using an XRCE Agent as a broker and providing a C API for developing XRCE Clients applications. The Micro XRCE-DDS Agent uses *eProsima Fast DDS* to reach the DDS Global Data Space.



5.3.2 DDS standard and Fast DDS

eProsima Fast DDS is a C++ implementation of the DDS standard and makes underneath use of the *RTPS (Real-Time Publish-Subscribe)* wire protocol, which provides publisher-subscriber communications over unreliable transports such as UDP. Both the DDS specification and the RTPS protocol are defined and maintained by the *OMG* consortium.

For more extensive information, please refer to the official documentation: eProsima Fast DDS.

5.3.3 Operations and entities

The communication between *XRCE Clients* and *XRCE Agents* is based upon *Operations* and responses. Clients request operations to the Agents, which generate responses with the result of these operations. The Client, once informed back on the result of th requested operations, will be able to perform subsequent actions and/or request further operations.

The communication starts with a handshake for the Agent to acknowledge that a Client is present in the network. This happens via a *Create session* operation forwarded from the Client to the Agent, with which the latter registers the Client. Without registering a session, all the subsequent operations sent to the Agent will be refused. Once registered, the Client can request operations to the Agent, through which it can create and query entities. The communication with DDS is handled by the Agent using these *Entities*.

The *Create entity* operation is the request used to create entities in the *Agent*. Each of these entities corresponds to an *eProsima Fast DDS* object.

For sending and receiving data from/to DDS, the Client has access to the *DataWriter* and *DataReader* entities. These entities handle the writing/reading operations. For sending and receiving any topic, the *Write Data* and *Read Data* operations must be used.

To remove any entity from the Agent, use the *Delete entity* operation. Also, to log out a Client session from the Agent, use the *Delete session* operation.

Operations

Operations are the possible actions the *eProsima Micro XRCE-DDS Client* can request to the *eProsima Micro XRCE-DDS Agent*. Operations revolve around *Entities*. The *eProsima Micro XRCE-DDS Agent* will respond to all the requests with the status of the operation.

Operation	Description
Create session	With this operation *Clients* asks the *Agents* to register a session. It is the first operation that must be performed. If this operation fails or is missing, any of the following operations will not work. If it is successful, it creates the session establishing the *Client-Agent* connection.
Delete session	This operation deletes the *Client-Agent* connection and removes all entities associated with it. After this operation, any other operation except *Create session* will fail.
Create entity	A session can create all the entities it needs. There is a *Create entity* operation for each entity the session can handle. Each *Create entity* operation is related to an ID for its management.
Delete entity	Analogously, a session can delete the entities that were previously created on the *Agent*. To drop an entity, the entity ID must be used.
Request Data	This operation configures the data reception, which the *Agent* will deliver from the DDS data space to the *Client*. Data are received asynchronously, according to the data delivery control set in this operation. Reading data is done using a *DataReader* entity.

Entities

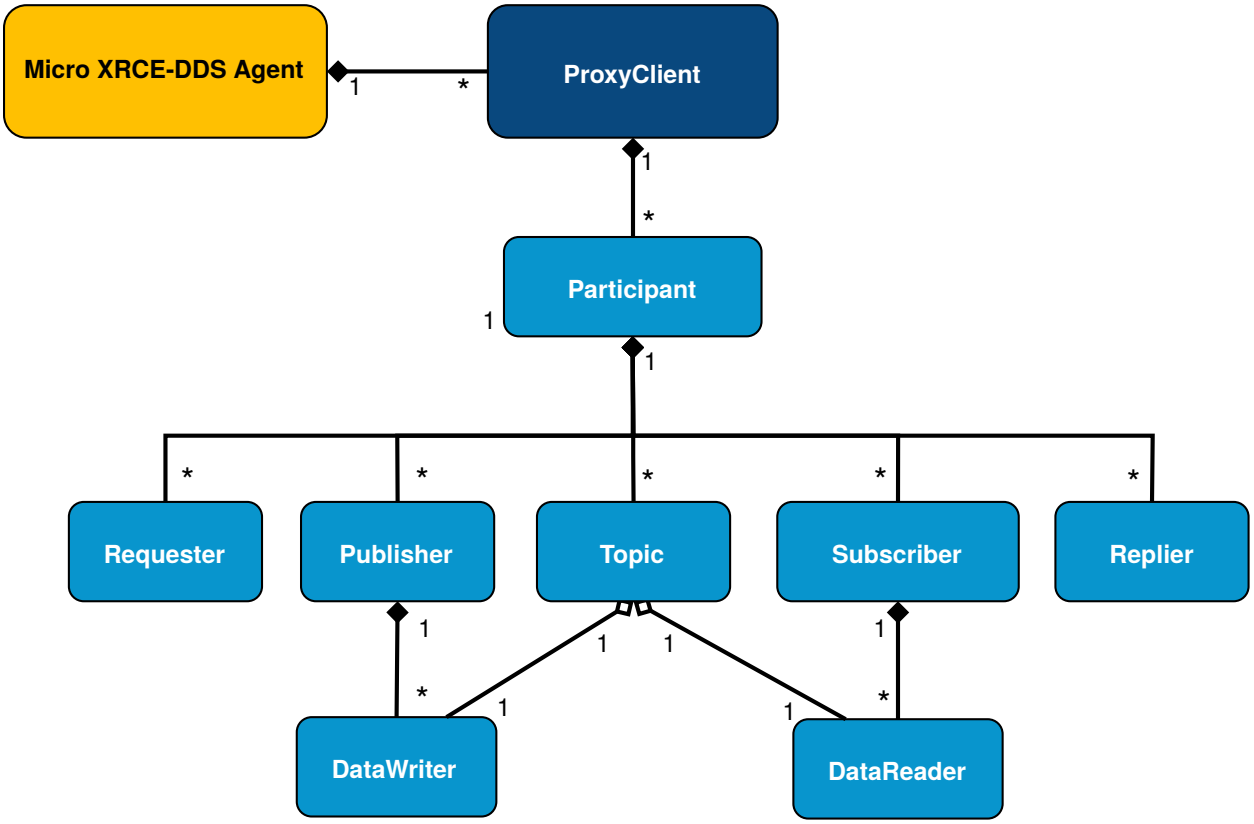
The protocol underlying *eProsima Micro XRCE-DDS* (DDS-XRCE), defines entities that have a direct correspondence with their analogous actors on *eProsima Fast DDS*. The entities manage the communication between *eProsima Micro XRCE-DDS Clients* and the DDS Global Data Space. Entities are stored in the *eProsima Micro XRCE-DDS Agent* and the *eProsima Micro XRCE-DDS Client* can create, use and destroy these entities.

The entities are uniquely identified by an ID called *Object ID*. The *Object ID* is the way a *Client* refers to them inside an *Agent*. In most of the *Client* request operations it is necessary to specify an ID referring to one of the *Client* entities stored in the *Agent*.

Find below a table describing the entities the *Client* can interact with.

Entity	Description
Participant	Participants can hold any number of Publishers and/or Subscribers.
Publisher	Publishers can hold any number of data writers.
Subscriber	Subscribers can hold any number of data readers.
Topic	Topic data is the base of the communication. A Topic is composed of a name and a data type.
DataWriter	This is the endpoint able to write Topic data.
DataReader	This is the endpoint able to read Topic data.
Requester	This is the endpoint able to write Request data and to read Reply data.
Replier	This is the endpoint able to read Request data and to write Reply data.

This figure shows the entities hierarchy



The creation the entities listed above needs to be done using the DDS XML configuration of the object to create. The XML configuration follows the same rules as in *eProsima Fast DDS*.

The data sent by the Client to the DDS Global Data Space closely resembles that of *eProsima Fast DDS*. The Interface Definition Language (IDL) is used to define the type and must be known by the Client. Having the type defined as *IDL*, we provide the *eProsima Micro XRCE-DDS Gen* tool. This tool can generate a compatible type that the XRCE Client can use to send and receive. The type has to match the one used on the DDS Side.

5.4 Quick start

eProsima Micro XRCE-DDS provides a C API which allows the creation of *eProsima Micro XRCE-DDS Clients* that can either publish/subscribe to topics from the DDS Global Data Space, or act as client-service applications following a request/reply pattern.

In this section, we will guide the user through the deployment of two out-of-the-box examples. In the first one, an *eProsima Micro XRCE-DDS Agent* bridges two *eProsima Micro XRCE-DDS Clients* publishing and subscribing to the DDS world. The second example shows instead the deployment of a client-service application using the Requester and Replier entities, also put into communication via an *eProsima Micro XRCE-DDS Agent*.

The section is organized as follows:

- *Running an Agent*
- *Running a Publisher-Subscriber example*
- *Running a Requester/Replier example*

5.4.1 Running an Agent

First of all, install the *Agent* as explained in the *Installing the Agent standalone* section. On Linux, this would be:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
$ cd Micro-XRCE-DDS-Agent
$ mkdir build && cd build
$ cmake ..
$ make
$ sudo make install
```

After having installed the *Agent* system-wide, it's possible to launch it.

For this example, the *Client-Agent* communication will be done through UDP, using the port `2019` and with the XML creation mode, which is the default mode for creating entities:

```
$ cd /usr/local/bin && MicroXRCEAgent udp4 -p 2019
```

5.4.2 Running a Publisher-Subscriber example

In this section, we guide the user through the configuration and deployment of a simple publish/subscribe example where the communication is mediated by the *Agent* created above.

Before considering the publisher and subscriber examples, it is useful to briefly summarize how the *Publisher* and *Subscriber* entities work, as well as to list the functions related to both entities.

Publisher The *Publisher* will be associated with a *Topic* and will handle a DDS publisher that publishes topics.

To create a *Publisher* entity, the `uxr_buffer_create_publisher_xml` or `uxr_buffer_create_publisher_ref` shall be used. Once created, topics can be published through `uxr_prepare_output_stream`.

Subscriber The *Subscriber* will be associated with a *Topic* and will handle a DDS subscriber that receives topics.

To create a *Subscriber* entity, the `uxr_buffer_create_subscriber_xml` or `uxr_buffer_create_subscriber_ref` shall be used. Topics can be received by sending a data request to the *Agent* with `uxr_buffer_request_data`, and through the *on_topic* callback which shall be set by the `uxr_set_topic_callback`. This callback has a parameter *request_id* which identifies the data request.

All the files and the code used in this example can be found in the Micro-XRCE-DDS-Client/examples/PublishHelloWorld and Micro-XRCE-DDS-Client/examples/SubscribeHelloWorld folders.

Publisher application

Let's now install the *Client* locally, and with the `-DUCLIENT_BUILD_EXAMPLES=ON` flag enabled, so as to activate the compilation of the examples. On Linux, this implies running the following:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Client.git
$ cd Micro-XRCE-DDS-Client
$ mkdir build && cd build
$ cmake .. -DUCLIENT_BUILD_EXAMPLES=ON
$ make
```

At this point, it's possible to launch the `PublishHelloWorldClient` executable located in the folder `Micro-XRCE-DDS-Client/build/examples/PublishHelloWorld`, which'll make the *Client* publish in the DDS World the `HelloWorld` topic (take a look at the IDL defining this topic in the file `Micro-XRCE-DDS-Client/examples/PublishHelloWorld/HelloWorld.idl`).

```
$ examples/PublishHelloWorld/PublishHelloWorldClient 127.0.0.1 2019
```

The source code of the `PublishHelloWorldClient` can be found in `Micro-XRCE-DDS-Client/examples/PublishHelloWorld/main.c`.

Subscriber application

After having executed the publisher app, we can launch the `SubscribeHelloWorldClient` executable, which is located in the folder `Micro-XRCE-DDS-Client/build/examples/SubscribeHelloWorld`, which'll make this *Client* subscribe to the same `HelloWorld` topic from the DDS World.

```
$ examples/SubscribeHelloWorld/SubscribeHelloWorldClient 127.0.0.1 2019
```

The source code of the `SubscribeHelloWorldClient` can be found in `Micro-XRCE-DDS-Client/examples/SubscribeHelloWorld/main.c`.

At this point, the subscriber will receive the topics that are being sent by the publisher.

In order to see the messages from the DDS Global Data Space point of view, use the *eProsima Fast DDS* HelloWorld example running a subscriber. Find more information on how to do so at Fast DDS HelloWorld.

5.4.3 Running a Requester/Replier example

This section shows an example of a client-service application using the *Requester* and *Replier* entities. This application has two ends, the client (*RequestAdder*) and the service (*ReplyAdder*). On the one hand, the client is in charge of sending requests which contain two integers, as well as receiving the responses from the service. On the other hand, the service is in charge of receiving the requests from the client, summing the two integers, and finally of sending the response to the client.

Before considering the client and service examples, it is useful to briefly summarize how the *Requester* and *Replier* entities work, as well as to list the functions related to both entities.

Requester The *Requester* entity is composed of a *Publisher* and a *Subscriber* associated with a *RequestTopic* and a *ReplyTopic* respectively. The *Publisher* is in charge of sending the request, while the *Susbscriber* receives the replies.

To create a *Requester* entity, the `uxr_buffer_create_requester_xml` or `uxr_buffer_create_requester_ref` shall be used. Once created, requests can be sent through `uxr_buffer_request`. Replies can be received by sending a data request to the *Agent* with `uxr_buffer_request_data`, and through the *on_reply* callback which shall be set by the `uxr_set_reply_callback`. This callback has a parameter `reply_id` which corresponds to the identifier returned by the `uxr_buffer_request` call.

Replier The *Reply* entity is a mirror of the *Requester*, that is, it contains a *Publisher* and a *Subscriber* as well, but the topic association is reversed, as the *Publisher* is associated with the *ReplyTopic* and the *Subscriber* to the *RequestTopic*. In this case, the *Subscriber* is in charge of receiving the request from the *Requester*, while the *Publisher* sends the replies.

To create a *Replier* entity, the `uxr_buffer_create_replier_xml` or `uxr_buffer_create_replier_ref` shall be used. Once created, replies can be sent through `uxr_buffer_reply`. Requests can be received by sending a data request to the *Agent* with `uxr_buffer_request_data`, and through the *on_request* callback which shall be set by the `uxr_set_request_callback`. This callback has a parameter *sample_id* which identifies the request and should be used in the `uxr_buffer_reply`.

All the files and the code used in this example can be found in the Micro-XRCE-DDS-Client/examples/RequestAdder and Micro-XRCE-DDS-Client/examples/ReplyAdder folders.

Requester application

Let's now install the *Client* locally, and with the `-DUCLIENT_BUILD_EXAMPLES=ON` flag enabled, so as to activate the compilation of the examples. On Linux, this implies running the following:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Client.git
$ cd Micro-XRCE-DDS-Client
$ mkdir build && cd build
$ cmake .. -DUCLIENT_BUILD_EXAMPLES=ON
$ make
```

At this point, it's possible to launch the `RequestAdder` executable located in the folder `Micro-XRCE-DDS-Client/build/examples/RequestAdder`, which'll make the *Client* send two integers as a request, and receive the sum of both integers as a response.

```
$ examples/RequestAdder/RequestAdder 127.0.0.1 2019
```

The source code of the `RequestAdder` can be found in `Micro-XRCE-DDS-Client/examples/RequestAdder/main.c`.

Replier application

After having executed the Requester app, we can launch the `ReplyAdder` executable, which is located in the folder `Micro-XRCE-DDS-Client/build/examples/ReplyAdder`, which'll make this *Client* receive requests composed by two integers, sum both numbers, and finally send the response.

```
$ examples/ReplyAdder/ReplyAdder 127.0.0.1 2019
```

The source code of the `ReplyAdder` can be found in `Micro-XRCE-DDS-Client/examples/ReplyAdder/main.c`.

At this point, the Requester and the Replier will start communicating.

5.4.4 Learn More

Find a detailed explanation of the code used to write and run these applications in the *Getting started* section.

Find other relevant material:

- *eProsima Fast DDS*: eProsima Fast DDS
- To learn how to install *eProsima Micro XRCE-DDS* read: *Installation*
- To learn more about *eProsima Micro XRCE-DDS* read: *Overview*
- To learn more about *eProsima Micro XRCE-DDS Gen* read: *eProsima Micro XRCE-DDS Gen*

5.5 Getting started

This page shows how to get started with the *eProsima Micro XRCE-DDS Client*. We will create a *Client* that can publish and subscribe to a topic, or engage in a request-reply kind of communication. Also, we illustrate how to create C code consumable by the client from a IDL file with *eProsima Micro XRCE-DDS Gen*. Finally, we provide a deployment example.

The section is organized as follows:

- *Prerequisites*
- *Generate code from an IDL*
- *Initialize a Session*
- *Setup a Participant*
- *Create topics*
- *Publishers & Subscribers*
- *DataWriters & DataReaders*
- *Requester & Replier*
- *Agent response*
- *Write Data*
- *Read Data*
- *Close the Client*
- *Deployment example*

Hint: The code shown here can be found in the examples below:

- `examples/PublishHelloWorld`
- `examples/SubscribeHelloWorld`
- `examples/ReplyAdder`
- `examples/RequestAdder`
- `examples/Deployment`

Note: This example makes use of the creation mode by XML, which is one of the two possible representation formats for creating DDS entities: by XML or by reference (see the *Creation Mode: Client* and *Creation Mode: Agent* sections).

5.5.1 Prerequisites

First, make sure to have correctly installed the following:

- *Installing the Agent standalone*.
- *Installing the Client standalone*.
- *Installing the Micro XRCE-DDS Gen tool*.

5.5.2 Generate code from an IDL

We will use HelloWorld as our Topic whose IDL is the following:

```
struct HelloWorld
{
    unsigned long index;
    string message;
};
```

In the *Client* we need to create an equivalent C type with its serialization/deserialization code. This is done automatically by *eProsima Micro XRCE-DDS Gen*:

```
$ microxrceddsgen HelloWorld.idl
```

5.5.3 Initialize a Session

In the source example file, we include the generated type code, to have access to its serialization/deserialization functions along to the writing function. Also, we will specify the max buffer for the streams and its historical associated for the reliable streams.

```
#include "HelloWorldWriter.h"

#define STREAM_HISTORY  8
#define BUFFER_SIZE     UXR_CONFIG_UDP_TRANSPORT_MTU * STREAM_HISTORY
```

Before create a Session we need to indicate the transport to use (the *Agent* must be configured for listening from UDP at port 2018).

```
uxrUDPTransport transport;
if (!uxr_init_udp_transport(&transport, UXR_IPv4, "127.0.0.1", "2018"))
{
    printf("Error at create transport.\n");
    return 1;
}
```

Next, we will create a session that allows us interacting with the *Agent*:

```
uxrSession session;
uxr_init_session(&session, &transport.comm, 0xABCDABCD);
uxr_set_topic_callback(&session, on_topic, NULL);
if(!uxr_create_session(&session))
{
    printf("Error at create session.\n");
    return 1;
}
```

The first function `uxr_init_session` initializes the `session` structure with the transport and the *Client Key* (the session identifier for an *Agent*). The `uxr_set_topic_callback` function is for registering the function `on_topic` which will be called when the *Client* receives a topic. Once the session has been initialized, we can send the first message for logging the *Client* in the *Agent* side: `uxr_create_session`. This function will try to connect with the *Agent* by `CONFIG_MAX_SESSION_CONNECTION_ATTEMPTS` attempts (configurable as a CMake argument).

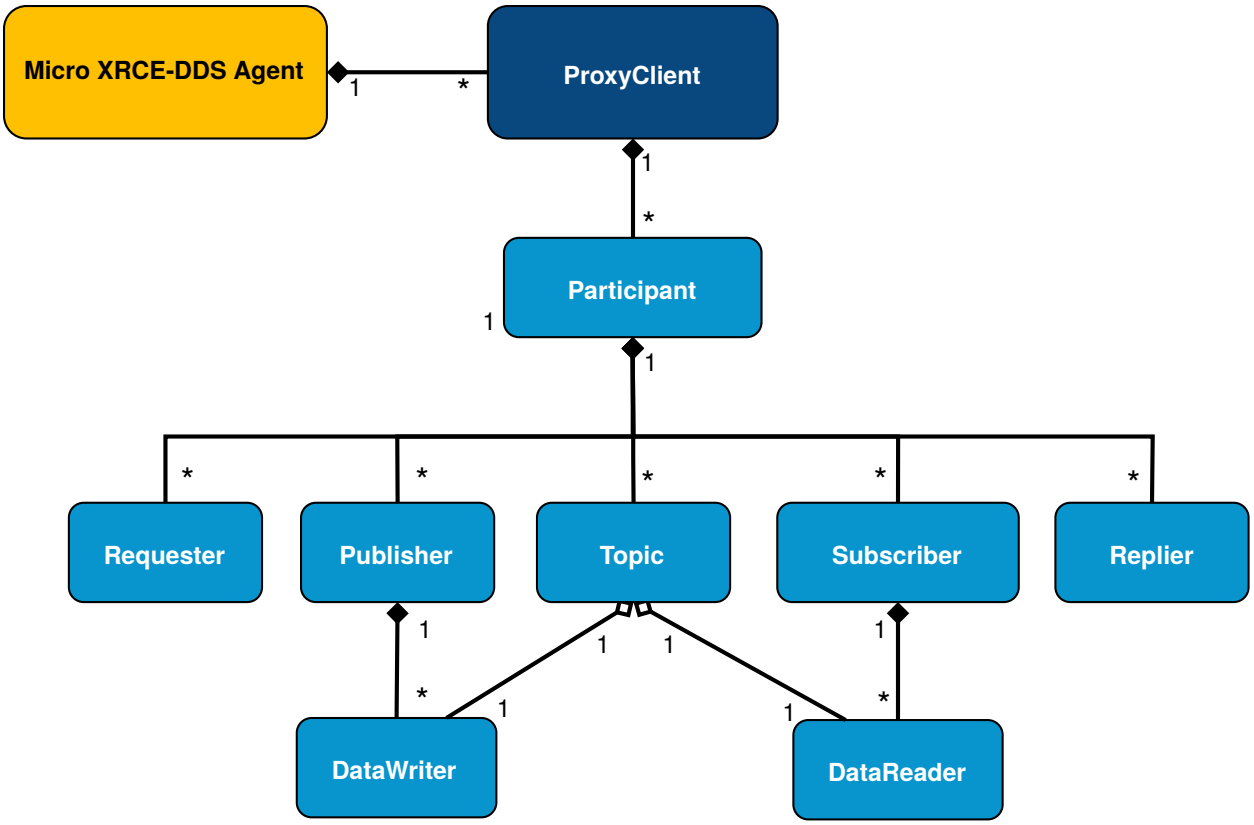
Optionally, we also could add a status callback with the function `uxr_set_status_callback`, but for this example, we do not need it.

Once we have logged in the session successfully, we can create the streams that we will use. In this case, we will use two, both reliables, for input and output.

```
uint8_t output_reliable_stream_buffer[BUFFER_SIZE];
uxrStreamId reliable_out = uxr_create_output_reliable_stream(&session, output_
↪reliable_stream_buffer, BUFFER_SIZE, STREAM_HISTORY);

uint8_t input_reliable_stream_buffer[BUFFER_SIZE];
uxrStreamId reliable_in = uxr_create_input_reliable_stream(&session, input_reliable_
↪stream_buffer, BUFFER_SIZE, STREAM_HISTORY);
```

To publish and/or subscribes to a topic, we need to create a hierarchy of XRCE entities in the *Agent* side. These entities will be created from the *Client*.



5.5.4 Setup a Participant

For establishing DDS communication, we need to create a *Participant* entity for the *Client* in the *Agent*. We can do this calling *Create participant* operation:

```
uxrObjectId participant_id = uxr_object_id(0x01, UXR_PARTICIPANT_ID);
const char* participant_xml = "<dds>"
                                  "<participant>"
                                      "<rtps>"
                                          "<name>default_xrce_participant</name>"
                                      "</rtps>"
                                  "</participant>"
                              "</dds>";
uint16_t participant_req = uxr_buffer_create_participant_ref(&session, reliable_out,␣
↪participant_id, participant_xml, UXR_REPLACE);
```

In any *XRCE Operation* that creates an entity, an *Object ID* is necessary. It is used to represent and manage the entity in the *Client* side. In this case, we will create the entity by its XML description, but also could be done by a reference of the entity in the *Agent*. Each operation returns a *Request ID*. This identifier of the operation can be used later for associating the status with the operation. In this case, the operation has been written into the stream `reliable_out`. Later, in the `run_session` function, the data written in the stream will be sent to the *Agent*.

5.5.5 Create topics

Once the *Participant* has been created, we can use *Create topic* operation to register a *Topic* entity within the *Participant*.

```
uxrObjectId topic_id = uxr_object_id(0x01, UXR_TOPIC_ID);
const char* topic_xml = "<dds>"
                            "<topic>"
                                "<name>HelloWorldTopic</name>"
                                "<dataType>HelloWorld</dataType>"
                            "</topic>"
                        "</dds>";
uint16_t topic_req = uxr_buffer_create_topic_xml(&session, reliable_out, topic_id,␣
↪participant_id, topic_xml, UXR_REPLACE);
```

As any other XRCE Operation used to create an entity, an Object ID must be specified to represent the entity. The `participant_id` is the participant where the Topic will be registered. To determine which topic will be used, an XML is sent to the *Agent* for creating and defining the Topic in the DDS Global Data Space. That definition consists of a name and a type.

5.5.6 Publishers & Subscribers

Similar to Topic registration, we can create *Publishers* and *Subscribers* entities. We create a publisher or subscriber on a participant entity, so it is necessary to provide the ID of the *Participant* which will hold those *Publishers* or *Subscribers*.

```
uxrObjectId publisher_id = uxr_object_id(0x01, UXR_PUBLISHER_ID);
const char* publisher_xml = "";
uint16_t publisher_req = uxr_buffer_create_publisher_xml(&session, reliable_out,␣
↪publisher_id, participant_id, publisher_xml, UXR_REPLACE);

uxrObjectId subscriber_id = uxr_object_id(0x01, UXR_SUBSCRIBER_ID);
const char* subscriber_xml = "";
uint16_t subscriber_req = uxr_buffer_create_subscriber_xml(&session, reliable_out,␣
↪subscriber_id, participant_id, subscriber_xml, UXR_REPLACE);
```

The *Publisher* and *Subscriber* XML information is given when the *DataWriter* and *DataReader* are created.

5.5.7 DataWriters & DataReaders

Analogously to publishers and subscribers entities, we create the *DataWriters* and *DataReaders* entities. These entities are in charge of sending and receiving the data. *DataWriters* are referred to as publishers, and *DataReaders* are referred to as subscribers. The configuration of these *DataReaders* and *DataWriters* are contained in the XML.

```
uxrObjectId datawriter_id = uxr_object_id(0x01, UXR_DATAWRITER_ID);
const char* datawriter_xml = "<dds>"
                                 "<data_writer>"
                                     "<topic>"
                                         "<kind>NO_KEY</kind>"
                                         "<name>HelloWorldTopic</name>"
                                         "<dataType>HelloWorld</dataType>"
                                     "</topic>"
                                 "</data_writer>"
                             "</dds>";
```

(continues on next page)

(continued from previous page)

```
uint16_t datawriter_req = uxr_buffer_create_datawriter_xml(&session, reliable_out,␣
↪datawriter_id, publisher_id, datawriter_xml, UXR_REPLACE);

uxrObjectId datareader_id = uxr_object_id(0x01, UXR_DATAREADER_ID);
const char* datareader_xml = "<dds>"
                                 "<data_reader>"
                                     "<topic>"
                                         "<kind>NO_KEY</kind>"
                                         "<name>HelloWorldTopic</name>"
                                         "<dataType>HelloWorld</dataType>"
                                     "</topic>"
                                 "</data_reader>"
                             "</dds>";
uint16_t datareader_req = uxr_buffer_create_datareader_xml(&session, reliable_out,␣
↪datareader_id, subscriber_id, datareader_xml, UXR_REPLACE);
```

5.5.8 Requester & Replier

There is another pair of coupled entities, the Requester and the Replier. These entities provide request-reply functionality using the underlining publish-subscribe pattern. It is achieved through a mirror configuration between a Requester and a Replier, that is, both entities contain a *Publisher* and a *Subscriber*, the *Publisher* of the *Requester* and the *Subscriber* of the *Replier* are associated with the same *Topic* and vice versa. In that way, each time a *Requester* publishes a request it will be received by the *Replier*, then the latter will generate a reply and publish it, and finally, this reply will be received by the *Requester*.

The following code shows how to create a *Requester* and a *Replier* using the XML representation.

```
uxrObjectId requester_id = uxr_object_id(0x01, UXR_REQUESTER_ID);
const char* requester_xml = "<dds>"
                                "<requester profile_name=\"my_requester\""
                                            "service_name=\"service_name\""
                                            "request_type=\"request_type\""
                                            "reply_type=\"reply_type\">"
                                "</requester>"
                            "</dds>";
uint16_t requester_req = uxr_buffer_create_requester_xml(&session, reliable_out,␣
↪requester_id, participant_id, requester_xml, UXR_REPLACE);

replier_id = uxr_object_id(0x01, UXR_REPLIER_ID);
const char* replier_xml = "<dds>"
                              "<replier profile_name=\"my_replier\""
                                       "service_name=\"service_name\""
                                       "request_type=\"request_type\""
                                       "reply_type=\"reply_type\">"
                              "</replier>"
                          "</dds>";
uint16_t replier_req = uxr_buffer_create_replier_xml(&session, reliable_out, replier_
↪id, participant_id, replier_xml, UXR_REPLACE);
```

5.5.9 Agent response

In operations such as create a session, create entity or request data from the *Agent*, a status is sent from the *Agent* to the *Client* indicating what happened.

For *Create session* or *Delete session* operations, the status value is stored into the `session.info.last_request_status`. For the rest of the operations, the statuses are sent to the input reliable stream `0x80`, that is, the first input reliable stream created, with index 0.

The different status values that the *Agent* can send to the *Client* are the following (defined in `uxr/client/core/session/session_info.h`):

```
UXR_STATUS_OK
UXR_STATUS_OK_MATCHED
UXR_STATUS_ERR_DDS_ERROR
UXR_STATUS_ERR_MISMATCH
UXR_STATUS_ERR_ALREADY_EXISTS
UXR_STATUS_ERR_DENIED
UXR_STATUS_ERR_UNKNOWN_REFERENCE
UXR_STATUS_ERR_INVALID_DATA
UXR_STATUS_ERR_INCOMPATIBLE
UXR_STATUS_ERR_RESOURCES
UXR_STATUS_NONE (never send, only used when the status is known)
```

The status can be handled by the `on_status_callback` callback (configured in `uxr_set_status_callback` function) or by the `run_session_until_all_status` as we will see.

```
uint8_t status[6]; // we have 6 request to check.
uint16_t requests[6] = {participant_req, topic_req, publisher_req, subscriber_req,␣
↪datawriter_req, datareader_req};
if(!uxr_run_session_until_all_status(&session, 1000, requests, status, 6))
{
    printf("Error at create entities\n");
    return 1;
}
```

The `run_session` functions are the main functions of the *eProsima Micro XRCE-DDS Client* library. They perform several tasks: send the stream data to the *Agent*, listen to data from the *Agent*, call callbacks, and manage the reliable connection. There are five variations of `run_session` function: - `uxr_run_session_time` - `uxr_run_session_until_timeout` - `uxr_run_session_until_confirmed_delivery` - `uxr_run_session_until_all_status` - `uxr_run_session_until_one_status`

Here we use the `uxr_run_session_until_all_status` variation that will perform these actions until all statuses have been confirmed or the timeout has been reached. This function will return `true` in case all statuses were *OK*. After calling this function, the status can be read from the `status` array previously declared.

5.5.10 Write Data

Once we have created a valid data writer entity, we can write data into the DDS Global Data Space using the writing operation. For creating a message with data, first, we must decide which stream we want to use, and write that topic in this stream.

```
HelloWorld topic = {count++, "Hello DDS world!"};

ucdrBuffer ub;
uint32_t topic_size = HelloWorld_size_of_topic(&topic, 0);
```

(continues on next page)

(continued from previous page)

```
(void) uxr_prepare_output_stream(&session, reliable_out, datawriter_id, &ub, topic_
↪size);
(void) HelloWorld_serialize_topic(&ub, &topic);

uxr_run_session_until_confirmed_delivery(&session, 1000);
```

`HelloWorld_size_of_topic` and `HelloWorld_serialize_topic` functions are automatically generated by *eProsima Micro XRCE-DDS Gen* from the IDL. The function `uxr_prepare_output_stream` requests a writing for a topic of `topic_size` size into the reliable stream represented by `reliable_out`, with a `datawriter_id` (correspond to the data writer entity used for sending the data in the *DDS World*). If the stream is available and the topic fits in it, the function will initialize the `ucdrBuffer` structure `ub`. Once the `ucdrBuffer` is prepared, the topic can be serialized into it. We are careless about `uxr_prepare_output_stream` return value because the serialization only will occur if the `ucdrBuffer` is valid.

After calling the writing function, the topic has been serialized into the buffer, but it has not been sent yet. To send the topic, it is necessary to call a `run_session` function. In this case, the function `uxr_run_session_until_confirmed_delivery` is called, which will wait until the message was confirmed or until the timeout has been reached.

5.5.11 Read Data

Once we have created a valid *DataReader* entity, we can read data from the DDS Global Data Space using the read operation. This operation configures how the *Agent* will send the data to the *Client*. The current implementation sends unlimited topics to the *Client*.

```
uxrDeliveryControl delivery_control = {0};
delivery_control.max_samples = UXR_MAX_SAMPLES_UNLIMITED;

uint16_t read_data_req = uxr_buffer_request_data(&session, reliable_out, datareader_
↪id, reliable_in, &delivery_control);
```

To configure how the *Agent* will send the topic, we must set the input stream. In this case, we use the input reliable stream previously defined. `datareader_id` corresponds with the *DataDeader* entity used for receiving the data. The `delivery_control` parameter is optional, and allows specifying how the data will be delivered to the *Client*. For the example purpose, we set it as *unlimited*, so any number HelloWorld topic will be delivered to the *Client*.

The `run_session` function will call the topic callback each time a topic will be received from the *Agent*.

```
void on_topic(uxrSession* session, uxrObjectId object_id, uint16_t request_id,␣
↪uxrStreamId stream_id, struct ucdrBuffer* ub, uint16_t length, void* args)
{
    (void) session; (void) object_id; (void) request_id; (void) stream_id; (void)␣
↪length; (void) args;

    HelloWorld topic;
    HelloWorld_deserialize_topic(ub, &topic);
}
```

To know which kind of Topic has been received, we can use the `object_id` parameter or the `request_id`. The `id` of the `object_id` corresponds to the *DataReader* that has read the Topic, so it can be useful to discretize among different topics. The `args` argument corresponds to user-free-data, that has been given at *uxr_set_status_callback* function.

5.5.12 Close the Client

To close a *Client*, we must perform two steps. First, we need to tell the *Agent* that the session is no longer available. This is done sending the next message:

```
uxr_delete_session(&session);
```

After this, we can close the transport used by the session.

```
uxr_close_udp_transport(&transport);
```

5.5.13 Deployment example

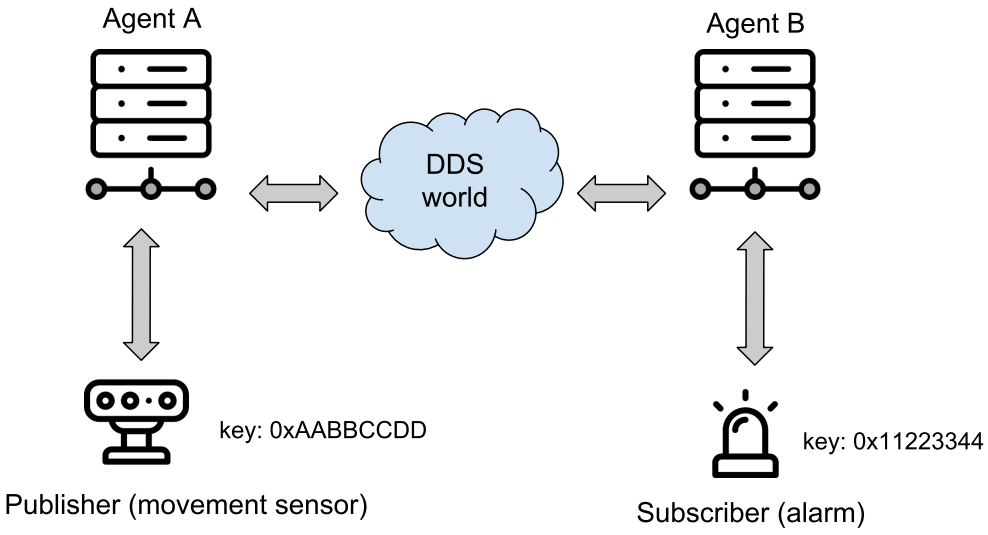
This section is devoted to illustrate how to deploy a system using *eProsima Micro XRCE-DDS* in a real environment. An example of this can be found in the `examples/Deployment` folder.

The tutorials above are based on *all in one* examples, that is, examples that create entities, publish or subscribe and then delete the resources. One possible real purpose of this consists in differentiating the logic of *creating entities* and the actions of *publishing and subscribing*. It can be done by creating two differents *Clients*, one in charge of configuring the entities in the *Agent*, which runs once, only for creating the entities at **compile-time**, and other/s that log(s) in the same session as the first *Client* (sharing the entities) and only publish(es) or subscrib(es) to data.

This allows creating *Clients* in a real scenario with the only purpose of sending and receiving data. The concept of *profiles* allows building the *Client* library only with the chosen behavior (only to publish or to subscribe, for example). See *eProsima Micro XRCE-DDS Client* for more information.

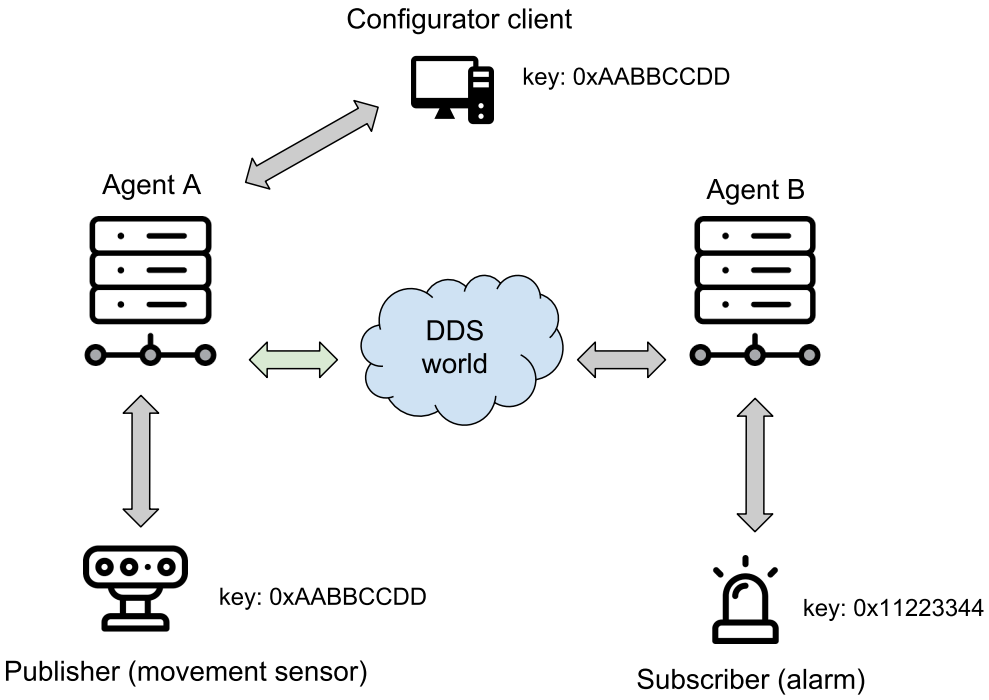
The diagram below shows an example of how to configure the environment using a *configurator client*.

Initial state



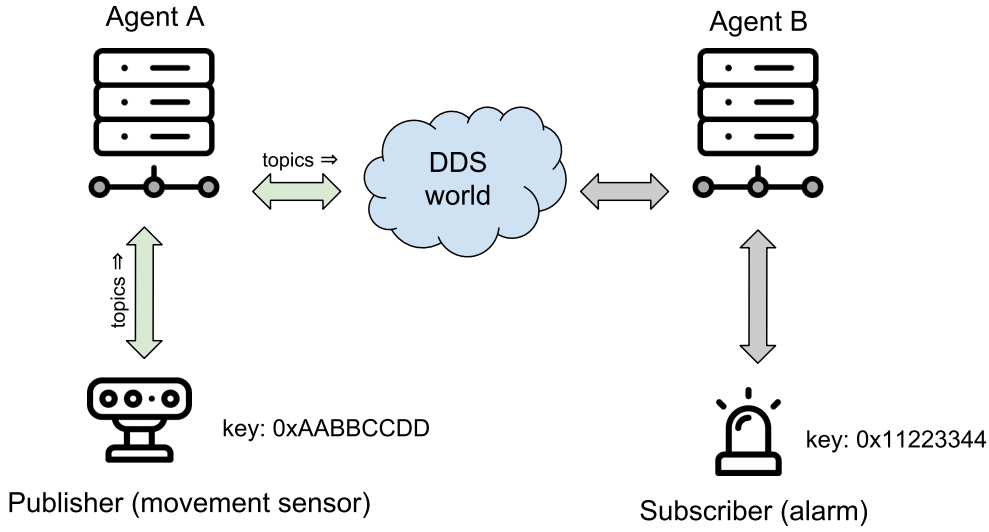
The environment contains two *Agents* (it's perfectly possible to use only one *Agent* too), and two *Clients*, one for publishing and another for subscribing.

Publisher configuration



In this state a *configurator client* is connected to the *Agent A* with the *client key* that will be used by the future *publisher client* (0xAABBCCDD). Once a session is logged in, the *configurator client* creates all the necessary entities for the *publisher client*. This implies the creation of *participant*, *topic*, *publisher*, and *datawriter* entities. These entities have a representation as DDS entities, and can be reached from the DDS world, that is, any *subscriber DDS entity* could already be listening to topics if it matches with such *publisher DDS entity* through the *DDS* world.

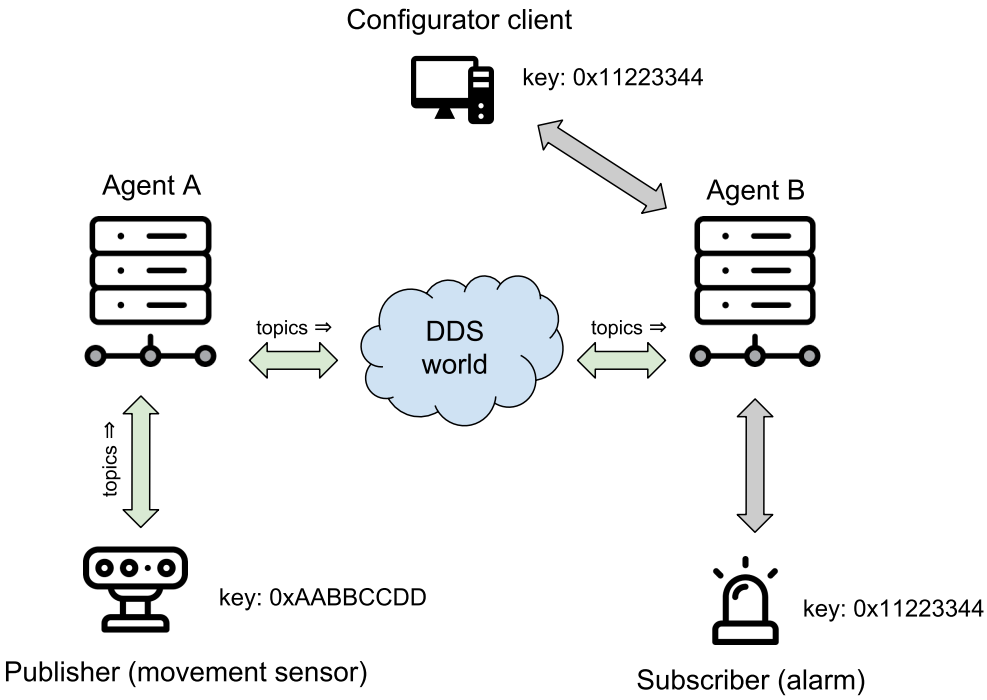
Publisher



Then, the *publisher client* is connected to the *Agent A*. This *Client* logs in a session with its *Client* key (0xAABBCCDD).

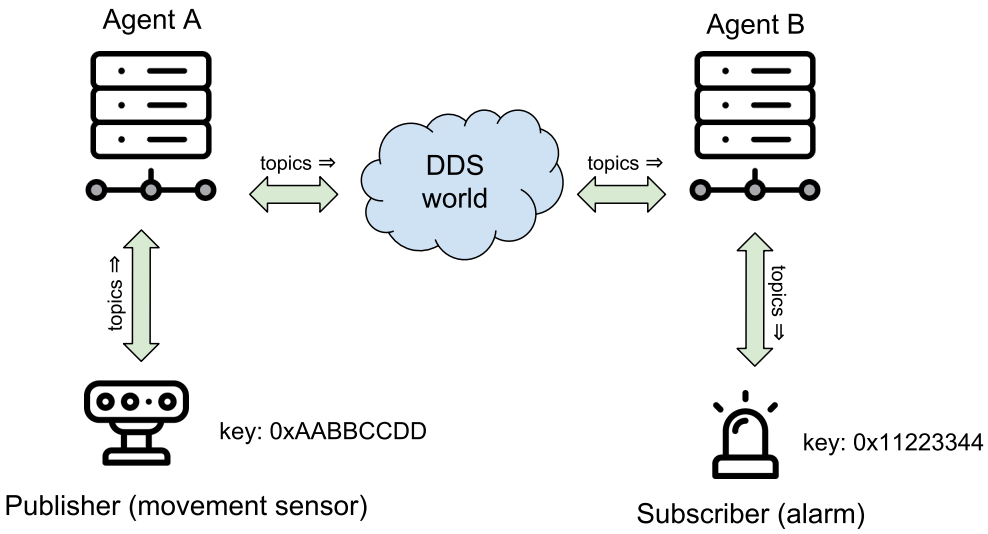
At that moment, it can use all entities created related to this *client key*. Because all entities that it uses were successfully created by the *configurator client*, the *publisher client* can immediately publish to *DDS*.

Subscriber configuration



Again, the *configurator client* connects and logs in, this time to *Agent B*, now with the subscriber's key (0x11223344). In this case, the entities that the *configurator client* creates are a *participant*, a *topic*, a *subscriber*, and a *datareader*. The entities created by the *configuraton client* will be available until the session is deleted.

Subscriber



Once the subscriber is configured, the *subscriber client* logs in the *Agent B*. Since all of its entities have been created previously, it only needs to configure the read after the login. Once the data request message has been sent, the subscriber will receive the topics from the publisher through the *DDS* world.

5.6 Shapes Demo

ShapesDemo is an interactive example for testing how *eProsima Fast RTPS* working in the *DDS Global Data Space*. Because *eProsima Micro XRCE-DDS* aims to connect an *XRCE Client* to the *DDS World*, in this example, we will create a *Client* which will interact with the *Shapes Demo*. It can be found at *examples/uxr/client/ShapeDemoClient* inside of the installation directory. This interactive *Client* waits for user input indicating commands to execute.

The available commands are the following:

create_session Creates a Session, if exists, reuse it.

create_participant <participant id>: Creates a Participant on the current session.

create_topic <topic id> <participant id>: Registers a Topic using <participant id> participant.

create_publisher <publisher id> <participant id>: Creates a Publisher on <participant id> participant.

create_subscriber <subscriber id> <participant id>: Creates a Subscriber on <participant id> participant.

create_datawriter <datawriter id> <publisher id>: Creates a DataWriter on the publisher <publisher id>.

create_datareader <datareader id> <subscriber id>: Creates a DataReader on the subscriber <subscriber id>.

write_data <datawriter id> <stream id> [<x> <y> <size> <color>]: Writes data into a <stream id> using <data writer id> DataWriter.

request_data <datareader id> <stream id> <samples>: Reads <sample> topics from a <stream id> using <datareader id> DataReader,

cancel_data <datareader id>: Cancels any previous request data of <datareader id> DataReader.

delete <id_prefix> <type>: Removes object with <id prefix> and <type>.

stream, default_output_stream <stream_id>: Changes the default output stream for all messages except of write data. <stream_id> can be 1-127 for best effort and 128-255 for reliable. The streams must be initially configured.

exit: Closes session and exit.

tree, entity_tree <id>: Creates the necessary entities for a complete publisher and subscriber. All entities will have the same <id> as id.

h, help: Shows this message.

For example, to create a publisher *Client* that sends a square Topic in reliable mode, run the following commands:

```
> create_session
> create_participant 1
> create_topic 1 1
> create_publisher 1 1
> create_datawriter 1 1
> write_data 1 128 200 200 40 BLUE
```

This *Client* will publish a topic in the reliable mode that will have color BLUE, x coordinate 200, y coordinate 200, and size 40.

In case of a subscriber *Client* that receives square topics in a reliable mode, run the following:

```
> create_session
> create_participant 1
> create_topic 1 1
> create_subscriber 1 1
> create_datareader 1 1
> request_data 1 128 5
```

This *Client* will receive 5 topics in reliable mode.

To create the entities tree easily, run the command `entity_tree <id>`. For example, the following command creates the necessary entities for publishing and subscribing data with id *3*:

```
> entity_tree 3
create_participant 3
create_topic 3 3
create_publisher 3 3
create_subscriber 3 3
create_datawriter 3 3
create_datareader 3 3
```

To modify the output default stream, change it with *stream <id>*.

The maximum available streams correspond to the `CONFIG_MAX_OUTPUT_BEST_EFFORT_STREAMS` and `CONFIG_MAX_OUTPUT_RELIABLE_STREAMS` properties as CMake arguments.

> stream 1

Now the messages will be sent in best-effort mode.

5.7 eProsima Micro XRCE-DDS Client

In *eProsima Micro XRCE-DDS*, a *Client* can communicate with the DDS Network as any other DDS actor could do. *Clients* can either publish and subscribe to data Topics in the DDS Global Data Space, or act as a client/service application following a request-reply pattern.

This section explains how the *Client-Agent* communication happens through streams that can be either best-effort or reliable. After this, it is explained how users can configure *Clients* applications and the communication with the *Agent* via sets of CMake flags (`-D<parameter>=<value>`) that enable/disable profiles and/or allow customize the size of several parameters. Finally, a table is presented to explain the creation mode options of the *Clients* on the *Agent*'s side. The section is organized as follows:

- *Streams*
- *Profiles*
- *Configurations*
- *Read Access Delivery Control*
- *Creation Mode: Client*
- *Creation Policy Table*

eProsima Micro XRCE-DDS provides the user with a C API to create *eProsima Micro XRCE-DDS Clients* applications. Find the full Client API in the *dedicated page*.

5.7.1 Streams

The *Client-Agent* communication is performed by streams. The streams can be seen as communication channels. There are two types of streams: best-effort and reliable. The user can define a maximum of 127 best-effort streams and 128 reliable streams, but for the majority of purposes, only one stream in either best-effort or reliable mode is used.

Best-effort streams Best-effort streams send and receive the data leaving the reliability to the transport layer. As a result, they consume fewer resources than reliable streams. Also, no history is stored and so the message size sent or received by a best-effort stream must be less or equal than the *MTU* defined in the transport layer.

Reliable streams Reliable streams perform the communication without loss, regardless of the transport layer used, and allow for message fragmentation to send and receive messages longer than the *MTU*.

To avoid loss of data, reliable streams use additional messages to confirm the delivery. Moreover, reliable streams have a history associated, used to store messages that can not be processed due to issues such as delivery order or incomplete fragments or messages that can not be confirmed yet. The size of the history can be tailored to fit the specific requirements of the application. The size of the stream corresponds to the *MTU* defined in the transport layer times the history.

If the history is full:

- The messages written to the *Agent* will be discarded until the history is freed and has space to store the new messages.
- The messages received from the agent will be discarded. The library will try to recover the discarded messages requesting them to the agent (increasing the bandwidth consumption in the process).

Summarizing:

- A short history causes more messages to be discarded, increasing the data traffic because they need to be sent again. At the same time, it consumes less memory.
- A long history will reduce the traffic of confirmation messages when the loss rate is high.

This internal management of the communication implies that a reliable stream is more expensive than a best-effort stream, in both memory and bandwidth, but it is possible to play with these values using the history size.

The streams are probably the highest memory load part of the application. For that, the choice of a right configuration for the application is highly recommendable, especially when the target is a limited resource device. The *Memory optimization* page explains more in detail how to achieve this.

5.7.2 Profiles

The *Client* library follows a profile concept that enables to choose, add or remove some features at **compile-time**, thus allowing to customize the *Client* library size, if there are features that are not used.

The profiles can be chosen using CMake arguments and start with the prefix `UCLIENT_PROFILE` (`-D<parameter>=<value>`) before the compilation.

By means of these profiles, the user can choose which transport to use, and whether to enable or not the discovery and framing functionalities.

Definition	Description	Values	Default
`UCLIENT_PROFILE_UDP`	Enables or disables the possibility to connect with the *Agent* by UDP.	`<bool>`	`ON`
`UCLIENT_PROFILE_TCP`	Enables or disables the possibility to connect with the *Agent* by TCP.	`<bool>`	`ON`
`UCLIENT_PROFILE_SERIAL`	Enables or disables the possibility to connect with the *Agent* by Serial.	`<bool>`	`ON`
`UCLIENT_PROFILE_CAN`	Enables or disables the possibility to connect with the *Agent* by CAN FD.	`<bool>`	`OFF`
`UCLIENT_PROFILE_CUSTOM_TRANSPORT`	Enables or disables the possibility to connect with the *Agent* by Custom Transport.	`<bool>`	`ON`
`UCLIENT_PROFILE_DISCOVERY`	Enables or disables the functions of the discovery feature (currently, only for POSIX).	`<bool>`	`ON`
`UCLIENT_PROFILE_STREAM_FRAMING`	Enables or disables the stream framing protocol.	`<bool>`	`ON`
`UCLIENT_PROFILE_MULTITHREAD`	Enables or disables the multithread locking operation of the library.	`<bool>`	`ON`
`UCLIENT_PROFILE_SHARED_MEMORY`	Enables or disables a basic local memory transport operation between entities in the same application.	`<bool>`	`ON`

Transport profiles

The implementation of the transport depends on the platform. As mentioned in the *Introductory page*, the *Client* is supported by the following platforms: Linux, Windows, FreeRTOS, Zephyr and NuttX. Linux and all three RTOSes present a POSIX-compliant API to some degree. Find below a table summarizing the compatibility of each these Operating Systems, according to their POSIX compliance, with the transports supported by the *eProsima Micro XRCE-DDS Client*.

The table below shows the current implementation.

Transport	POSIX	Windows
UDP	X	X
TCP	X	X
Serial	X	
CAN FD	X	
Custom	X	X

Each available transport can be activated or desactivated via the opportune CMake flag: `UCLIENT_PROFILE_<transport>`, where `<transport> = UDP, TCP, SERIAL, CAN`, or `UCLIENT_PROFILE_CUSTOM_TRANSPORT` in the case Custom transport is to be used.

eProsima Micro XRCE-DDS provides a user API that allows interfacing with the lowest level transport layer at runtime. In this way, a user is enabled to implement its own transports based on one of the two communication approaches: stream-oriented or packet-oriented. By means of this API, a user can set four callbacks which will be in charge of opening and closing the transport, and writing and reading from it. This custom transport API is enabled by setting the CMake argument `UCLIENT_PROFILE_CUSTOM_TRANSPORT=<bool>` to true. In the case that stream-oriented transport is used `UCLIENT_PROFILE_STREAM_FRAMING=<bool>` should also be enabled.

Find out more in the *Transport* section of the *Client API*.

Discovery profile

The discovery profile allows discovering *Agents* in the network by UDP. The reachable *Agents* will respond to the discovery call sending information about themselves, as their IP and port. This can happen in two ways: multicast or unicast. The discovery phase can be performed before the *uxr_create_session* call to determine the *Agent* to connect with. The declaration of these functions can be found in `uxr/client/profile/discovery/discovery.h`. This profile is enabled when the `UCLIENT_DISCOVERY_PROFILE` is `ON`.

Find out more in the *dedicated section* of the API.

Note: This feature is only available on Linux.

Framing profile

The framing profile enables *HDLC Framing* for using *stream-oriented transports* such as Serial transports or Custom transports that require framing.

Multithread profile

The multithread profile enables the thread-safe operation with the Micro XRCE-DDS Client library. It lockguards all the critical sections of the API and allows the usage from concurrent tasks.

Shared memory profile

The multithread profile enables a simple intraprocess communication. This profile is intended to be used whithin devices without memory protection units where all tasks or processes have access to the whole memory space.

5.7.3 Configurations

There are several definitions for configuring and building the *Client* library at **compile-time**. These definitions allow users to create a version of the library according to their requirements. These parameters can be selected as CMake flags (`-D<parameter>=<value>`) before the compilation. By means of these flags, the user can change the default value of all the parameters listed below.

Definition	Description	Values	Default
UCLIENT_MAX_OUTPUT_BEST_EFFORT_STREAMS	Configures the maximum output best-effort streams that a session could have. The calls to the uxr_create_output_best_effort_stream function for a session must be less than or equal to this value.	<number>	1
UCLIENT_MAX_OUTPUT_RELIABLE_STREAMS	Configures the maximum output reliable streams that a session could have. The calls to the uxr_create_output_realiable_stream function for a session must be less than or equal to this value.	<number>	1
UCLIENT_MAX_INPUT_BEST_EFFORT_STREAMS	Configures the maximum input best-effort streams that a session could have. The calls to the uxr_create_input_best_effort_stream function for a session must be less than or equal to this value.	<number>	1
UCLIENT_MAX_INPUT_RELIABLE_STREAMS	Configures the maximum input reliable streams that a session could have. The calls to the uxr_create_input_realiable_stream function for a session must be less than or equal to this value.	<number>	1
UCLIENT_MAX_SESSION_CONNECTION_ATTEMPTS	This value indicates the number of attempts that create_session and delete_session will perform until receiving a status message.	<number>	10
UCLIENT_MIN_SESSION_CONNECTION_INTERVAL	This value represents how long it will take to send a new create_session or delete_session if the first attempt was left answered.	<number>	1000
UCLIENT_MIN_HEARTBEAT_TIME_INTERVAL	In reliable communication, this value represents how long it will take for the first heartbeat to be sent. The wait time for the next heartbeat will be double. It is measured in milliseconds.	<number>	100
UCLIENT_BIG_ENDIANNESS	This value must correspond to the memory endianness of the device in which the *Client* is running. OFF implies that the machine is little-endian and ON implies big-endian.	<bool>	OFF
UCLIENT_UDP_TRANSPORT_MTU	This value corresponds to the *Maximum Transmission Unit (MTU)* that can be sent and/or received by UDP. It is measured in bytes and, internally, it corresponds to the creation of a buffer this size.	<number>	512
UCLIENT_TCP_TRANSPORT_MTU	This value corresponds to the *Maximum Transmission Unit (MTU)* that can be sent and/or received by TCP. It is measured in bytes and, internally, it corresponds to the creation of a buffer this size.	<number>	512
UCLIENT_SERIAL_TRANSPORT_MTU	This value corresponds to the *Maximum Transmission Unit (MTU)* that can be sent and/or received by Serial. It is measured in bytes and, internally, it corresponds to the creation of a buffer this size.	<number>	512
UCLIENT_CUSTOM_TRANSPORT_MTU	This value corresponds to the *Maximum Transmission Unit (MTU)* that can be sent and/or received by Custom transport. It is measured in bytes and, internally, it corresponds to the creation of a buffer this size.	<number>	512
UCLIENT_SHARED_MEMORY_MAX_ENTITIES	This value corresponds to the *Max number of entities involved in shared memory*.	<number>	4
UCLIENT_SHARED_MEMORY_MAX_BUFFERS	This value corresponds to the *Max number data buffers stored in shared memory*.	<number>	10
UCLIENT_HARD_LIVELINESS_CHECK	Enables Micro XRCE-DDS Client hard liveliness check.	<bool>	OFF
UCLIENT_HARD_LIVELINESS_CHECK_TIMEOUT	Sets Micro XRCE-DDS Client hard liveliness check timeout in milliseconds. Maximum value is 999999 ms.	<number>	10000

Note: The MTU of the CAN transport is fixed to 64 bytes, which is the maximum payload supported by CAN FD frames. Take this into account to calculate the size of the streams for the requirements of the application.

5.7.4 Read Access Delivery Control

The Read Access Delivery Control handles the read operation from a *datareader* previously created on the *Agent* to fetch data from the middleware. It comes with an optional `control` argument, that allows the *Client* setting the following parameters:

- `max_bytes_per_second`: Maximum rate at which data messages may be returned, measured in bytes per secpond.
- `max_elapsed_time`: Maximum amount of time that can be spent by the Agent in delivering the topic, measured in seconds.
- `max_samples`: Maximum number of topics that the Agent can send to the Client.
- `min_pace_period`: Minimum elapsed time between two topics deliveries, measured in milliseconds,.

For more information, consult the *Read access* of the *Client API*.

5.7.5 Creation Mode: Client

The creation of *Entities* on the *Agent* which can act on behalf of the *Clients* in the DDS world can be done in three ways: by XML, by reference or by binary. In this section, we explain these three creation modes and provide guidance on their usage.

XML In the XML case, when creating the entities in the *Client* application, the user must provide each `entity` with a *const char* <entity>_xml* parameter containing a string of text with XML syntax, matching the DDS rules for creating a DDS entity with an XML profile, as explained here.

For instance, when creating a *participant* or a *topic*, the profiles shall look as follows:

```
<!-- PARTICIPANT -->
const char* participant_xml = "<dds>"
                                  "<participant>"
                                      "<rtps>"
                                          "<name>[PARTICIPANT NAME]</name>"
                                      "</rtps>"
                                  "</participant>"
                              "</dds>";

<!-- TOPIC -->
const char* topic_xml = "<dds>"
                            "<topic>"
                                "<name>[TOPIC NAME]</name>"
                                "<dataType>[TOPIC TYPE]</dataType>"
                            "</topic>"
                        "</dds>"
```

As detailed in the *Getting started* section, *participants*, *topics*, *datawriters*, *datareaders*, *requesters* and *repliers* work similarly. *Publishers* and *subscribers*, instead, inherit their XML fields from their associated *dataWriters* and *dataReaders*.

Creation by XML has the advantage of being configurable direclty within the *Client* application, but comes with the drawback of offering a very limited set of options as regards the QoS with which the DDS entities profiles can be configured. Indeed, only best-effort or reliable communication streams can be set with this creation mode. In many cases, these QoS configurations alone may not be enough. For these cases, *eProsima Micro XRCE-DDS* allows the users to use the creation by references mode.

References Creation by references happens by feeding the *Agent* with an XML profile containing a string of text similar to the snippets provided above, with a label associated to it. Therefore, when creating an entity, the

Client will only need to provide a reference to this label in spite of the complete XML profile. This creation mode comes with two advantages:

- It consumes less *Client* memory, making the application more lightweight.
- It allows the *Clients* to write their own XML QoS and run the *Agent* with a custom configuration which can benefit of the *full set* of QoS available in DDS.

For instance, when creating a *participant* or a *topic*, the profiles shall look as follows:

```
<!-- PARTICIPANT -->
const char* participant_ref = "participant_label";

<!-- TOPIC -->
const char* topic_ref = "topic_label"
```

Binary

Creation by binary provides a comprehensive API in the Micro XRCE-DDS Client library that can be used to generate and send over the XRCE-DDS middleware binary representations of the entities that are being created. This creation mode comes with two advantages:

- It consumes less *Client* memory than XML mode, making the application more lightweight.
- It provides much more flexibility than the REF mode in the client side.

For instance, when creating a *participant* or a *topic*, the profiles shall look as follows:

```
uxrQoS_t qos = {
  .reliability = UXR_RELIABILITY_RELIABLE, .durability = UXR_DURABILITY_
↪TRANSIENT_LOCAL,
  .history = UXR_HISTORY_KEEP_LAST, .depth = 0
};
uxr_buffer_create_topic_bin(&session, reliable_out, topic_id, participant_id,
↪ "ExampleTopic", "ExampleType", UXR_REPLACE);
uxr_buffer_create_datawriter_bin(&session, reliable_out, datawriter_id,␣
↪publisher_id, topic_id, qos, UXR_REPLACE);
```

Find more information in the *Creation Mode: Agent* section in the *eProsima Micro XRCE-DDS Agent* page.

5.7.6 Creation Policy Table

The following table summarizes the behaviour of the *Agent* under entity creation request.

Creation flags	Entity exists	Result
Don't care	NO	Entity is created.
`0`	YES	No action is taken, and `UXR_STATUS_ERR_ALREADY_EXITS` is returned.
`UXR_REPLACE`	YES	Existing entity is deleted, requested entity is created and `UXR_STATUS_OK` is returned.
`UXR_REUSE`	YES	If entity matches no action is taken and `UXR_STATUS_OK_MATCHED` is returned. If entity does not match any action is taken and `UXR_STATUS_ERR_MISMATCH` is returned.
`UXR_REUSE | UXR_REPLACE`	YES	If entity matches no action is taken and `UXR_STATUS_OK_MATCHED` is returned. If entity does not match, exiting entity is deleted, requested entity is created and `UXR_STATUS_OK` is returned.

5.8 eProsima Micro XRCE-DDS Agent

The *eProsima Micro XRCE-DDS Agent* acts as a server between the DDS Network and *eProsima Micro XRCE-DDS Clients* applications. The *Agents* receive messages containing operations from the *Clients*, and keep track of the *Clients* and of the entities they create. These entities are used by the *Agents* to interact with the DDS Global Data Space on behalf of the *Clients*.

The communication between a *Client* and an *Agent* currently supports UDP, TCP, Serial, CAN FD, and Custom transports, depending on the peripherals and communication technologies offered by the platforms. A section dedicated to the configuration and use of the Custom Transport can be found at the end of this *page*.

While running, the *Agent* attends any received requests from the *Clients* and answers back with the result of those requests.

This section is organized as follow:

- *Agent CLI*
- *Custom transport*
- *Configuration*
- *Creation Mode: Agent*

- *Middleware Abstraction Layer*

5.8.1 Agent CLI

To run the *Agent*, first of all build it as indicated in the *Installation* page. Once it is built successfully, launch it by executing one of the following commands:

UDP transport The communication via UDP can be executed using two modes, IPv4 and IPv6; and configured as follows:

```
$ ./MicroXRCEAgent [ udp4 | udp6 ]  [OPTIONS]

Options:
  -h,--help                                    Print the help message.
  -p,--port UINT REQUIRED                      Select the IP port.
  -m,--middleware TEXT in {ced,rtps,dds}=dds   Select the kind of middleware␣
↪among the supported ones. By default, it will be FastDDS.
  -r,--refs FILEPATH                           Load a references file from the␣
↪given path.
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4       Select log level from none (0) to␣
↪full verbosity (6).
  -d,--discovery UINT=7400                     Activate the Discovery server. If␣
↪no port is specified, 7400 will be used.
  --p2p UINT                                   Activate the P2P profile, using␣
↪the given port.
```

TCP transport The communication via TCP can be executed using two modes, IPv4 and IPv6; and configured as follows:

```
$ ./MicroXRCEAgent [ tcp4 | tcp6 ] [OPTIONS]

Options:
  -h,--help                                    Print the help message.
  -p,--port UINT REQUIRED                      Select the IP port.
  -m,--middleware TEXT in {ced,rtps,dds}=dds   Select the kind of middleware␣
↪among the supported ones. By default, it will be FastDDS.
  -r,--refs FILEPATH                           Load a references file from the␣
↪given path.
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4       Select log level from none (0) to␣
↪full verbosity (6).
  -d,--discovery UINT=7400                     Activate the Discovery server. If␣
↪no port is specified, 7400 will be used.
  --p2p UINT                                   Activate the P2P profile, using␣
↪the given port.
```

Communication via Serial transport (only Linux) The communication via Serial transport can be executed and configured as follows:

```
$ ./MicroXRCEAgent serial [OPTIONS]

Options:
  -h,--help                                    Print the help message.
  -D,--dev FILE REQUIRED                       Specify the serial device.
  -f,--file FILE REQUIRED                      Specify a text file with the␣
↪serial device name.
  -b,--baudrate TEXT=115200                    Select the baudrate.
  -m,--middleware TEXT in {ced,rtps,dds}=dds   Select the kind of middleware␣
↪among the supported ones. By default, it will be FastDDS.
```

(continues on next page)

(continued from previous page)

```
  -r,--refs FILEPATH                              Load a references file from the␣
↪given path.
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4          Select log level from none (0) to␣
↪full verbosity (6).
```

Note: The *Agent* will check and wait for the proper availability of the Serial port to start the connection. Its expected to start the transport with a disconnected Serial port.

Communication via Multiserial transport (only Linux) This transport allows multiple serial connections on the same *Agent* instance. The communication via Multiserial transport can be executed and configured as follows:

```
$ ./MicroXRCEAgent multiserial [OPTIONS]

Options:
  -h,--help                                       Print the help message.
  -D,--devs FILE REQUIRED                         Specify the serial devices.
  -f,--file FILE REQUIRED                         Specify a text file with one␣
↪serial device per line.
  -b,--baudrate TEXT=115200                       Select the baudrate.
  -m,--middleware TEXT in {ced,rtps,dds}=dds      Select the kind of middleware␣
↪among the supported ones. By default, it will be FastDDS.
  -r,--refs FILEPATH                              Load a references file from the␣
↪given path.
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4          Select log level from none (0) to␣
↪full verbosity (6).
```

Note: The *Agent* will check and wait for the proper availability of each Serial port to start the connection. Its expected to start the transport with multiple disconnected ports.

Communication via CAN FD transport (only Linux) The communication via CAN FD transport can be executed and configured as follows:

```
$ ./MicroXRCEAgent canfd [OPTIONS]

Options:
  -h,--help                                       Print the help message.
  -D,--dev INTERFACE REQUIRED                     Specify the CAN interface.
  -m,--middleware TEXT in {ced,rtps,dds}=dds      Select the kind of middleware␣
↪among the supported ones. By default, it will be FastDDS.
  -r,--refs FILEPATH                              Load a references file from the␣
↪given path.
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4          Select log level from none (0) to␣
↪full verbosity (6).
```

Note: The used interface must support CAN FD frames with a maximum payload of 64 bytes. The agent will use the received message identifiers from each client on its output frames.

Communication via pseudo terminal (only Linux) The communication via pseudo serial can be executed and configured as follow:

```
$ ./MicroXRCEAgent pseudoterminal [OPTIONS]

Options:
  -h,--help                                    Print the help message.
  -D,--dev FILE REQUIRED                       Specify the pseudo serial device.
  -b,--baudrate TEXT=115200                    Select the baudrate.
  -m,--middleware TEXT in {ced,rtps,dds}=dds   Select the kind of middleware␣
↪among the supported ones. By default, it will be FastDDS.
  -r,--refs FILEPATH                           Load a references file from the␣
↪given path.
  -v,--verbose UINT in {0,1,2,3,4,5,6}=4       Select log level from none (0) to␣
↪full verbosity (6).
```

- The reference file shall be composed by a set of Fast DDS profiles following the XML syntax described in the *eProsima Fast DDS* documentation. The `profile_name` attribute of each profile represents a reference to an XRCE entity, so that it can be used by the *Clients* to create entities by reference.
- The `-b, --baudrate <baudrate>` options sets the baud rate of the communication. It can take the following values: 0, 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 240, 4800, 9600, 19200, 38400, 57600, 115200 (default), 230400, 460800, 500000, 576000, 921600, 1000000, 1152000, 1500000, 2000000, 2500000, 3000000, 3500000 or 4000000 bauds.
- The `-v, --verbose <level[0-6]>` option sets log level from less to more verbose, where level 0 corresponds to the logger being off. Then, from 1 to 6, the following logging levels are activated: *critical*, *error*, *warning*, *info*, *debug* and *trace*.
- The option `-m, --middleware <middleware-impl>` sets the middleware implementation to use. There are three: RTPS (based on eProsima Fast RTPS), DDS (specified by the XRCE standard and using Fast DDS) and Centralized (topic are managed by the Agent similarly to MQTT). More information about the supported middlewares can be found *here*.
- The `--p2p <port>` option enables P2P communication, this option is only available on network transports. Centralized middleware is necessary for this option.

5.8.2 Custom transport

If none of the transports specified above is suitable for the target application, users can easily create an instance of a *Micro XRCE-DDS Agent*, together with a custom transport implementation.

For this purpose, the `eprosima::uxr::CustomAgent` class was developed. It follows the policy of giving users function signatures to implement, which hide as much as possible the underneath implementation details of the *Agent*. Thus, this methods provide common parameters used when implementing a receive/send message method, such as an octet pointer to a raw data buffer, buffer/message length, timeout, and so on.

More details on how to implement a custom transport can be found in the *Custom Transport* Agent's section of this documentation.

5.8.3 Configuration

There are several parameters which can be set at **compile-time** to configure the *eProsima Micro XRCE-DDS Agent*. These parameters can be selected as CMake flags (`-D<parameter>=<value>`) before the compilation. The following is a table listing these parameters and the functionalities they carry out:

Definition	Description	Values	Default
`UAGENT_CONFIG_RELIABLE_STREAM_DEPTH`	Specifies the history of the reliable streams.	`<number>`	16
`UAGENT_CONFIG_BEST_EFFORT_STREAM_DEPTH`	Specifies the history of the best-effort streams.	`<number>`	16
`UAGENT_CONFIG_HEARTBEAT_PERIOD`	Specifies the `HEARTBEAT` message period in millisecond.	`<number>`	200
`UAGENT_CONFIG_TCP_MAX_CONNECTIONS`	Specifies the maximum number of connections that the *Agent* can manage.	`<number>`	100
`UAGENT_CONFIG_TCP_MAX_BACKLOG_CONNECTIONS`	Specifies the maximum number of incoming connections (pending to be established) that the *Agent* can manage.	`<number>`	100
`UAGENT_CONFIG_SERVER_QUEUE_MAX_SIZE`	Maximum server's queues size.	`<number>`	32000
`UAGENT_CONFIG_CLIENT_DEAD_TIMEOUT`	Client dead time in milliseconds.	`<number>`	30000
`UAGENT_CONFIG_CLIENT_DEAD_TIMEOUT`	Client dead time in milliseconds.	`<number>`	30000
`UAGENT_SERVER_BUFFER_SIZE`	Server buffer size.	`<number>`	65535

5.8.4 Creation Mode: Agent

As explained in the *Creation Mode: Client* section in the *eProsima Micro XRCE-DDS Client* page, the creation of *Entities* on the *Agent* can be done in two ways: by XML, or by reference. While the creation by XML is configured directly on the *Client*, creation by reference must be configured on the *Agent*, via an `agent.refs` file which must be loaded as a CLI parameter by using the `-r` option followed by the path to the reference file. If a Custom trasnport is used, the `agent.refs` file must be fed to the :ref:`load_config_file <load_config_file>` function defined in the *Agent*.

The `agent.refs` file should define the desired profiles as follows:

```
<profiles>
    <participant profile_name="default_xrce_participant">
        <rtps>
            <name>default_xrce_participant</name>
        </rtps>
    </participant>
    <data_writer profile_name="shapetype_data_writer">
        <topic>
            <kind>WITH_KEY</kind>
            <name>Square</name>
            <dataType>ShapeType</dataType>
        </topic>
    </data_writer>
    <data_reader profile_name="shapetype_data_reader">
        <topic>
            <kind>WITH_KEY</kind>
            <name>Square</name>
            <dataType>ShapeType</dataType>
        </topic>
    </data_reader>
    <topic profile_name="shapetype_topic">
        <kind>WITH_KEY</kind>
        <name>Square</name>
```

(continues on next page)

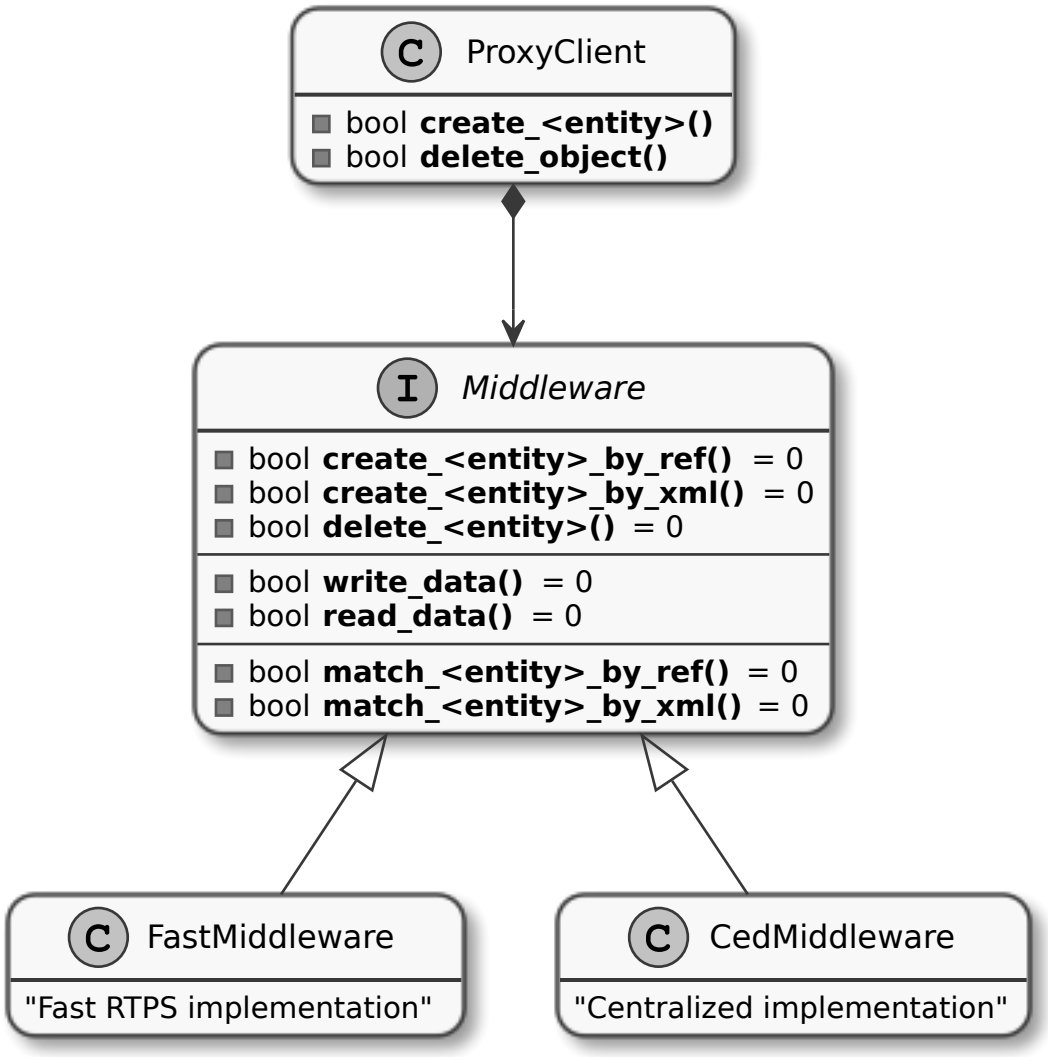
(continued from previous page)

```
        <dataType>ShapeType</dataType>
    </topic>
    <requester profile_name="shapetype_requester"
               service_name="shapetype_service"
               request_type="request_type"
               reply_type="reply_type">
    </requester>
    <replier profile_name="shapetype_replier"
             service_name="shapetype_service"
             request_type="request_type"
             reply_type="reply_type">
    </replier>
</profiles>
```

In the reference file, each entity must be associated to a `profile_name` which serves as a label to which the *Client* can refere when creating entities.

5.8.5 Middleware Abstraction Layer

The Middleware Abstraction Layer is an interface whose purpose is to isolate the XRCE core from the middleware, as well as to allow providing multiple middleware implementations. The interface has a set of pure virtual functions, which are called by the *ProxyClient* each time a *Client* requests to create/delete an entity or to write/read data.



For the moment, the *Agent* counts with two active middleware implementations (*FastDDSMiddleware* and *CedMiddleware*) and another one that is currently deprecated (*FastMiddleware*).

FastDDSMiddleware

The *FastDDSMiddleware* uses *eProsima Fast DDS*, a C++ implementation of the DDS standard.

This middleware allows the *Clients* to produce and consume data in the DDS Global Data Space, and as such also in the ROS 2 ecosystem. The *Agent* has the behaviour described in the *DDS-XRCE* standard, that is, for each *DDS-XRCE* entity a DDS proxy entity is created, and the writing/reading action produces a publishing/subscribing operation in the DDS world.

CedMiddleware

The *CedMiddleware* (Centralized Middleware) works similar to MQTT, that is, the *Agent* acts as a broker but has no output to the DDS world. It:

- Accepts connection from the *Clients*,
- Accepts messages published by the *Clients*,
- Processes *subscribe* and *unsubscribe* requests from the *Clients*,
- Forwards messages that match the *Clients*' subscriptions,
- Closes the connection opened by the *Clients*.

By default, this middleware does not allow communication between *Clients* connected to different *Agents*, but the *P2P communication* enables this feature.

FastMiddleware

The *FastMiddleware* uses *eProsima Fast RTPS*, a C++ implementation of the RTPS (Real Time Publish Subscribe) protocol. This middleware allows *Client* to produce and consume data in the DDS Global Data Space, and as such also in the ROS 2 ecosystem. As in the case of the *FastDDSMiddleware*, the *Agent* has the behaviour described in the *DDS-XRCE* standard, that is, for each *DDS-XRCE* entity a DDS proxy entity is created, and the writing/reading action produces a publishing/subscribing operation in the DDS world.

Warning: This implementation is deprecated at the moment.

How to add a middleware

Adding a new middleware implementation is quite simple, if the steps below are followed:

1. Create a class that implement the *Middleware* class (see *inclue/uxr/agent/middleware/fast/FastMiddleware.hpp* and *src/cpp/middleware/fast.cpp* as examples).
2. Add a *enum* member protected by defines in *Middleware::Kind* at *include/uxr/agent/middleware/Middleware.hpp*.
3. Add a case in the switch of the *ProxyClient* constructor at *src/cpp/client/ProxyClient.cpp*.
4. In *CMakeLists.txt* add an option similar to *UAGENT_FAST_PROFILE* and add the source to *SRCS* variable.
5. In *include/uxr/agent/config.hpp.in* add a *#cmakedefine* with the name of the CMake option.
6. Finally, add the CLI middleware option in *MiddlewareOpt* constructor at *include/uxr/agent/utils/CLI.hpp*.

5.9 API

This section provides the detailed information for programming *Client* and *Agent* applications with the API provided by *eProsima Micro XRCE-DDS*.

5.9.1 Client API

eProsima Micro XRCE-DDS provides the user with a C API to create *eProsima Micro XRCE-DDS Clients* applications. **All** functions needed to set up the *Client* can be found in the `client.h` header. That is the only header the user needs to include.

In this section, we provide the full API for the *Micro XRCE-DDS Client*. As a nomenclature, this API uses the `uxr_` prefix in all of its public functions and the `uxr` prefix in the types. In constants values, the `UXR_` prefix is used. The functions belonging to the public interface of the library are only those with the tag `UXRDDLAPI` in their declarations.

The functions are grouped as follows:

- *Session*
- *Create entities by XML*
- *Create entities by reference*
- *Create entities by binary*
- *Create entities common profile*
- *Read access*
- *Write access profile*
- *Discovery profile*
- *Topic serialization*
- *General utilities*
- *Transport*

Session

These functions are available even if no profile has been enabled. The declaration of these functions can be found in `uxr/client/core/session/session.h`.

```
void uxr_init_session(uxrSession* session, uxrCommunication* comm, uint32_t key);
```

Initializes a session structure. Once this function is called, a `create_session` call can be performed.

session Session structure where to manage the session data.

comm Communication used for connecting to the *Agent*. All different transports have a common attribute `uxrCommunication`. This parameter can not be shared between active sessions.

key The key identifier of the *Client*. All *Clients* connected to an *Agent* must have a different key.

```
void uxr_set_status_callback(uxrSession* session, uxrOnStatusFunc on_status_func,␣
↪void* args);
```

Assigns the callback for the *Agent* status messages.

session Session structure previously initialized.

on_status_func Function callback that is called when a valid status message comes from the *Agent*.

args User pointer data. The args are provided to the `on_status_func` function.

The function signature for the `on_status_func` callback is:

```
typedef void (*uxrOnStatusFunc) (struct uxrSession* session, uxrObjectId object_id,␣
↪uint16_t request_id,
                                 uint8_t status, void* args);
```

session Session structure related to the status.

object_id The identifier of the entity related to the status.

request_id Status request id.

status Status value.

args User pointer data.

```
void uxr_set_topic_callback(uxrSession* session, uxrOnTopicFunc on_topic_func, void*␣
↪args);
```

Assigns the callback for topics. The topics are received only if a `request_data` function has been called.

session Session structure previously initialized.

on_status_func Function callback that is called when a valid data message comes from the *Agent*.

args User pointer data. The args are provided to the `on_topic_func` function.

The function signature for the `on_topic_func` callback is:

```
typedef void (*uxrOnTopicFunc) (struct uxrSession* session, uxrObjectId object_id,␣
↪uint16_t request_id, uxrStreamId stream_id,
                                struct ucdrBuffer* ub, uint16_t length, void* args);
```

session Session structure related to the topic.

object_id The identifier of the entity related to the topic.

request_id Request id of the``request_data`` transaction.

stream_id Id of the stream used for the communication.

ub Serialized topic data.

length Length of the serialized data.

args User pointer data.

```
void uxr_set_time_callback(uxrSession* session, uxrOnTimeFunc on_time_func, void*␣
↪args);
```

Assigns the time callback, to let the user perform custom time calculations based on client and agent timestamps.

session Session structure previously initialized.

on_time_func Function callback that is called .. ?

args User pointer data. The args are provided to the `on_time_func` function.

The function signature for the `on_time_func` callback is:

```
typedef void (*uxrOnTimeFunc) (struct uxrSession* session, int64_t current_timestamp,␣
↪int64_t transmit_timestamp,
                               int64_t received_timestamp, int64_t originate_
↪timestamp, void* args);
```

session Session structure related to the topic.

current_timestamp Client's timestamp of the response packet reception.

transmit_timestamp Client's timestamp of the request packet transmission.

received_timestamp Agent's timestamp of the request packet reception.

originate_timestamp Agent's timestamp of the response packet transmission.

args User pointer data.

```
void uxr_set_request_callback(uxrSession* session, uxrOnRequestFunc on_request_func,␣
↪void* args);
```

Sets the request callback, which is called when the *Agent* sends a `READ_DATA` submessage associated with a `Requester`.

session Session structure previously initialized.

on_request_func Function callback that is called when the *Agent* sends a `READ_DATA` submessage associated with a `Requester`.

args User pointer data. The args are provided to the `on_request_func` function.

The function signature for the `on_request_func` callback is:

```
typedef void (*uxrOnRequestFunc) (struct uxrSession* session, uxrObjectId object_id,␣
↪uint16_t request_id,
                                  SampleIdentity* sample_id, struct ucdrBuffer* ub,␣
↪uint16_t length, void* args);
```

session Session structure related to the topic.

object_id The identifier of the entity related to the request.

request_id Request id of the``request_data`` transaction.

sample_id Identifier of the request.

ub Serialized request data.

length Length of the serialized data.

args User pointer data.

```
void uxr_set_reply_callback(uxrSession* session, uxrOnReplyFunc on_reply_func, void*␣
↪args);
```

Sets the reply callback, which is called when the *Agent* sends a `READ_DATA` submessage associated with a `Replier`.

session Session structure previously initialized.

on_reply_func Function callback that is called when the *Agent* sends a `READ_DATA` submessage associated with a `Replier`

args User pointer data. The args are provided to `on_reply_func` function.

```
typedef void (*uxrOnReplyFunc) (struct uxrSession* session, uxrObjectId object_id,␣
↪uint16_t request_id, uint16_t reply_id,
                                struct ucdrBuffer* ub, uint16_t length, void* args);
```

session Session structure related to the topic.

object_id The identifier of the entity related to the request.

request_id Request id of the``request_data`` transaction.

reply_id Identifier of the reply.

ub Serialized request data.

length Length of the serialized data.

args User pointer data.

```
bool uxr_create_session(uxrSession* session);
```

Creates a new session on the *Agent*. This function logs in a session, enabling any other *XRCE* communication with the *Agent*.

session Session structure previously initialized.

```
void uxr_create_session_retries(uxrSession* session, size_t retries);
```

Attempts to establish a new session on the *Agent* `retries` times. This function logs in a session, enabling any other *XRCE* communication with the *Agent*.

session Session structure previously initialized.

retries Number of attempts for creating a session.

```
bool uxr_delete_session(uxrSession* session);
```

Deletes a session previously created. All *XRCE* entities created with the session are removed. This function logs out a session, disabling any other *XRCE* communication with the *Agent*.

session Session structure previously initialized and created.

```
bool uxr_delete_session_retries(uxrSession* session, size_t retries);
```

Attempts to delete a previously created session `retries` times. All *XRCE* entities created with the session are removed. This function logs out a session, disabling any other *XRCE* communication with the *Agent*.

session Session structure previously initialized and created.

retries Number of attempts for deleting a session.

```
uxrStreamId uxr_create_output_best_effort_stream(uxrSession* session, uint8_t* buffer,
↪ size_t size);
```

Creates and initializes an output best-effort stream for writing. The `uxrStreamId` returned represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_OUTPUT_BEST_EFFORT_STREAMS` CMake argument.

session Session structure previously initialized and created.

buffer Memory block where the messages are written.

size Buffer size.

```
uxrStreamId uxr_create_output_reliable_stream(uxrSession* session, uint8_t* buffer,␣
↪size_t size, size_t history);
```

Creates and initializes an output reliable stream for writing. The `uxrStreamId` returned represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_OUTPUT_RELIABLE_STREAMS` CMake argument.

session Session structure previously initialized and created.

buffer Memory block where the messages are written.

size Buffer size.

history History used for reliable connection. The buffer size is split into a `history` number of smaller buffers. The history must be a divisor of the buffer size and a power of two.

```
uxrStreamId uxr_create_input_best_effort_stream(uxrSession* session);
```

Creates and initializes an input best-effort stream for receiving messages. The `uxrStreamId` returned represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_INPUT_BEST_EFFORT_STREAMS` CMake argument.

session Session structure previously initialized and created.

```
uxrStreamId uxr_create_input_reliable_stream(uxrSession* session, uint8_t* buffer,␣
↪size_t size, size_t history);
```

Creates and initializes an input reliable stream for receiving messages. The returned `uxrStreamId` represents the new stream and can be used to manage it. The number of available calls to this function must be less or equal than `CONFIG_MAX_INPUT_RELIABLE_STREAMS` CMake argument.

session Session structure previously initialized and created.

buffer Memory block where the messages are stored.

size Buffer size.

history History used for reliable connection. The buffer size is split into a `history` number of smaller buffers. The history must be a divisor of the buffer size and a power of two.

```
void uxr_flash_output_streams(uxrSession* session);
```

Flashes all output streams sending the data through the transport.

session Session structure previously initialized and created.

```
void uxr_run_session_time(uxrSession* session, int timeout_ms);
```

This function processes the internal functionality of a session. It implies:

1. Flushing all output streams sending the data through the transport.
2. If there is any reliable stream, it performs the associated reliable behaviour to ensure communication.
3. Listening to messages from the *Agent* and calling the associated callback if it exists (which can be a *status*/*topic*/*time*/*request* or *reply* callback)

The `time` suffix function version perform these actions and listens to messages for a `timeout_ms` duration, which is refreshed each time a new message is received, that is, the counter restarts for another `timeout_ms` period. Only when the wait time for a message overcomes the `timeout_ms` duration, the function finishes. The function returns `true` if the sending data have been confirmed, `false` otherwise.

session Session structure previously initialized and created.

timeout_ms Time for waiting for each new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

```
void uxr_run_session_timeout(uxrSession* session, int timeout_ms);
```

This function processes the internal functionality of a session. It implies:

1. Flushing all output streams sending the data through the transport.
2. If there is any reliable stream, it performs the associated reliable behaviour to ensure communication.
3. Listening to messages from the *Agent* and calling the associated callback if it exists (which can be a *status*/*topic*/*time*/*request* or *reply* callback)

The `timeout` suffix function version performs these actions and listens to messages for a *total* `timeout_ms` duration. Each time a new message is received, the counter retakes from where it left, that is, for a period equal to `timeout_ms` minus the time spent waiting for the previous message(s). When the *total* wait time overcomes the `timeout_ms` duration, the function finishes. The function returns `true` if the sending data have been confirmed, `false` otherwise.

session Session structure previously initialized and created.

timeout_ms Total time for waiting for a new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

```
void uxr_run_session_until_timeout(uxrSession* session, int timeout_ms);
```

This function processes the internal functionality of a session. It implies:

1. Flushing all output streams sending the data through the transport.
2. If there is any reliable stream, it performs the associated reliable behaviour to ensure communication.

3. Listening to messages from the *Agent* and calling the associated callback if it exists (which can be a *status*/*topic*/*time*/*request* or *reply* callback)

The `until_timeout` suffix function version performs these actions until receiving one message, for a `timeout_ms` time duration. Once a message has been received or the timeout has been reached, the function finishes. The function returns `true` if it has received a message, `false` if the timeout has been reached.

session Session structure previously initialized and created.

timeout_ms Maximum time for waiting for a new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

```
bool uxr_run_session_until_confirm_delivery(uxrSession* session, int timeout_ms);
```

This function processes the internal functionality of a session. It implies:

1. Flushing all output streams sending the data through the transport.
2. If there is any reliable stream, it performs the associated reliable behaviour to ensure communication.
3. Listening to messages from the *Agent* and calling the associated callback if it exists (which can be a *status*/*topic*/*time*/*request* or *reply* callback)

The `until_confirm_delivery` suffix function version performs these actions during `timeout_ms` or until the output reliable streams confirm that the sent messages have been received by the *Agent*. The function returns `true` if the sent data have been confirmed, `false` otherwise.

session Session structure previously initialized and created.

timeout_ms Maximum waiting time for a new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

```
bool uxr_run_session_until_all_status(uxrSession* session, int timeout_ms, const␣
↪uint16_t* request_list,
                                      uint8_t* status_list, size_t list_size);
```

This function processes the internal functionality of a session. It implies:

1. Flushing all output streams sending the data through the transport.
2. If there is any reliable stream, it performs the associated reliable behaviour to ensure communication.
3. Listening to messages from the *Agent* and calling the associated callback if it exists (which can be a *status*/*topic*/*time*/*request* or *reply* callback)

The `until_all_status` suffix function version performs these actions during a `timeout_ms` duration or until all requested statuses have been received. The function returns `true` if all statuses have been received and all of them have the value `UXR_STATUS_OK` or `UXR_STATUS_OK_MATCHED`, `false` otherwise.

session Session structure previously initialized and created.

timeout_ms Maximum waiting time for a new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

request_list An array of requests to confirm with a status.

status_list An uninitialized array with the same size as `request_list` where the status values are written. The position of each status in the *status_list* matches the corresponding request position in the `request_list`.

list_size The size of the `request_list` and `status_list` arrays.

```
bool uxr_run_session_until_one_status(uxrSession* session, int timeout_ms, const␣
↪uint16_t* request_list,
                                      uint8_t* status_list, size_t list_size);
```

This function processes the internal functionality of a session. It implies:

1. Flushing all output streams sending the data through the transport.
2. If there is any reliable stream, it performs the associated reliable behaviour to ensure communication.
3. Listening to messages from the *Agent* and calling the associated callback if it exists (which can be a *status*/*topic*/*time*/*request* or *reply* callback)

The `until_one_status` suffix function version performs these actions during a `timeout_ms` duration or until one requested status has been received. The function returns `true` if one status has been received and has the value `UXR_STATUS_OK` or `UXR_STATUS_OK_MATCHED`, `false` otherwise.

session Session structure previously initialized and created.

timeout_ms Maximum waiting time for a new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

request_list An array of requests to confirm with a status.

status_list An uninitialized array with the same size as `request_list` where the status values are written. The position of each status in the *status_list* matches the corresponding request position in the `request_list`.

list_size The size of the `request_list` and `status_list` arrays.

```
bool uxr_run_session_until_data(uxrSession* session, int timeout_ms);
```

This function processes the internal functionality of a session. It implies:

1. Flushing all output streams sending the data through the transport.
2. If there is any reliable stream, it operates according to the associated reliable behaviour to ensure communication.
3. Listening to messages from the *Agent* and calling the associated callback if it exists (which can be a *status*/*topic*/*time*/*request* or *reply* callback)

The `until_data` suffix function version performs these actions during a `timeout_ms` duration or until a subscription data, request or reply is received. The function returns `true` if a subscription data, request or reply is received, and `false` otherwise.

session Session structure previously initialized and created.

timeout_ms Maximum waiting time for a new message, in milliseconds. For waiting without timeout, set the value to `UXR_TIMEOUT_INF`

```
bool uxr_sync_session(uxrSession* session, int time);
```

This function synchronizes the session time with the *Agent* using the NTP protocol by default.

session Session structure previously initialized and created.

time The waiting time in milliseconds.

```
int64_t uxr_epoch_millis(uxrSession* session);
```

This function returns the epoch time in milliseconds, taking into account the offset computed during the time synchronization.

session Session structure previously initialized.

```
int64_t uxr_epoch_nanos(uxrSession* session);
```

This function returns the epoch time in nanoseconds taking into account the offset computed during the time synchronization.

session Session structure previously initialized and created.

Create entities by XML

The declaration of these functions can be found in `uxr/client/profile/session/create_entities_xml.h`.

```
uint16_t uxr_buffer_create_participant_xml(uxrSession* session, uxrStreamId stream_id,
↪ uxrObjectId object_id,
                                           uint16_t domain, const char* xml, uint8_t␣
↪mode);
```

Creates a *participant* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_PARTICIPANT_ID`.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_topic_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                     uxrObjectId participant_id, const char* xml,␣
↪uint8_t mode);
```

Creates a *topic* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_TOPIC_ID.

participant_id The identifier of the associated participant. The type must be UXR_PARTICIPANT_ID.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_publisher_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                         uxrObjectId participant_id, const char* xml,␣
↪uint8_t mode);
```

Creates a *publisher* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the uxr_flash_output_streams or the uxr_run_session function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_PUBLISHER_ID.

participant_id The identifier of the associated participant. The type must be UXR_PARTICIPANT_ID.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_subscriber_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                          uxrObjectId participant_id, const char* xml,
↪ uint8_t mode);
```

Creates a *subscriber* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the uxr_flash_output_streams or the uxr_run_session function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_SUBSCRIBER_ID.

participant_id The identifier of the associated participant. The type must be UXR_PARTICIPANT_ID.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_datawriter_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                          uxrObjectId publisher_id, const char* xml,␣
↪uint8_t mode);
```

Creates a *datawriter* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAWRITER_ID`.

publisher_id The identifier of the associated publisher. The type must be `UXR_PUBLISHER_ID`.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_datareader_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                          uxrObjectId subscriber_id, const char* xml,␣
↪uint8_t mode);
```

Creates a *datareader* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAREADER_ID`.

subscriber_id The identifier of the associated subscriber. The type must be `UXR_SUBSCRIBER_ID`.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_requester_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                         uxrObjectId participant_id, const char* xml,␣
↪uint8_t mode);
```

Creates a *requester* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_REQUESTER_ID`.

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_replier_xml(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                       uxrObjectId participant_id, const char* xml,␣
↪uint8_t mode);
```

Creates a *replier* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_REPLIER_ID`.

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`.

xml An XML representation of the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

Create entities by reference

The declaration of these functions can be found in `uxr/client/profile/session/create_entities_ref.h`.

```
uint16_t uxr_buffer_create_participant_ref(uxrSession* session, uxrStreamId stream_id,
↪ uxrObjectId object_id,
                                           const char* ref, uint8_t mode);
```

Creates a *participant* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_PARTICIPANT_ID`

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_topic_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                     uxrObjectId participant_id, const char* ref,␣
↪uint8_t mode);
```

Creates a *topic* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_TOPIC_ID`

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_datawriter_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                          uxrObjectId publisher_id, const char* ref,␣
↪uint8_t mode);
```

Creates a *datawriter* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAWRITER_ID`

publisher_id The identifier of the associated publisher. The type must be `UXR_PUBLISHER_ID`

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_datareader_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                          uxrObjectId subscriber_id, const char* ref,␣
↪uint8_t mode);
```

Creates a *datareader* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAREADER_ID`.

subscriber_id The identifier of the associated subscriber. The type must be `UXR_SUBSCRIBER_ID`.

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_requester_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                         uxrObjectId participant_id, const char* ref,␣
↪uint8_t mode);
```

Creates a *requester* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_REQUESTER_ID`.

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`.

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_replier_ref(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
                                       uxrObjectId participant_id, const char* ref,␣
↪uint8_t mode);
```

Creates a *replier* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_REPLIER_ID`.

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`.

ref A reference to the new entity.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

Create entities by binary

The declaration of these functions can be found in `uxr/client/profile/session/create_entities_ref.h`.

```
uint16_t uxr_buffer_create_participant_bin(uxrSession* session, uxrStreamId stream_id,
↪ uxrObjectId object_id,
    uint16_t domain_id, const char* participant_name, uint8_t mode);
```

Creates a *participant* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_PARTICIPANT_ID`

domain_id DDS Domain ID for the participant.

participant_name Participant name.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_topic_bin(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
    uxrObjectId participant_id, const char* topic_name, const char* type_name, uint8_
↪t mode);
```

Creates a *topic* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_TOPIC_ID`

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`

topic_name Topic name.

type_name Type name.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uint16_t uxr_buffer_create_publisher_bin(uxrSession* session, uxrStreamId␣
↪stream_id, uxrObjectId object_id,
    uxrObjectId participant_id, uint8_t mode);
```

Creates a *publisher* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_PUBLISHER_ID`.

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uint16_t uxr_buffer_create_subscriber_bin(uxrSession* session, uxrStreamId␣
↪stream_id, uxrObjectId object_id,
    uxrObjectId participant_id, uint8_t mode);
```

Creates a *subscriber* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_SUBSCRIBER_ID`.

participant_id The identifier of the associated participant. The type must be `UXR_PARTICIPANT_ID`.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_datawriter_bin(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
    uxrObjectId publisher_id, uxrObjectId topic_id, uxrQoS_t qos, uint8_t mode);
```

Creates a *datawriter* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAWRITER_ID`

publisher_id The identifier of the associated publisher. The type must be `UXR_PUBLISHER_ID`

topic_id The identifier of the associated topic. The type must be `UXR_TOPIC_ID`

qos `uxrQoS_t` struct describing QoS.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_datareader_bin(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
    uxrObjectId subscriber_id, uxrObjectId topic_id, uxrQoS_t qos, uint8_t mode);
```

Creates a *datareader* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be `UXR_DATAREADER_ID`.

subscriber_id The identifier of the associated subscriber. The type must be `UXR_SUBSCRIBER_ID`.

topic_id The identifier of the associated topic. The type must be `UXR_TOPIC_ID`

qos uxrQoS_t struct describing QoS.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_requester_bin(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
    uxrObjectId participant_id, const char* service_name, const char* request_type,␣
↪const char* reply_type,
    const char* request_topic_name, const char* reply_topic_name, uxrQoS_t qos, uint8_
↪t mode);
```

Creates a *requester* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the uxr_flash_output_streams or the uxr_run_session function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_REQUESTER_ID.

participant_id The identifier of the associated participant. The type must be UXR_PARTICIPANT_ID.

service_name Service name.

request_type Request type name.

reply_type Reply type name.

request_topic_name Request topic name.

reply_topic_name Reply topic name.

qos uxrQoS_t struct describing QoS.

mode Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the UXR_REUSE and UXR_REPLACE flags (see *Creation Policy Table*).

```
uint16_t uxr_buffer_create_replier_bin(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id,
    uxrObjectId participant_id, const char* service_name, const char* request_type,␣
↪const char* reply_type,
    const char* request_topic_name, const char* reply_topic_name, uxrQoS_t qos, uint8_
↪t mode);
```

Creates a *replier* entity in the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the uxr_flash_output_streams or the uxr_run_session function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the new entity. Later, the entity can be referenced with this id. The type must be UXR_REPLIER_ID.

participant_id The identifier of the associated participant. The type must be UXR_PARTICIPANT_ID.

service_name Service name.

request_type Request type name.

reply_type Reply type name.

request_topic_name Request topic name.

reply_topic_name Reply topic name.

qos `uxrQoS_t` struct describing QoS.:mode: Determines the creation entity mode. The Creation Policy Table describes the entities' creation behaviour according to the `UXR_REUSE` and `UXR_REPLACE` flags (see *Creation Policy Table*).

Create entities common profile

These functions are enabled when either `PROFILE_CREATE_ENTITIES_XML` or `PROFILE_CREATE_ENTITIES_REF` are activated as CMake arguments. The declaration of these functions can be found in `uxr/client/profile/session/common_create_entities.h`.

```
uint16_t uxr_buffer_delete_entity(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId object_id);
```

Removes an entity. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

object_id The identifier of the object which is deleted.

Read access

The *Read Access* is used by the *Client* to handle the read operation on the *Agent*. The declaration of these functions can be found in `uxr/client/profile/session/read_access.h`.

```
uint16_t uxr_buffer_request_data(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId datareader_id,
                                 uxrStreamId data_stream_id, const␣
↪uxrDeliveryControl* const control);
```

This function requests a *datareader* previously created on the *Agent* to perform a read operation that fetches data from the middleware. The returned value is an identifier of the request. All received topics have the same request identifier. The topics are received on the callback topic through the `run_session` function. If there is no error with the request data, a status callback with the value `UXR_STATUS_OK` is generated along with the topics retrieval. If there is an error, a status error is sent by the *Agent*. The message is written into the stream buffer. To send the message, it is necessary to call either the `uxr_flash_output_streams` or the `uxr_run_session` function.

session Session structure previously initialized and created.

stream_id The output stream ID used to send messages to the *Agent*.

datareader_id The ID of the *datareader* that reads the topics from the middleware.

data_stream_id The input stream ID where the data is received.

control Optional information allowing the *Client* to configure the data delivery from the *Agent*. Details on the configurable parameters can be found in the *Read Access Delivery Control* of the *eProsima Micro XRCE-DDS Client* page. A `NULL` value is accepted, in which case only one topic is received.

```
uint16_t uxr_buffer_cancel_data(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId datareader_id);
```

This function requests a *datareader*, *requester* or *replier* previously created on the *Agent* to cancel the data received from the middleware. It does so by resetting the `delivery_control` parameters and the input stream ID used to receive the data. The returned value is an identifier of the request.

session Session structure previously initialized and created.

stream_id The output stream ID used to send messages to the *Agent*.

datareader_id The ID of the *datareader* that reads the topics from the middleware.

Write access profile

The *Write Access* is used by the *Client* to handle the write operation on the *Agent*. The declaration of these functions can be found in `uxr/client/profile/session/write_access.h`.

```
uint16_t uxr_prepare_output_stream(uxrSession* session, uxrStreamId stream_id,␣
↪uxrObjectId entity_id,
                                   ucdrBuffer* ub, uint32_t data_size);
```

This function requests a *datawriter*, *requester* or *replier* previously created on the *Agent* to perform a write operation into a specific output stream. It initializes a `ucdrBuffer` struct where a data of `data_size` size must be serialized. If there is sufficient space for writing `data_size` bytes into the stream, the returned value is the XRCE request ID, otherwise it is `0`. The topic is sent in the following `run_session` function. If `UCLIENT_PROFILE_MULTITHREAD` is enabled, user should unlock the stream lock after serializing the requested amount of data using `UXR_UNLOCK_STREAM_ID(session, stream_id);`

Note: All `data_size` bytes requested are sent to the *Agent* after a `run_session` call, no matter if the `ucdrBuffer` has been used or not.

---

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

entity_id The ID of the *datawriter*, *requester* or *replier* that writes data into the middleware.

ub The `ucdrBuffer` struct used to serialize the data. This struct points to the requested memory slot in the stream.

data_size The slot, in bytes, that is reserved in the stream.

```
bool uxr_buffer_request(uxrSession* session, uxrStreamId stream_id, uxrObjectId␣
↪requester_id, uint8_t* buffer, size_t len);
```

This function buffers a request into a specific output stream. The request is sent in the following `run_session` function call.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

requester_id The ID of the *requester* that forwards the request to the middleware.

buffer The raw buffer that contains the serialized request.

len The size of the serialized request.

```
bool uxr_buffer_reply(uxrSession* session, uxrStreamId stream_id, uxrObjectId replier_
↪id, uint8_t* buffer, size_t len);
```

This function buffers a reply into a specific output stream. The request is sent in the following `run_session` function call.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

replier_id The ID of the *replier* that writes the reply to the middleware.

buffer The raw buffer that contains the serialized reply.

len The size of the serialized reply.

```
bool uxr_buffer_topic(uxrSession* session, uxrStreamId stream_id, uxrObjectId␣
↪datawriter_id, uint8_t* buffer, size_t len);
```

This function buffers a topic into a specific output stream. The request is sent in the following `run_session` function call.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

datawriter_id The ID of the *datawriter* that writes the reply to the middleware.

buffer The raw buffer that contains the serialized topic.

len The size of the serialized topic.

```
uint16_t uxr_prepare_output_stream_fragmented(uxrSession* session, uxrStreamId stream_
↪id, uxrObjectId entity_id,
                                              struct ucdrBuffer* ub, size_t data_size,␣
↪uxrOnBuffersFull flush_callback);
```

This function requests a *datawriter*, *requester* or *replier* previously created on the *Agent* to allocate an output stream of `data_size` bytes for a write operation. This function initializes an `ucdrBuffer` struct where a topic of `data_size` size is serialized. If there is sufficient space for writing `data_size` bytes into the stream, the returned value is the XRCE request ID, otherwise it is `0`. The topic is sent in the following `run_session` function. If, during the serialization process, the buffer gets overfilled, the `flush_callback` function is called and the user has to run a session for flushing the stream. If `UCLIENT_PROFILE_MULTITHREAD` is enabled, user should unlock the stream lock after serializing the requested amount of data using `UXR_UNLOCK_STREAM_ID(session, stream_id);`

Note: This approach is not valid for best-effort streams.

session Session structure previously initialized and created.

stream_id The output stream ID where the messages are written.

entity_id The ID of the *datawriter*, *requester* or *replier* that writes data into the middleware.

ub The `ucdrBuffer` struct used to serialize the topic. This struct points to the requested memory slot in the stream.

data_size The slot, in bytes, that is reserved in the stream.

flush_callback Callback for flushing the output buffers.

The function signature for the `flush_callback` callback is:

```
typedef bool (*uxrOnBuffersFull) (struct uxrSession* session);
```

session Session structure related to the buffer to be flushed.

Discovery profile

The discovery profile allows discovering *Agents* in the network by UDP. The reachable *Agents* respond to the discovery call by sending information about themselves, as their IP and port. There are two modes: unicast and multicast. The discovery phase precedes the call to the `uxr_create_session` function, as it serves to determine the *Agent* to connect with. These functions are enabled when `PROFILE_DISCOVERY` is activated as a CMake argument. The declaration of these functions can be found in `uxr/client/profile/discovery/discovery.h`.

Note: This feature is only available on Linux.

```
bool uxr_discovery_agents_default(uint32_t attempts, int period, uxrOnAgentFound on_
↪agent_func, void* args);
```

This function looks for *Agents* in the network using UDP/IP multicast with address "239.255.0.2" and port 7400 (which is the default used by the *Agent*).

attempts The times a discovery message is sent across the network.

period The wait time between two successive discovery calls.

on_agent_func The callback function that is called when an *Agent* is discovered. It returns a boolean value. A `true` means that the discovery routine has finished. A `false` implies that the discovery routine must continue searching *Agents*.

args User arguments passed to the callback function.

The function signature for the `on_agent_func` callback is:

```
typedef bool (*uxrOnAgentFound) (const TransportLocator* locator, void* args);
```

locator Transport locator of a discovered agent.

args User pointer data.

```
bool uxr_discovery_agents(uint32_t attempts, int period, uxrOnAgentFound on_agent_
↪func, void* args,
                          const TransportLocator* agent_list, size_t agent_list_size);
```

This function looks for *Agents* in the network using UDP/IP unicast, using a list of unicast directions with the addresses and ports set by the user.

attempts The times a discovery message is sent across the network.

period The wait time between two successive discovery calls.

on_agent_func The callback function that is called when an *Agent* is discovered. It returns a boolean value. A `true` means that the discovery routine has finished. A `false` implies that the discovery routine must continue searching *Agents*.

args User arguments passed to the callback function.

agent_list The list of addresses used for discovering *Agents*.

agent_list_size The size of the `agent_list`.

The function signature for the `on_time_func` callback is the same as above.

Topic serialization

Functions to serialize and deserialize topics. These functions are generated automatically by the *eProsima Micro XRCE-DDS Gen* utility fed with an IDL file with a topic `TOPICTYPE`. The declaration of these functions can be found in the generated file `TOPICTYPE.h`.

```
bool TOPICTYPE_serialize_topic(struct ucdrBuffer* writer, const TOPICTYPE* topic);
```

This function serializes a topic into an `ucdrBuffer`. The returned value indicates if the serialization was successful.

writer The `ucdrBuffer` representing the buffer for the serialization.

topic Struct to serialize.

```
bool TOPICTYPE_deserialize_topic(struct ucdrBuffer* reader, TOPICTYPE* topic);
```

This function deserializes a topic from an `ucdrBuffer`. The returned value indicates if the deserialization was successful.

reader An `ucdrBuffer` representing the buffer for the deserialization.

topic Struct to deserialize.

```
uint32_t TOPICTYPE_size_of_topic(const TOPICTYPE* topic, uint32_t size);
```

This function counts the number of bytes that the topic needs in an *ucdrBuffer*.

topic Struct to count the size.

size Number of bytes already written into the *ucdrBuffer*. Typically, its value is *0* if the purpose is to use in `uxr_prepare_output_stream` function.

General utilities

Utility functions. The declaration of these functions can be found in `uxr/client/core/session/stream_id.h` and `uxr/client/core/session/object_id.h`.

```
uxrStreamId uxr_stream_id(uint8_t index, uxrStreamType type, uxrStreamDirection␣
↪direction);
```

This function creates a stream identifier. This function does not create a new stream, it only creates its identifier to be used in the *Client* API.

index Identifier of the stream. Its value corresponds to the creation order of the stream, different for each `type`.

type The type of the stream, it can be `UXR_BEST_EFFORT_STREAM` or `UXR_RELIABLE_STREAM`.

direction Represents the direction (input or output) of the stream. It can be `UXR_INPUT_STREAM` or `UXR_OUTPUT_STREAM`.

```
uxrStreamId uxr_stream_id_from_raw(uint8_t stream_id_raw, uxrStreamDirection␣
↪direction);
```

This function creates a stream identifier. This function does not create a new stream, it only creates its identifier to be used in the *Client* API.

stream_id_raw Identifier of the stream. It goes from **0** to **255**. **0** is for internal library use. **1** to **127** are for best effort. **128** to **255** are for reliable.

direction Represents the direction (input or output) of the stream. It can be `UXR_INPUT_STREAM` or `MT_OUTPUT_STREAM`.

```
uxrObjectId uxr_object_id(uint16_t id, uint8_t type);
```

This function creates an identifier to reference an entity.

id Identifier of the object, different for each `type`. There can be several objects with the same ID, provided they have different types.

type The type of the entity. It can be: `UXR_PARTICIPANT_ID`, `UXR_TOPIC_ID`, `UXR_PUBLISHER_ID`, `UXR_SUBSCRIBER_ID`, `UXR_DATAWRITER_ID`, `UXR_DATAREADER_ID`, `UXR_REQUESTER_ID`, or `UXR_REPLIER_ID`.

```
bool uxr_ping_agent_session(struct uxrSession* session, const int timeout_ms, const␣
↪uint8_t attempts);
```

This function pings a *Micro XRCE-DDS Agent* to check if it is already up and running.

This method does require an XRCE session to be established beforehand. Internally it spins the session until ping answer is received or it timeouts.

It returns `true` if a response was received from the *Agent*, `false` otherwise.

session A pointer to a properly initialized XRCE-DDS session, used to send the ping request.

timeout_ms The maximum time that the *Client* will wait to receive the answer (*pong*) message, before returning.

attempts Maximum amount of times that the *Client* will try to ping the *Agent* and receive a response back.

```
bool uxr_ping_agent(const uxrCommunication* comm, const int timeout_ms);
```

This function pings a *Micro XRCE-DDS Agent* to check if it is already up and running.

This method does not require an XRCE session to be established beforehand. It acts directly over the transport layer so if a session is running simultaneouly, data can be loss.

It returns `true` if a response was received from the *Agent*, `false` otherwise.

comm A pointer to a properly initialized XRCE-DDS communication structure, used to send the ping request.

timeout_ms The maximum time that the *Client* will wait to receive the answer (*pong*) message, before returning.

```
bool uxr_ping_agent_attempts(const uxrCommunication* comm, const int timeout_ms,␣
↪const uint8_t attempts);
```

This function provides the same functionality as the method described in `uxr_ping_agent`, but allows to specify the number of ping attempts before returning a value.

comm A pointer to a properly initialized XRCE-DDS communication structure, used to send the ping request.

timeout_ms The maximum time **per attempt** that the *Client* will wait to receive the answer (*pong*) message, before returning.

attempts Maximum amount of times that the *Client* will try to ping the *Agent* and receive a response back.

Transport

These functions are platform-dependent. The declaration of these functions can be found in the `uxr/client/profile/transport/` folder. The common init transport functions follow the nomenclature below.

```
bool uxr_init_udp_transport(uxrUDPTransport* transport, uxrIpProtocol ip_protocol,␣
↪const char* ip, const char* port);
```

This function initializes a UDP connection.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

ip_protocol IPv4 or IPv6.

ip *Agent* IP.

port *Agent* port.

```
bool uxr_init_tcp_transport(uxrTCPTransport* transport, uxrIpProtocol ip_protocol,␣
↪const char* ip, const char* port);
```

This function initializes a TCP connection. In the case of TCP, the behaviour of best-effort streams is similar to that of reliable streams in UDP.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

ip_protocol IPv4 or IPv6.

ip *Agent* IP.

port *Agent* port.

```
bool uxr_init_serial_transport(uxrSerialTransport* transport, const int fd, uint8_t␣
↪remote_addr, uint8_t local_addr);
```

This function initializes a Serial connection using a file descriptor.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

fd File descriptor of the serial connection. Usually, the `fd` comes from the `open` OS function.

remote_addr Identifier of the *Agent* in the connection. By default, the *Agent* identifier in a seria connection is **0**.

local_addr Identifier of the *Client* in the serial connection.

```
bool uxr_init_can_transport(uxrCANTransport* transport, const char* dev, uint32_t can_
↪id);
```

This function initializes a CAN FD connection using a network interface.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

dev Interface name of the CAN FD bus.

can_id Can identifier of this Client.

Note: The used interface must support CAN FD frames with a maximum payload of 64 bytes. The can identifier will be used on the CAN frames and should be unique for each client.

```
bool uxr_init_custom_transport(uxrCustomTransport* transport, void * args);
```

This function initializes a Custom connection using user-defined arguments.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection. `args` is accesible as `transport->args`.

```
bool uxr_close_PROTOCOL_transport(PROTOCOLTransport* transport);
```

This function closes a transport previously opened. `PROTOCOL` can be `udp`, `tcp`, `serial`, `can` or `custom`.

transport The structure used for managing the transport that must be closed.

```
void uxr_set_custom_transport_callbacks(uxrCustomTransport* transport, bool framing,
↪open_custom_func open,
                        close_custom_func close, write_custom_func write, read_
↪custom_func read);
```

This function assigns the callback for custom transport.

transport The uninitialized structure used for managing the transport. This structure must be accessible during the connection.

framing Enables or disables Stream Framing Protocol for a custom transport.

open Callback for opening a custom transport.

close Callback for closing a custom transport.

write Callback for writing to a custom transport.

read Callback for reading from a custom transport.

The function signatures for the above callbacks are:

```
typedef bool (*open_custom_func) (struct uxrCustomTransport* transport);
```

transport Custom transport structure. Has the `args` passed through `bool uxr_init_custom_transport(uxrCustomTransport* transport, void * args);`.

```
typedef bool (*close_custom_func) (struct uxrCustomTransport* transport);
```

transport Custom transport structure. Has the `args` passed through `bool uxr_init_custom_transport(uxrCustomTransport* transport, void * args);`.

```
typedef size_t (*write_custom_func) (struct uxrCustomTransport* transport, const
↪uint8_t* buffer, size_t length, uint8_t* error_code);
```

transport Custom transport structure. Has the `args` passed through `bool uxr_init_custom_transport(uxrCustomTransport* transport, void * args);`.

buffer Buffer to be sent.

length Length of buffer.

error_code Error code that should be set in case the write process experiences some error.

This function should return the number of successfully sent bytes.

```
typedef size_t (*read_custom_func) (struct uxrCustomTransport* transport, uint8_t*␣
↪buffer, size_t length, int timeout, uint8_t* error_code);
```

transport Custom transport structure. Has the `args` passed through `bool uxr_init_custom_transport(uxrCustomTransport* transport, void * args);`.

buffer Buffer to write.

length Maximum length of buffer.

timeout Maximum timeout of the read operation.

error_code Error code that should be set in case the write process experiences some error.

This function should return the number of successfully received bytes.

5.9.2 Agent API

The *Micro XRCE-DDS Agent* is developed using a fully compliant C++11 API. This allowed to focus its development on modularity and usability, while keeping it simple for the final user.

Keeping up with this philosophy, the API provided to the user attempts to be as much intuitive as possible, while allowing to configure all the different aspects related to the *Agent*'s behaviour.

That being said, most user will find out that, with te provided *MicroXRCEAgent* standalone application, it is more than enough to launch an *Agent* and start the communication process with *Micro XRCE-DDS Client* applications. This is possible thanks to the intuitive built-in *Agent CLI* and its multiple configuration parameters.

Alternatively, users can access the underneath *Agent* implementation and fine-tune all of its parameters, options, and behaviour in their final application. This is specially useful when dealing with a *Custom Transport* implementation.

The *Agent* API section is organized as follows:

- *eprosima::uxr::AgentInstance*
- *eprosima::uxr::Agent*

eprosima::uxr::AgentInstance

The CLI manager, along with all the built-in *Agents* available for the supported transports (UDP, TCP, Serial, CAN) are encapsulated in the `eprosima::uxr::AgentInstance` class.

```
AgentInstance& getInstance();
```

This function gets a reference to the singleton `AgentInstance` wrapper class, which allows launching a *Micro XRCE-DDS Agent* with user-given parameters.

```
bool create(int argc, char** argv);
```

This function creates a UDP/TCP/Serial/CAN *Micro XRCE-DDS Agent*, based on the given arguments. The created *Agent* will start automatically on success.

Returns `true` if the arguments were valid and an *Agent* was successfully created, `false` otherwise.

argc Number of arguments provided by the user via the CLI. This is usually inherited from the `main` loop.

argv List of arguments to be parsed by the CLI engine.

```
void run();
```

This function blocks until the previously created *Agent* is ended by the stop function, a user's interrupt or a process error.

```
void stop();
```

Stops a previously created *Agent*, blocking until the stop process is completed. Note that this will trigger a call on the `transport::fini` method.

To restart a stopped agent, the `create` method should be used.

```
template <typename ... Args>
void add_middleware_callback(const Middleware::Kind& middleware_kind, const␣
↪middleware::CallbackKind& callback_kind, std::function<void (Args ...)>&& callback_
↪function);
```

This function sets a user-defined callback function for a specific create/delete middleware entity operation.

middleware_kind Enum defining all the supported middlewares (see *Middleware Abstraction Layer*).

callback_kind Enum holding all the possible create/delete operations:

```
enum class CallbackKind : uint8_t
{
    CREATE_PARTICIPANT,
    CREATE_DATAWRITER,
    CREATE_DATAREADER,
    CREATE_REQUESTER,
    CREATE_REPLIER,
    DELETE_PARTICIPANT,
    DELETE_DATAWRITER,
    DELETE_DATAREADER,
    DELETE_REQUESTER,
    DELETE_REPLIER
};
```

callback_function Callback to be defined by the user. It must follow a certain signature, depending on the middleware used.

---

eprosima::uxr::Agent

However, it is also possible for users to create and instantiate their own *Agent*, for example, to implement a *Custom transport*. Also, in some scenarios, it could be useful to have all the necessary `ProxyClients` and their associated DDS entities created by the *Agent* even before *Clients* are started, so that *Clients* applications can avoid the process of creating the session and the DDS entities, and can focus on the communication.

This would allow a Micro XRCE-DDS Client application to be as tiny as it can be in terms of memory consumption.

The following API is provided to fulfill these requirements:

```
bool create_client(uint32_t key, uint8_t session, uint16_t mtu, Middleware::Kind␣
↪middleware_kind, OpResult& op_result);
```

This function allows to create a `ProxyClient` entity, which can act on behalf of an external *Client* to request the creation/deletion of DDS entities.

It returns `true` if the creation was successful, `false` otherwise.

key The `ProxyClient`'s identifier.

session The session ID to which the created `ProxyClient` is attached to.

mtu The *Maximum Transmission Unit* size.

middleware_kind The middleware used by the `ProxyClient`, to be chosen among the ones presented in the *Middleware Abstraction Layer*.

op_result The result status of this operation.

```
bool delete_client(uint32_t key, OpResult& op_result);
```

This function deletes a given `ProxyClient` from the client proxy database, given its ID.

Returns `true` if the operation was completed successfully, `false` otherwise (for example, if the provided ID was not registered to any `ProxyClient`).

key The identifier of the `ProxyClient` to be removed.

op_result The result status of the operation.

```
bool create_participant_by_xml(uint32_t client_key, uint16_t participant_id, int16_t␣
↪domain_id, const char* xml, uint8_t flag, OpResult& op_result);
```

This function creates a DDS participant for a given *Client*, given its self-contained description in an XML file.

The participant will act as an entry point for the rest of the DDS entities to be created.

It returns `true` if the creation was successful, `false` otherwise.

client_key The identifier of the `ProxyClient` to which the resulting participant will be attached to.

participant_id The identifier of the participant to be created.

domain_id The DDS domain ID associated to the participant.

xml The XML describing the participant properties.

flag It determines the creation mode of the new participant (see *Creation Mode: Client* and *Creation Mode: Agent*).

op_result The result status of this operation.

```
bool create_participant_by_ref(uint32_t client_key, uint16_t participant_id, int16_t␣
↪domain_id, const char* ref, uint8_t flag, OpResult& op_result);
```

This function creates a DDS participant for a given *Client*, given a reference to its description hosted in a certain XML descriptor file.

This reference file must have been previously loaded to the *Agent*.

The participant will act as an entry point for the rest of the DDS entities to be created.

Returns `true` if the creation was successful, `false` otherwise.

client_key The identifier of the `ProxyClient` to which the resulting participant will be attached to.

participant_id The identifier of the participant to be created.

domain_id The DDS domain ID associated to the participant.

ref The reference tag which will retrieve the participant description from the file where the references are defined, previously loaded to the *Agent*.

flag It determines the creation mode of the new participant (see *Creation Mode: Client* and *Creation Mode: Agent*).

op_result The result status of this operation.

```
bool delete_participant(uint32_t client_key, uint16_t participant_id, OpResult& op_
↪result);
```

This function removes a DDS participant from a certain client proxy. Returns `true` if the participant was deleted, `false` otherwise.

client_key The identifier of the `ProxyClient` from which the participant must be deleted.

participant_id The ID of the participant to be deleted.

op_result The result status of the operation.

```
bool create_<entity>_by_xml(uint32_t client_key, uint16_t <entity>_id, uint16_t
↪<associated_entity>_id, const char* xml, uint8_t flag, OpResult& op_result);
```

This function creates a certain DDS entity attached to an existing `ProxyClient`, given its client key.

An XML must be provided, containing the DDS description of the entity to be created.

There are as many methods available as existing DDS entities, replacing the parameters <*entity*> and <*associated_entity*> as follows:

Agent's API available DDS entities and their associated entities

<entity>	<associated_entity>
topic	participant
publisher	participant
subscriber	participant
datawriter	publisher
datareader	subscriber
requester	participant
replier	participant

This operation returns `true` if the entity is successfully created and linked to its associated entity (which must previously exist in the given `ProxyClient`), `false` otherwise.

client_key The identifier of the `ProxyClient` to which the resulting entity will be attached to.

<entity>_id The ID of the DDS entity to be created.

<associated_entity>_id The identifier of the DDS entity to which this one will be assocciated.

xml The XML describing the entity properties.

flag It determines the creation mode of the new entity (see see *Creation Mode: Client* and *Creation Mode: Agent*).

op_result The result status of this operation.

```
bool create_<entity>_by_ref(uint32_t client_key, uint16_t <entity>_id, uint16_t
↪<associated_entity>_id, const char* ref, uint8_t flag, OpResult& op_result);
```

This function creates a certain DDS entity attached to an existing `ProxyClient`, given its client key.

The description of the entity to be created is hosted in a certain file where all the required references are defined, and must be tagged with the same tag name, provided as the `ref` parameter to this method.

There are as many methods available as existing DDS entities, replacing the parameters *<entity>* and *<associated_entity>* as shown above in the previous method description (see *Agent's API available DDS entities and their associated entities*).

This operation returns `true` if the entity is successfully created and linked to its associated entity (which must previously exist in the given `ProxyClient`), `false` otherwise.

client_key The identifier of the `ProxyClient` to which the resulting entity will be attached to.

<entity>_id The ID of the DDS entity to be created.

<associated_entity>_id The identifier of the DDS entity to which this one will be assocciated.

ref The reference tag which will retrieve the DDS entity description from the file hosting the referenced entities definitions.

flag It determines the creation mode of the new entity (see see *Creation Mode: Client* and *Creation Mode: Agent*).

op_result The result status of this operation.

```
bool delete_<entity>(uint32_t client_key, uint16_t <entity>_id, OpResult& op_result);
```

This function deletes a certain entity from a `ProxyClient`. Its associated entities will also be deleted, if applicable.

There exist as many method signatures of this type in the agent's API as available entities. See the *Agent's API available DDS entities and their associated entities* table for further information.

It returns `true` if the entity is correctly removed, `false` otherwise.

client_key The identifier of the `ProxyClient` from which the entity must be deleted.

<entity>_id The ID of the DDS entity to be deleted.

op_result The result status of the operation.

```
bool load_config_file(const std::string& file_path);
```

This function loads a configuration file that provides the tagged XML definitions of the desired XRCE entities that can be created using the reference creation mode (see see *Creation Mode: Client* and *Creation Mode: Agent*).

The used syntax must match the one defined for FastDDS XML profile syntax, where the *profile name* attributes represent the reference names.

It returns `true` if the file was correctly loaded, `false` otherwise.

file_path Relative path to the file containing the DDS entities description in XML format, tagged accordingly to be referenced by the API.

Note: This function needs to be called when implementing a Custom transport in the case creation of entities by reference is used. This function must be called before *eprosima::uxr::Server* `start` method.

```
void reset();
```

This function deletes all the `ProxyClient` instances and their associated DDS entities.

```
void set_verbose_level(uint8_t verbose_level);
```

This function sets the verbose level of the logger, from **0** (logger is off) to **6** (critical, error, warning, info, debug, and trace messages are displayed).

Intermediate tracing levels display information up to the position in the aforementioned list; for example, level **4** shows critical, error, warning and info messages.

Note: This function must be called before *eprosima::uxr::Server* `start` method.

---

verbose_level The level to be set.

```
template <typename ... Args>
void add_middleware_callback(const Middleware::Kind& middleware_kind, const␣
↪middleware::CallbackKind& callback_kind, std::function<void (Args ...)>&& callback_
↪function);
```

This function exposes the same functionality as the one described in `add_middleware_callback`, for *eprosima::uxr::AgentInstance*.

Note: This function must be called before *eprosima::uxr::Server* `start` method.

eprosima::uxr::Server

This class inherits from *eprosima::uxr::Agent* and it is the base class used for implementing any of the built-in *Agent* servers that are available by default in the standalone executable that is generated when the library is compiled and installed (see *Installation Manual*), and that can be launched and used by means of the built-in *Agent CLI*.

Also, when creating a *Custom Agent*, which inherits directly from *eprosima::uxr::Server* users will need to call the *start()* method after configuring the *Agent*, if applicable (namely, by using `load_config_file`, `set_verbose_level` or `add_middleware_callback` methods). An example on how to do this can be found here.

```
bool start();
```

Launches the threads involved in the *Agent* server communication, namely, receiver and sender thread for getting/dispatching messages; processing thread, to process the messages; and a heartbeat and an error handler thread. After calling this method, the communication between the *Agent* and the *Clients* can effectively start.

This method returns `true` if the server has been correctly started, or `false` if some error happened during startup.

```
bool stop();
```

Stops a previously launched *eprosima::uxr::Server* and all of its associated threads.

This method returns `true` if the stopping process was successful, or `false` otherwise.

5.10 eProsima Micro XRCE-DDS Gen

eProsima Micro XRCE-DDS Gen is a Java application used to generate source code for the *eProsima Micro XRCE-DDS* software.

This tool can generate from a given IDL specification file, the C struct associated with the Topic, as well as the serialization and deserialization methods. Also, it can generate a sample demo that works with the proposed topic.

To find out how to install this package, refer to the *Installing the Micro XRCE-DDS Gen tool* section.

As an example of the potential of this tool, the following shows the source code generated from the ShapeDemo IDL file.

```
// ShapeType.idl

struct ShapeType {
    @key string color;
    long x;
    long y;
    long shapesize;
};
```

If we will perform the following command:

```
$ microxrceddsgen ShapeType.idl
```

it will generate the following header file and its corresponding source:

```
/*!
 * @file ShapeType.h
 * This header file contains the declaration of the described types in the IDL file.
 *
 * This file was generated by the tool gen.
 */

#ifndef _ShapeType_H_
#define _ShapeType_H_

#include <stdint.h>
#include <stdbool.h>

/*!
 * @brief This struct represents the structure ShapeType defined by the user in the␣
↪IDL file.
 * @ingroup SHAPETYPE
 */
typedef struct ShapeType
{
    char color[255];
    int32_t x;
    int32_t y;
    int32_t shapesize;

} ShapeType;

struct ucdrBuffer;

bool ShapeType_serialize_topic(struct ucdrBuffer* writer, const ShapeType* topic);
bool ShapeType_deserialize_topic(struct ucdrBuffer* reader, ShapeType* topic);
uint32_t ShapeType_size_of_topic(const ShapeType* topic, uint32_t size);

#endif // _ShapeType_H_
```

eProsima Micro XRCE-DDS Gen is also able to generate both *publisher* and *subscriber* source code examples, related with the topic specified in the IDL file, by adding the flag `-example`:

```
$ microxrceddsgen -example <file.idl>
```

The *Client* library must be compiled with the `WRITE_ACCESS_PROFILE` option for the *publisher*, to use these examples and the `READ_ACCESS_PROFILE` option for the *subscriber*.

Note: At present, *eProsima Micro XRCE-DDS Gen* only supports Structs composed of integer, string, array and sequence types, even though it is planned to enhance the capabilities of the *eProsima Micro XRCE-DDS Gen* tool in a near future.

5.11 Memory optimization

This section explains how memory is managed by the *eProsima Micro XRCE-DDS* library and how it can be configured and customized by the user. For more information regarding the internal handling of the memory in *eProsima Micro XRCE-DDS*, refer to the detailed analysis and memory profiling here.

5.11.1 Executable code size

To tune the executable code size, the library can be compiled enabling or disabling several profiles. To add or remove profiles from the library, enable or disable them as CMake arguments. More information can be found at: *eProsima Micro XRCE-DDS Client*.

Important: When compiling with *gcc*, it is highly recommended to compile it with the linker flag: `-Wl,--gc-sections`. It will remove the code that the app doesn't use from the final executable.

5.11.2 Runtime size

The *Client* is **dynamic** and **static** memory free: the whole memory footprint only depends on how the **stack** grows during the execution. Several values can be modified to control the stack growth:

Stream buffers

Streams number It's possible to define a maximum of *127* best-effort streams and of *128* reliable streams. However, for most purposes, only one stream - either best effort or reliable - is needed.

History of a reliable stream The history is used for recovering lost messages and for speeding up the communication. For output streams, a bigger history will allow writing and sending more messages without having to wait for confirmation. However, if the history of an output stream is full (no messages confirmed by the *Agent* yet), no more messages can be stored in the stream. For input streams, the history is used for recovering lost messages faster while reducing the bandwidth. If the connection is highly reliable and saving memory is a priority, a reduced history can be used.

Stream size In case of reliable communication, this size is equal to `MAX_MESSAGE_SIZE * HISTORY`. In case of a best-effort stream, the size simply equals `MAX_MESSAGE_SIZE`, as no history is made available in this case. The `MAX_MESSAGE_SIZE` represents the maximum message size that can be sent without fragmenting the message, and it must be less or equal than the *MTU* chosen for the selected transport.

MTU A different *MTU* can be chosen for each transport available. The *MTU* value can be defined as a CMake argument, and fixes the `MAX_MESSAGE_SIZE` that can be sent or received. The transport uses the *MTU* value to create an internal buffer.

5.12 Transport

This section shows how the transport layer is implemented in both *eProsima Micro XRCE-DDS Agent* and *eProsima Micro XRCE-DDS Client*. Furthermore, this section describes how to add a Custom transport in *eProsima Micro XRCE-DDS*. It is organized as follows:

- *Introduction*
- *Agent Transport Architecture*
- *Client Transport Architecture*
- *Stream Framing Protocol*
- *Custom Transport*

5.12.1 Introduction

In contrast to other IoT middlewares such as MQTT and CoaP, which work over a particular transport protocol, the DDS-XRCE protocol is designed to support multiple transport protocols natively. This feature of DDS-XRCE is enhanced by *eProsima Micro XRCE-DDS* in two ways. On the one hand, the logic of both the *Agent* and the *Client* is completely separated from the transport protocol underneath through a set of interfaces, which will be explained in the following sections.

On the other hand, taking advantage of the transport interface flexibility, the Client comes with a framing protocol implemented that enables using the DDS-XRCE wire protocol over stream-oriented transports. This feature allows using *eProsima Micro XRCE-DDS* over two kinds of transports layers:

- **Packet-oriented transports**: communication protocols that allow sending whole packets.
- **Stream-oriented transports**: communication protocols that follow a stream logic.

5.12.2 Agent Transport Architecture

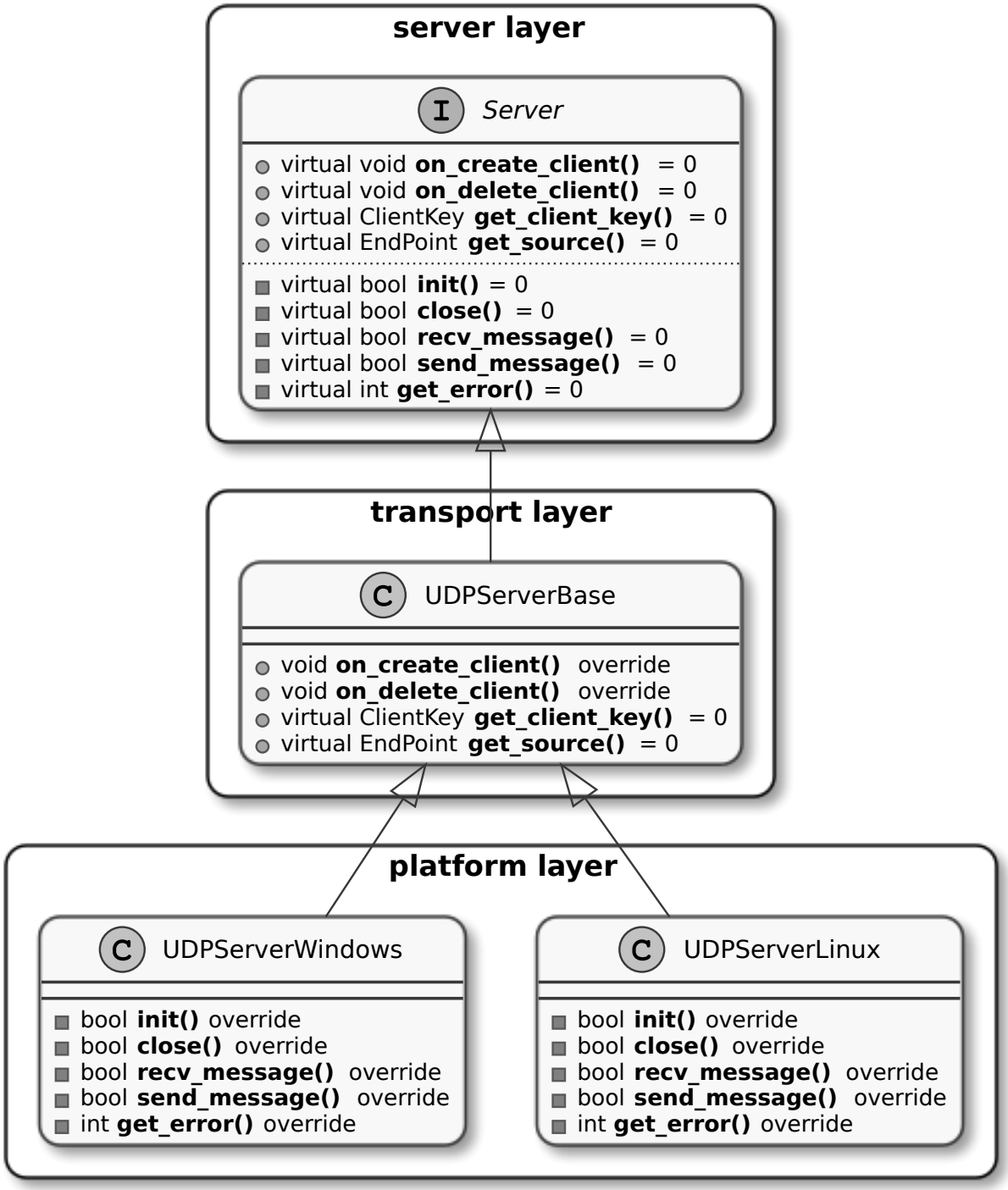
The *Agent* transport architecture is composed by 3 different layers:

- **Server Layer**: is an interface from which each transport-specific server inherits. This interface implements four different threads:
  - *Sender thread*: in charge of sending the messages to the *Clients*.
  - *Receiver thread*: in charge of receiving the messages from the *Clients*.
  - *Processing thread*: in charge of processing the messages received from the *Clients*.
  - *Heartbeat thread* in charge of handling reliability with the *Clients*.
- **Transport Layer**: is a transport-specific class which manages the sessions established between the *Agent* and the *Clients*. This class inherits from the *Server* interface.

- **Platform Layer**: is a platform-specific class which implements the sending and receiving functions for a given transport in a given platform. It should be noted that it is the only class that has platform dependencies.

UDP Server Example

As an example, this subsection describes how the UDP server is implemented in *eProsima Micro XRCE-DDS Agent*. The figure below shows the *Agent* transport architecture for the UDP servers.



At the top of this architecture, there is a `Server` interface (Server Layer). This `Server` interface has the following pure virtual functions:

```
/* Transport Layer */
virtual void on_create_client(EndPoint* source, const dds::xrce::ClientKey& client_
↪key) = 0;
virtual void on_delete_client(EndPoint* source) = 0;
```

(continues on next page)

(continued from previous page)

```
virtual const dds::xrce::ClientKey get_client_key(EndPoint* source) = 0;
virtual std::unique_ptr<EndPoint> get_source(const dds::xrce::ClientKey& client_key)␣
↪= 0;

/* Platform Layer */
virtual bool init() = 0;
virtual bool close() = 0;
virtual bool recv_message(InputPacket& input_packet, int timeout) = 0;
virtual bool send_message(OutputPacket output_packet) = 0;
virtual int get_error() = 0;
```

The first four virtual functions are transport specific (Transport Layer). These functions are overridden by the `UDPServerBase` class, which is in charge of managing the sessions between *Clients* and the *Agent*.

On the other hand, the last five virtual functions are platform specific (Platform Layer). These functions are override by the `UDPServerLinux` and `UDPServerWindows` for Linux and Windows systems, respectively.

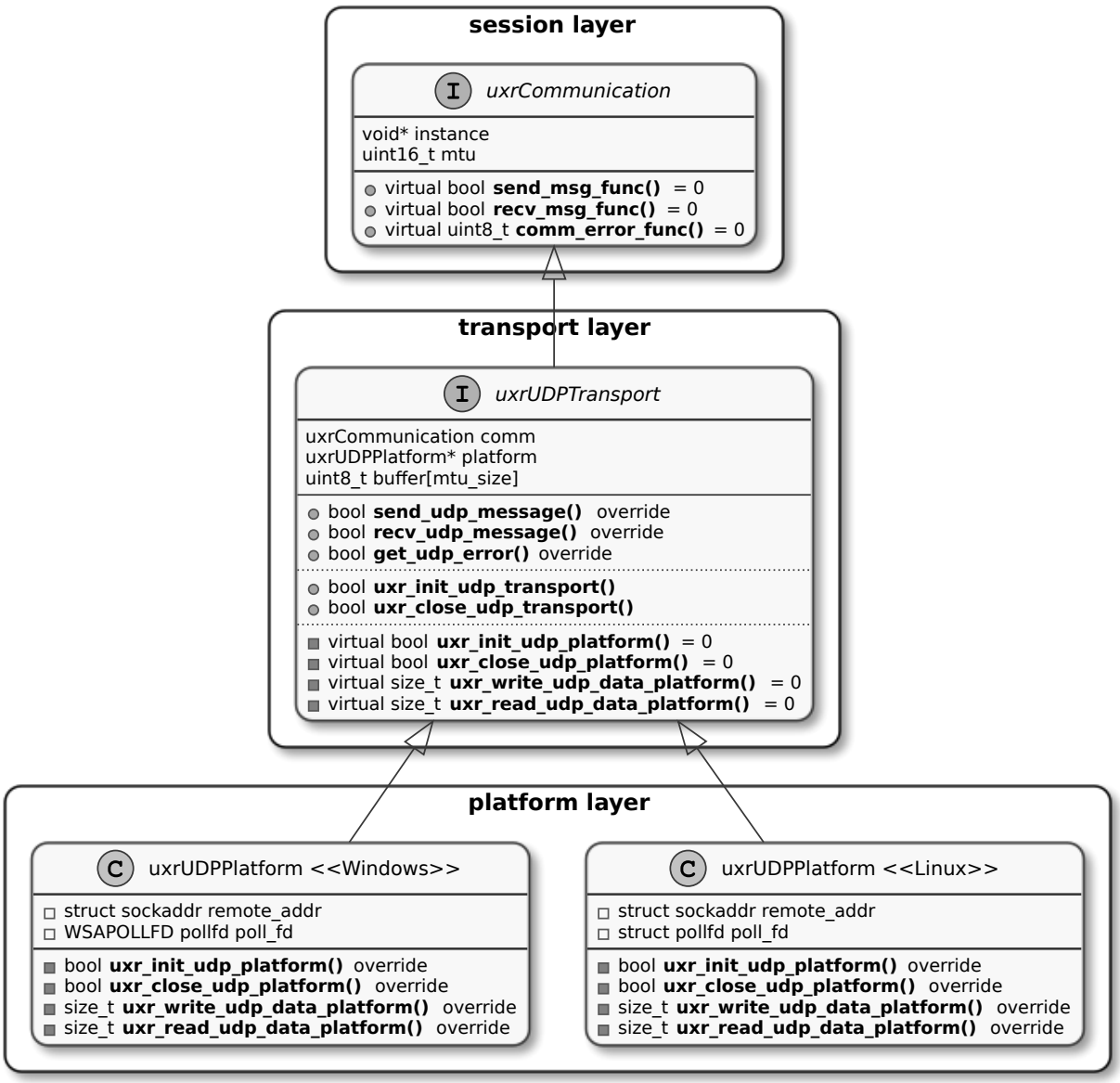
5.12.3 Client Transport Architecture

The *Client* transport architecture is analogous to the *Agent* architecture. There are also three different layers, but instead of the Server Layer, there is a Session Layer.

- **Session Layer**: implements the XRCE protocol logic, and it only knows about sending and receiving messages.
- **Transport Layer**: implements the sending and receiving **message functions** for each transport protocol, calling to the Platform Layer functions. This layer only knows about sending and receiving messages through a given transport protocol.
- **Platform Layer**: implements the sending and receiving **data functions** for each platform. This layer only knows about sending and receiving raw data through a given transport in a given platform.

UDP Transport Example

As an example, this subsection describes how the UDP transport is implemented in *eProsima Micro XRCE-DDS Client*. The figure below shows the *Client* transport architecture for UDP transport.



Similar to the *Agent* architecture, there is also an interface, `uxrCommunication`, whose function pointers are used from the Session Layer. That is, each time a `run_session` is called, the Session Layer calls to `send_msg_func` and `recv_msg_func` without worrying about the transport protocol or the platform in use. This struct has the following function pointers:

```
bool send_msg_func(void* instance, const uint8_t* buf, size_t len);
bool recv_msg_func(void* instance, uint8_t** buf, size_t* len, int timeout);
uint8_t comm_error_func(void);
```

These functions are implemented by the `uxrUDPTransport`, which is in charge of two main tasks:

1. Provide an implementation for the communication interface functions. For example, in the case of the UDP protocol, these functions are the following:

```
bool send_udp_msg(void* instance, const uint8_t* buf, size_t len);
bool recv_udp_msg(void* instance, uint8_t** buf, size_t* len, int timeout);
```

(continues on next page)

(continued from previous page)

```
uint8_t get_udp_error(void);
```

2. Offer to the user the initialization and close functions related to the transport protocol. For example, in the case of the UDP protocol, these functions are the following:

```
bool uxr_init_udp_transport(uxrUDPTransport* transport, const char* ip, uint8_t port);
bool uxr_close_udp_transport(uxrUDPTransport* transport);
```

For each platform, there is an implementation of these functions defined in the Transport Layer interface. For example, in the case of Linux under UDP transport protocol, the `uxrUDPPlatform` implements the following functions:

```
bool uxr_init_udp_platform(uxrUDPPlatform* platform, const char* ip, uint16_t port);
bool uxr_close_udp_platform(uxrUDPPlatform* platform);
size_t uxr_write_udp_data_platform(uxrUDPPlatform* platform, const uint8_t* buf, size_
↪t len, uint8_t* errcode);
size_t uxr_read_udp_data_platform(uxrUDPPlatform* platform, uint8_t* buf, size_t len,␣
↪int timeout, uint8_t* errcode);
```

5.12.4 Stream Framing Protocol

eProsima Micro XRCE-DDS has a **Stream Framing Protocol** with the following features:

- **HDLC Framing**: each frame begins with a `begin_frame` octet `(0x7E)`, and the rest of the frame undergoes byte stuffing, using the `space` octet `(0x7D)` followed by the original octet exclusive-or with `0x20`. For example, if the frame contains the octet *0x7E*, it is encoded as *0x7D, 0x5E*; and the same for the octet *0x7D* which is encoded as *0x7D, 0x5D*.
- **CRC Calculation**: frames end with the CRC-16 for detecting frame corruption. The CRC-16 is computed using the polynomial `x^16 + x^12 + x^5 + 1` after the frame stuffing for each octet of the frame and including the `begin_frame`, as it is described in the RFC 1662 (see sec. C.2).
- **Routing header**: the Stream Framing Protocol provides `source` and `remote` addresses in the framing, which can be used to implement a routing protocol.

All the previous features are addressed using the following frame format:

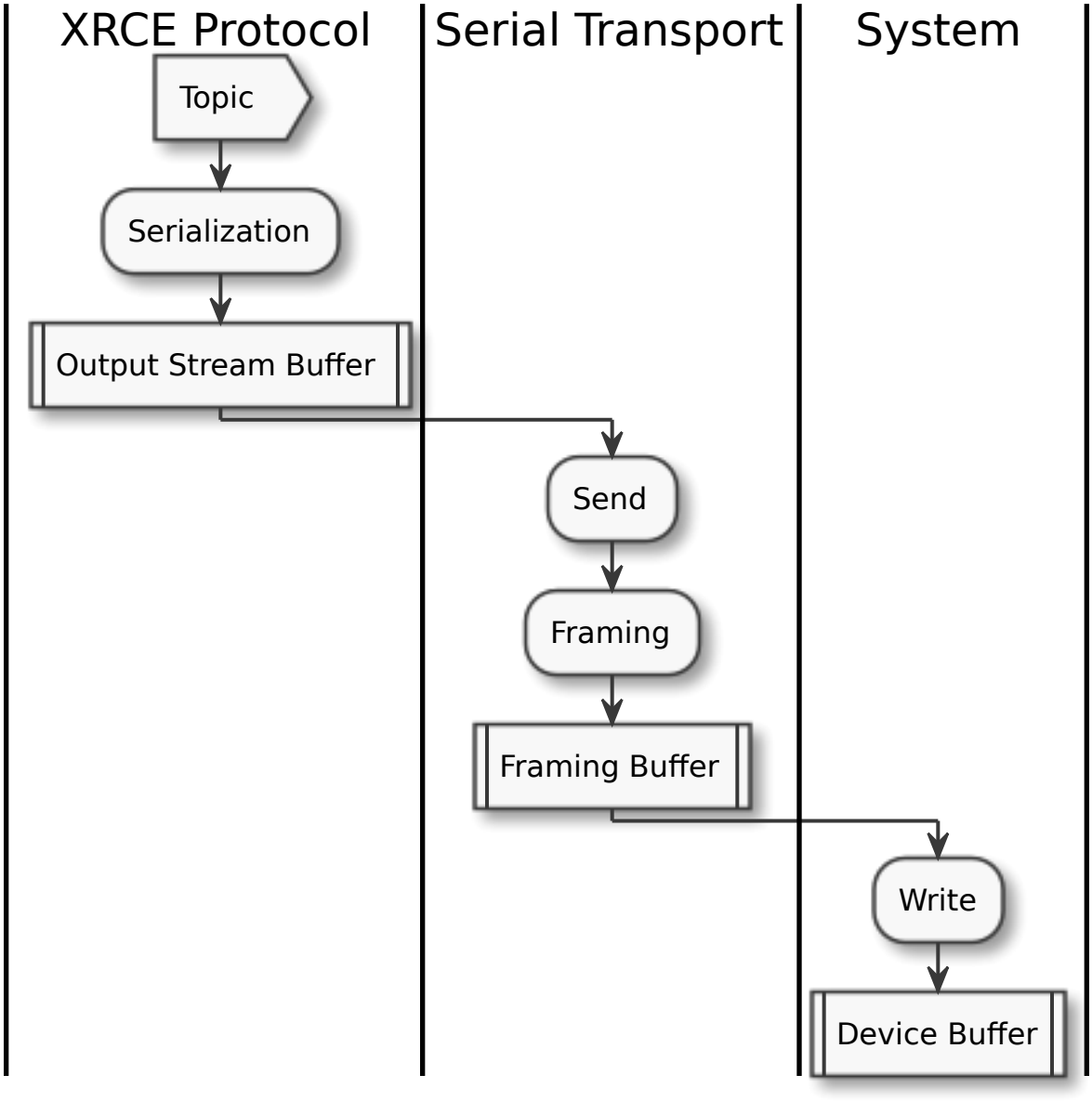
```
0        8        16       24                40                  X                 X+16
+--------+--------+--------+--------+--------+--------//--------+--------+--------+
|  FLAG  |  SADD  |  RADD  |        LEN       |       PAYLOAD     |        CRC       |
+--------+--------+--------+--------+--------+--------//--------+--------+--------+
```

- `FLAG`: is a `begin_frame` octet for frame initialization.
- `SADD`: is the address of the device which sent the message, that is, the `source` address.
- `RADD`: is the address of the device which should receive the message, that is, the `remote` address.
- `LEN`: is the length of the **payload without framing**. It is encoded as a 2-bytes array in little-endian.
- `PAYLOAD`: is the payload of the message.
- `CRC`: is the CRC of the message **after the stuffing**.

Data Sending

The figure below shows the workflow of the data sending. This workflow could be divided into the following steps:

1. A publisher application calls the *Client* library to send a given topic.
2. The *Client* library serializes the topic inside an XRCE message using *Micro CDR*. As a result, the XRCE message with the topic is stored in an **Output Stream Buffer**.
3. The *Client* library calls the Stream Framing Protocol to send the serialized message.
4. The Stream Transport frames the message, that is, it adds the header, the payload, and CRC of the frame, taking into account the stuffing. This step takes place in an auxiliary buffer called **Framing Buffer**.
5. Each time the Framing Buffer is full, the data is flushed into the **Device Buffer**, calling the writing system function.



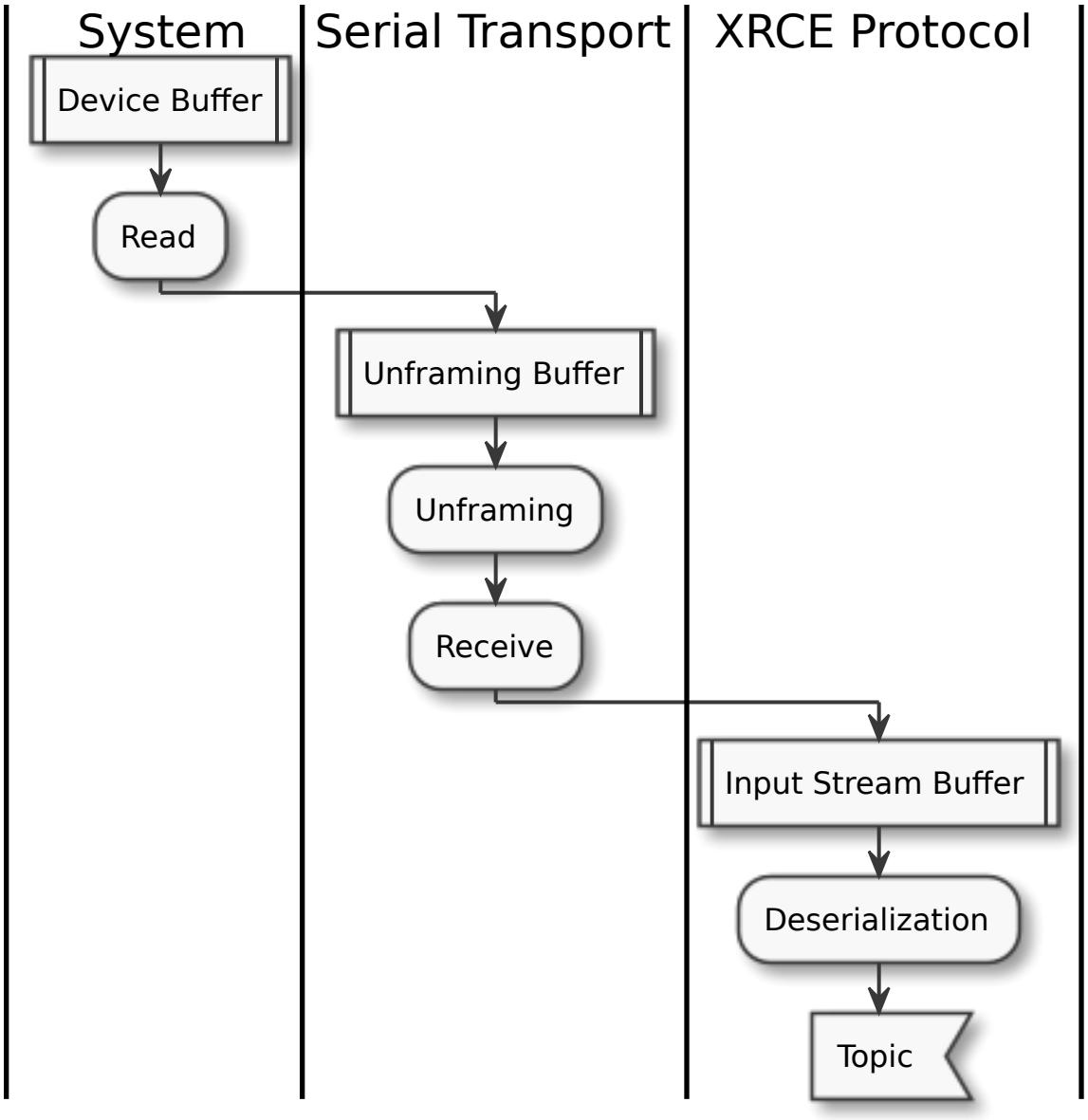
This approach has some advantages which should be pointed out:

1. The HDLC framing and the CRC control provide **integrity** and **security** to the Stream Framing.
2. The framing technique allows to **reduce memory usage**. The reason is that the Framing Buffer size (42 bytes) bounds the Device Buffer size.
3. The framing technique also allows sending **large data** over stream-oriented transports. The reason is that the message size is not bounded by the Device Buffer size, since the message is fragmented and has undergone byte stuffing during the framing stage.

Data Receiving

The workflow of the data receiving is analogous to the data sending workflow:

1. A subscriber application calls the *Client* library to receive a given topic.
2. The *Client* library calls the Stream Framing Protocol to receive the stream message.
3. The Stream Framing Protocol reads data from the **Device Buffer** and unframes the raw data received from the Device Buffer in the **Unframing Buffer**.
4. Once the Unframing Buffer is full, the Stream Framing Protocol appends the fragment into the **Input Stream Buffer**. This operation is repeated until a complete message is received.
5. The *Client* library deserializes the topic from the Input Stream Buffer to the user topic struct.



It should point out that this approach has the same advantages that the sending one.

Shapes Topic Example

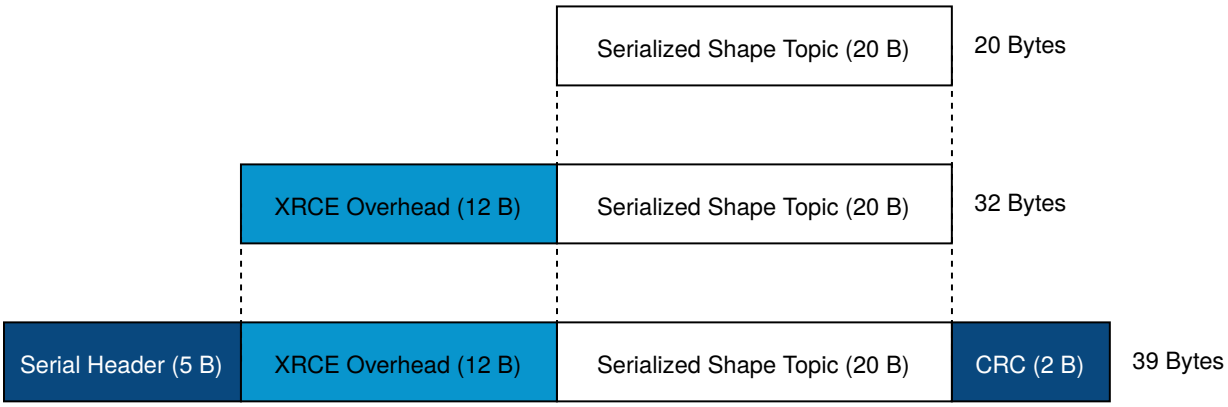
This subsection shows how a **Shapes Topic**, defined by the IDL below, is packed into the Serial Transport.

```
typedef struct ShapeType
{
    char color[128];
    int32_t x;
    int32_t y;
    int32_t shapesize;
} ShapeType;

ShapeType topic = {"red", 11, 11, 89};
```

In Serial Transport, the topic's packaging could be divided into two steps:

1. The Session Layer adds the XRCE header and subheader. It adds an overhead of 12 bytes to the topic.
2. The Serial Transport adds the serial header, CRC and stuffing the payload. In the best case, it adds an overhead of 7 bytes to the topic.



The figure above shows the overhead added by Serial Transport. In the best case, it is **only 19 bytes**, but it should be noted that, in this example, the message stuffing has been neglected.

5.12.5 Custom Transport

eProsima Micro XRCE-DDS provides a user API that allows interfacing with the lowest level transport layer at runtime, which enables users to implement their own transports in both the *Client* and *Agent* libraries. Thanks to this, the Micro XRCE-DDS wire protocol can be transmitted over virtually any protocol, network or communication mechanism. In order to do so, two general communication modes are provided:

- **Stream-oriented mode**: the communication mechanism implemented does not have the concept of packet. HDLC framing (*Stream Framing Protocol*) will be used.
- **Packet-oriented mode**: the communication mechanism implemented is able to send a whole packet that includes an XRCE message.

These two modes can be selected by activating and deactivating the `framing` parameter in both the *Client* and the *Agent* functions.

The relevant API can be found in the *Transport* section of the *Client API*.

Micro XRCE-DDS Client

In order to enable the *eProsima Micro XRCE-DDS Client* profile for custom transports, the CMake argument `UCLIENT_PROFILE_CUSTOM_TRANSPORT=<bool>` must be set to true. By doing so, the user will enable the functionality for setting the transport-related callbacks explained in the *Transport* section of the *Client API*.

An example on how to set these external transport callbacks in the *Client* API is:

```
uxrCustomTransport transport;
uxr_set_custom_transport_callbacks(
    &transport,
    true, // Framing enabled here. Using Stream-oriented mode.
    my_custom_transport_open,
    my_custom_transport_close,
    my_custom_transport_write,
```

(continues on next page)

(continued from previous page)

```
    my_custom_transport_read);

struct custom_args {
    ...
}

struct custom_args args;

if(!uxr_init_custom_transport(&transport, (void *) &args))
{
    printf("Error at create transport.\n");
    return 1;
}
```

It is important to notice that in `uxr_init_custom_transport` a pointer to custom arguments is set. This reference will be copied to the `uxrCustomTransport` and will be available to every callbacks call.

In general, four functions need to be implemented. The behavior of these functions is sightly different, depending on the selected mode:

Open function

```
bool my_custom_transport_open(uxrCustomTransport* transport)
{
    ...
}
```

This function should open and init the custom transport. It returns a boolean indicating if the opening was successful.

`transport->args` have the arguments passed through `uxr_init_custom_transport`.

Close function

```
bool my_custom_transport_close(uxrCustomTransport* transport)
{
    ...
}
```

This function should close the custom transport. It returns a boolean indicating if closing was successful.

`transport->args` have the arguments passed through `uxr_init_custom_transport`.

Write function

```
size_t my_custom_transport_write(
        uxrCustomTransport* transport,
        const uint8_t* buffer,
        size_t length,
        uint8_t* errcode)
{
    ...
}
```

This function should write data to the custom transport. It returns the number of Bytes written.

`transport->args` have the arguments passed through `uxr_init_custom_transport`.

- **Stream-oriented mode:** The function can send up to `length` Bytes from `buffer`.

- **Packet-oriented mode:** The function should send `length` Bytes from `buffer`. If less than `length` Bytes are written `errcode` can be set.

Read function

```
size_t my_custom_transport_read(
        uxrCustomTransport* transport,
        uint8_t* buffer,
        size_t length,
        int timeout,
        uint8_t* errcode)
{
    ...
}
```

This function should read data to the custom transport. It returns the number of Bytes read

`transport->args` have the arguments passed through `uxr_init_custom_transport`.

- **Stream-oriented mode:** The function should retrieve up to `length` Bytes from transport and write them into `buffer` in `timeout` milliseconds.
- **Packet-oriented mode:** The function should retrieve `length` Bytes from transport and write them into `buffer` in `timeout` milliseconds. If less than `length` Bytes are read `errcode` can be set.

Micro XRCE-DDS Agent

The *eProsima Micro XRCE-DDS Agent* profile for custom transports is enabled by default.

An example on how to set the external transport callbacks in the Micro XRCE-DDS Agent API is:

```
eprosima::uxr::Middleware::Kind mw_kind(eprosima::uxr::Middleware::Kind::FASTDDS);
eprosima::uxr::CustomEndPoint custom_endpoint;

// Add transport endpoing parameters
custom_endpoint.add_member<uint32_t>("param1");
custom_endpoint.add_member<uint16_t>("param2");
custom_endpoint.add_member<std::string>("param3");

eprosima::uxr::CustomAgent custom_agent(
    "my_custom_transport",
    &custom_endpoint,
    mw_kind,
    true, // Framing enabled here. Using Stream-oriented mode.
    my_custom_transport_open,
    my_custom_transport_close,
    my_custom_transport_write
    my_custom_transport_read);

custom_agent.start();
```

CustomEndPoint

The `custom_endpoint` is an object of type `eprosima::uxr::CustomEndPoint` and it is in charge of handling the endpoint parameters. The *Agent*, unlike the *Client*, can receive messages from multiple *Clients* so it must be able to differentiate between them. Therefore, the `eprosima::uxr::CustomEndPoint` should be provided with information about the origin of the message in the read callback, and with information about the destination of the message in the write callback.

In general, the members of a `eprosima::uxr::CustomEndPoint` object can be unsigned integers and strings.

`CustomEndPoint` defines three methods:

Add member

```
bool eprosima::uxr::CustomEndPoint::add_member<*KIND*>(const std::string& member_
↪name);
```

Allows to dynamically add a new member to the endpoint definition.

Returns `true` if the member was correctly added, `false` if something went wrong (for example, if the member already exists).

KIND To be chosen from: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `uint128_t` or `std::string`.

member_name The tag used to identify the endpoint member.

Set member value

```
void eprosima::uxr::CustomEndPoint::set_member_value(const std::string& member_
↪name, const *KIND* & value);
```

Sets the specific value (numeric or string) for a certain member, which must previously exist in the `CustomEndPoint`.

member_name The member whose value is going to be modified.

value The value to be set, of *KIND*: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `uint128_t` or `std::string`.

Get member

```
const *KIND* & eprosima::uxr::CustomEndPoint::get_member(const std::string&␣
↪member_name);
```

Gets the current value of the member registered with the given parameter. The retrieved value might be an `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `uint128_t` or `std::string`.

member_name The *CustomEndPoint* member name whose current value is requested.

CustomAgent user-defined methods

As in the *Client* API, four functions should be implemented. The behaviour of these functions is sightly different depending on the selected mode:

Open function

```
eprosima::uxr::CustomAgent::InitFunction my_custom_transport_open = [&]() -> bool
{
    ...
}
```

This function should open and init the custom transport. It returns a boolean indicating if the opening was successful.

Close function

```
eprosima::uxr::CustomAgent::FiniFunction my_custom_transport_close = [&]() -> bool
{
    ...
}
```

This function should close the custom transport. It returns a boolean indicating if the closing was successful.

Write function

```
eprosima::uxr::CustomAgent::SendMsgFunction my_custom_transport_write = [&](
    const eprosima::uxr::CustomEndPoint* destination_endpoint,
    uint8_t* buffer,
    size_t length,
    eprosima::uxr::TransportRc& transport_rc) -> ssize_t
{
    ...
}
```

This function should write data to the custom transport. It must use the `destination_endpoint` members to set the data destination. It returns the number of Bytes written. It should set `transport_rc` indicating the result of the operation.

- **Stream-oriented mode:** The function can send up to `length` Bytes from `buffer`.
- **Packet-oriented mode:** The function should send `length` Bytes from `buffer`. If less than `length` Bytes are written, `transport_rc` can be set.

Read function

```
eprosima::uxr::CustomAgent::RecvMsgFunction my_custom_transport_read = [&](
        eprosima::uxr::CustomEndPoint* source_endpoint,
        uint8_t* buffer,
        size_t length,
        int timeout,
        eprosima::uxr::TransportRc& transport_rc) -> ssize_t
{
    ...
}
```

This function should read data to the custom transport. It must fill `source_endpoint` members with data source. It returns the number of Bytes read. It should set `transport_rc` indicating the result of the operation.

- **Stream-oriented mode:** The function should retrieve up to `length` Bytes from transport and write them into `buffer` in `timeout` milliseconds.
- **Packet-oriented mode:** The function should retrieve `length` Bytes from transport and write them into `buffer` in `timeout` milliseconds. If less than `length` Bytes are read transport_rc can be set.

5.13 P2P Communication

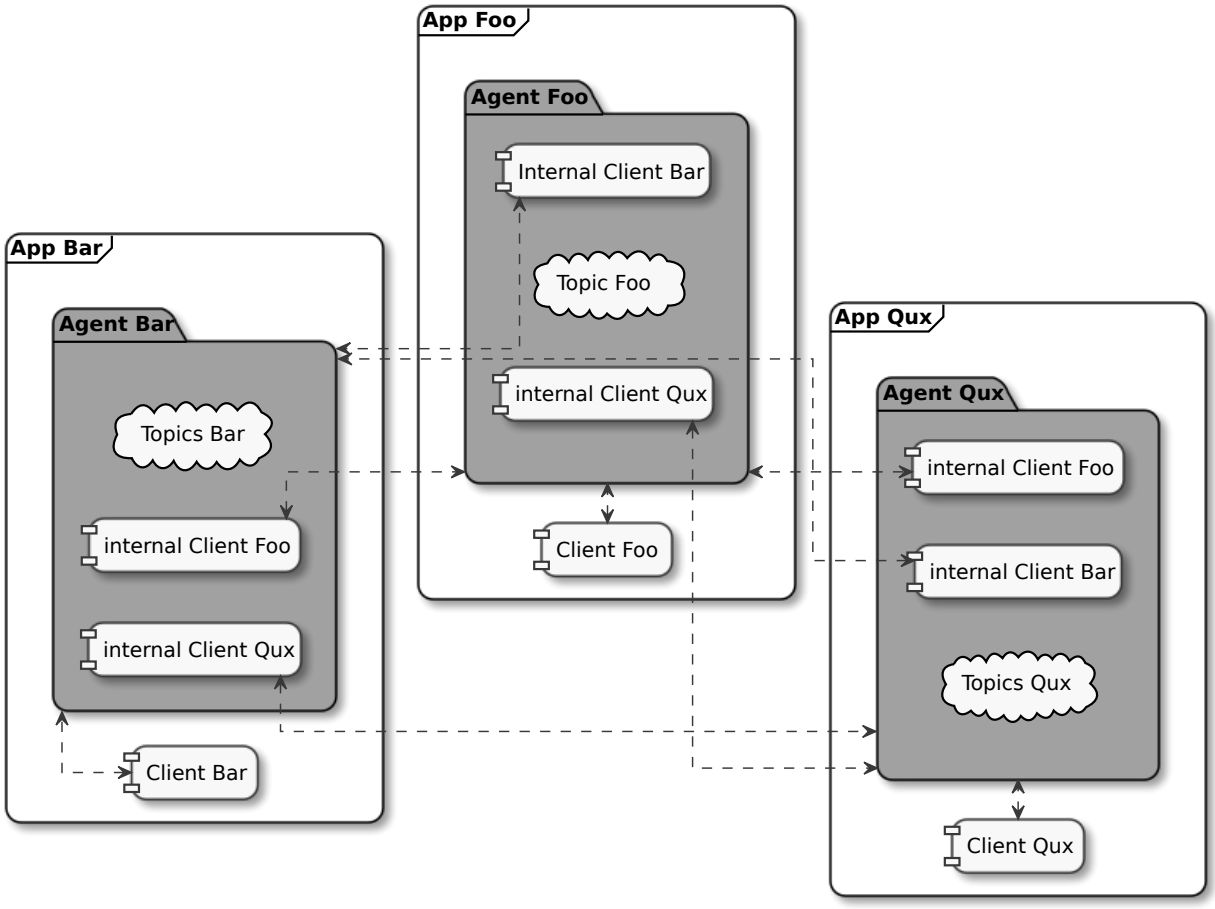
This section illustrates the peer-to-peer communication mode offered by *eProsima Micro XRCE-DDS*. It is organized as follows:

- *Introduction*
- *Publish/Subscribe P2P Example*

5.13.1 Introduction

The peer-to-peer (P2P) mode allows direct communication between applications without DDS, where by *application* is to be understood the combination of an *Agent* and one or more *Clients*.

In this communication mode, the *Agent* uses the *CedMiddleware*. The *Agent* is in charge of discovering other *External Agents* and create an *Internal Client* for each one of them. Each *Internal Client* connects to an *External Agent* and subscribes to the set of Topics managed by its own *Agent*. Thus, a cloud of interconnected *Agents* is created.



Some consideration shall be taken into account in order to use the P2P communication:

1. Only the `create_<entity>_by_ref()` functions shall be used.
2. The Topic's reference represents the name of the Topic.
3. The DataWriter's and DataReader's references need to match the Topic's reference.
4. Publishers and Subscribers have no role.
5. Agents use the CedMiddleware and the Discovery mechanism.

5.13.2 Publish/Subscribe P2P Example

This kind of behaviour can be probed by putting in communication a Publisher P2P application with a Subscriber P2P application.

Agent with CedMiddleware

First of all, install the *Agent* as explained in the *Installing the Agent standalone* section. On Linux, this would be:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
$ cd Micro-XRCE-DDS-Agent
$ mkdir build && cd build
$ cmake ..
$ make
$ sudo make install
```

After having installed the *Agent* system-wide, launch it with the `ced` option activated:

```
$ ./MicroXRCEAgent udp4 -p <port> -m ced -d
```

Client P2P publisher app

Let's now install the *Client* locally, and with the `-DUCLIENT_BUILD_EXAMPLES=ON` flag enabled, so as to activate the compilation of the examples. On Linux, this implies running the following:

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Client.git
$ cd Micro-XRCE-DDS-Client
$ mkdir build && cd build
$ cmake .. -DUCLIENT_BUILD_EXAMPLES=ON
$ make
```

At this point, it's possible to launch the `PublishHelloWorldClientP2P` executable located in the folder `Micro-XRCE-DDS-Client/build/examples/PublishHelloWorldP2P`, which'll make the *Client* publish in the DDS World the `HelloWorld` topic (take a look at the IDL defining this topic in the file `Micro-XRCE-DDS-Client/examples/PublishHelloWorldP2P/HelloWorld.idl`).

```
$ examples/PublishHelloWorld2P2/PublishHelloWorldClientP2P 127.0.0.1 2019
```

The source code of the `PublishHelloWorldClientP2P` can be found in `Micro-XRCE-DDS-Client/examples/PublishHelloWorldP2P/main.c`.

Client P2P subscriber app

After having executed the publisher app, we can launch the `SubscribeHelloWorldClientP2P` excutable, which is located in the folder `Micro-XRCE-DDS-Client/build/examples/SubscribeHelloWorldP2P`, which'll make this *Client* subscribe to the same `HelloWorld` topic from the DDS World.

```
$ examples/SubscriberHelloWorldP2P/SubscribeHelloWorldClientP2P 127.0.0.1 2019
```

The source code of the `SubscribeHelloWorldClientP2P` can be found in `Micro-XRCE-DDS-Client/examples/SubscribeHelloWorldP2P/main.c`.

At this point, the subscriber will receive the topics that are being sent by the publisher.

5.14 Time synchronization

eProsima Micro XRCE-DDS offers a synchronization mechanism based on the NTP protocol. It allows synchronizing *Client* with *Agents*, something very useful when working in embedded environments that do not provide any time synchronization mechanism. The use of this feature is quite simple. The *Client* library provides the function `uxr_sync_session` which is all that is needed. This function involves an exchange of messages between *Client* and *Agent* that allows the *Client* compute its time offset using the NTP protocol. Apart from it, the *Client* library also provides a callback that allows users to implement their time synchronization protocol.

Find the code for a *Client* application making use of this time synchronization mechanism in the TimeSync example app.

Another useful example shows how to use the time-synchronization callback. In particular, this example implements the NTP protocol with average computation to increase the time-offset's accuracy. Find the code for this example in the TimeSync with callback example app.

5.15 eProsima Docker Image

eProsima provides the eProsima XRCE-DDS Suite Docker image for those who want a quick demonstration of XRCE-DDS running on an Ubuntu platform.

This Docker image was built for Ubuntu 20.04 (Focal Fossa).

To run this container you need Docker installed; from a terminal run:

```
$ sudo apt install docker.io
```

5.15.1 XRCE-DDS Suite

This Docker image contains the complete XRCE-DDS suite, which includes:

- *eProsima XRCE-DDS libraries and examples*: XRCE-DDS libraries bundled with several examples that showcase a variety of capabilities of eProsima's Fast DDS implementation.
- *eProsima XRCE-DDS Agent*: eProsima XRCE-DDS Agent is a ready to use implementation of DDS-XRCE Agent and it is available for using along with provided examples.

To load this image into your Docker repository, run:

```
$ docker load -i xrcedds-suite:<XRCE-DDS-Version>.tar
```

You can run this Docker container as follows:

```
$ docker run -it xrcedds-suite:<XRCE-DDS-Version>
```

From the resulting Bash Shell you can run each feature.

Client Hello World Example

Included in this Docker container is a set of binary examples that showcase some of the functionalities of the XRCE-DDS libraries.

This is a minimal example that will perform a Publisher/Subscriber match and start sending samples. This example is not constrained to the current instance. It's possible to run several instances of this container to check the communication between them by running the following from each container:

```
$ docker run -it xrcedds-suite:<XRCE-DDS-Version> helloworld_pub
```

or

```
$ docker run -it xrcedds-suite:<XRCE-DDS-Version> helloworld_sub
```

XRCE-DDS Agent

This command creates an instance of the eProsima XRCE-DDS Agent, required for communicating XRCE-DDS Client. In order to use it run:

```
$ docker run -it xrcedds-suite:<XRCE-DDS-Version> xrce_agent [ARGUMENTS]
```

More information about the eProsima XRCE-DDS Agent CLI can be found *here*

5.16 Version 2.2.1

Agent 2.2.1 | Client 2.2.1 | Micro-CDR 2.0.0

- **Agent 2.2.1:**
  - **This release includes the following bugfixes:**
    - Fix exception on Heartbeat filter (#314)
    - Fix default QoS in Requester and Replier (#313)
  - **This release includes the following minor changes:**
    - Bump Fast DDS to v2.8 and Fast CDR to v1.0.24 (#315)
- **Client 2.2.1:**
  - **This release includes the following bugfixes:**
    - Check setsockopt return (#325)
- **Micro-CDR 2.0.0:**
  - This release is not modified

5.17 Version 2.2.0

Agent 2.2.0 | Client 2.2.0 | Micro-CDR 2.0.0

- **Agent 2.2.0:**
  - **This release includes the following bugfixes:**
    - Fix select timeout format (#311)
    - Default services to preallocated with realloc (#310)
  - **This release includes the following minor changes:**
    - Implement hard liveliness check (#308)
- **Client 2.2.0:**
  - **This release includes the following bugfixes:**
    - SuperBuild.cmake: pass C, CXX and LINKER flags too (#315)
    - Add a nopoll version of the POSIX TCP transport profile (#318)
    - Fix wait_session_status listen timeout (#322)
  - **This release includes the following minor changes:**
    - Implement hard liveliness check (#316)
- **Micro-CDR 2.0.0:**
  - This release is not modified

5.18 Version 2.1.1

Agent 2.1.1 | Client 2.1.1 | Micro-CDR 2.0.0

- **Agent 2.1.1:**
  - **This release includes the following bugfixes:**
    - Fix write destination id (#292)
    - Add sub entities destruction on FastDDS entities (#295)
    - Add reuse socket to TCP agent (#301)
    - Fix linux compile (#297)
  - **This release includes the following minor changes:**
    - Add CAN payload len on first frame byte (#293)
    - Add CAN transport flag to cmake / Upgrade splog version (#296)
    - Add Twitter and Readthedocs shields (backport #298) (#299)
    - Add use system spdlog flag (#303)
    - Implement GET_STATUS implementation result (#304)
- **Client 2.1.1:**
  - **This release includes the following bugfixes:**
    - Fix fragment capacity overflow (#296)

        * Fix fragmentation header alignment (#300)
        * Fix run session timeouts (#299)
        * Fix code scanning alert (#302)
        * Fix exit run session condition (#305)
        * Fix multithread interlock (#303)
        * Reset stream on created session (#304)
        * Fix subscriber example (#309)
        * Fix Req Res example (#314)
    - **This release includes the following minor changes:**
        * RTEMS Serial Transport support (#297)
        * Add payload lenght on CAN messages (#298)
        * Add Twitter and Readthedocs shields (#307)
        * Implement GET_STATUS implementation result (#312)
- **Micro-CDR 2.0.0:**
    - This release is not modified

5.19 Version 2.1.0

Agent 2.1.0 | Client 2.1.0 | Micro-CDR 2.0.0

- **Agent 2.1.0:**
    - **This release includes the following bugfixes:**
        * Style corrections (#238)
        * Fix packaging test (#241)
        * Fix serial error detection (#251)
        * Server: Add wait for error_handle (#252)
        * Fix use FastDDS profiles (#260)
        * Fix session key log (#265)
        * Fix custom transport bug (#259)
        * Add missing define if logger is disabled (#267)
        * Fix warning when CED disabled (#272)
        * FramingIO optimizations (#278)
        * Fix stream type on entities creation/destruction (#284)
    - **This release includes the following minor changes:**
        * Add wait for a serial port connection (#246)
        * Set runtime check for discovery and p2p protocols (#254)
        * Add flag for using system Fast-CDR (#255, #256)

        * Add LOG_INFO traces when entities are created (#257)
        * Add stop functionality (#268)
    - **This release includes the following major changes:**
        * Client shared memory support (#236)
        * Binary entity creation mode (#239, #245, #248, #250, #273)
        * Off-standard 64 kB write limit tweak (#249)
        * Multiserial agent functionality (#253, #262)
        * Build agent with Android NDK (#280, #282, #283)
        * Incoming heartbeats filter (#277)
        * Support for CAN/FD (#285)
        * Updated Fast-DDS to v2.4.1 and Fast-CDR to v1.0.22
- **Client 2.1.0:**
    - **This release includes the following bugfixes:**
        * Minor fixes in FreeRTOS (#236, #239, #270)
        * Style corrections (#222, #223, #231, #237, #247, #248)
        * Fix missing declarations of inet_to family for POSIX_NOPOLL (#272)
        * Modified heartbeat calculations (#251)
        * FramingIO performance improvements (#259, #267)
        * Fix conditional compilation Shapes Demo Windows (#262)
        * Fix uxr_run_session_until_all_status (#279)
        * Add check to stream type on fragmented output (#293)
    - **This release includes the following minor changes:**
        * Doxygen updates (#226, #229, #292)
        * XRCE-DDS sessions runs at least once when timeout is 0 ms (#212)
        * Add argument to continuous fragment mode callback (#260)
        * Add flag to force micro-CDR build (#264)
        * Support building for Android with NDK. (#269)
        * Allow for pinging once and and return (#282)
        * Allow wait session with no timeout (#280)
    - **This release includes the following major changes:**
        * Binary entity creation mode (#224, #232, #241, #246, #266)
        * Multithread support and shared memory transport (#216, #234, #240, #243, #245, #238, #263, #274, #289, #290, #291, #294)
        * Off-standard 64 kB write limit tweak (#244)
        * Support for CAN/FD (#278, #284)
        * Support for RTEMS RTOS (#283, #287)

- **Micro-CDR 2.0.0:**
  - **This release includes the following bugfixes:**
    - Fixed buffer handling in fragmentation for compatibility with FastDDS (#69).
  - **This release includes the following minor changes:**
    - Only add -wsign-conversion if supported (#68)
    - Avoid enabling CXX language (#67)
    - Fix memcmp in tests (#66)
    - Only add -wdouble-promotion if supported (#65)
    - Update ABI Stability section (#64)

5.20 Version 2.0.0

Agent 2.0.0 | Client 2.0.0 | Micro-CDR 1.2.1

This version includes the following changes in both Agent and Client:

- **Agent 2.0.0:**
  - **Add**
    - Micro XRCE-DDS Agent Snap package
    - Middleware callbacks API
    - Client to Agent ping feature without a session
    - Custom transports API
  - **Fix / Modify**
    - Simplified CLI and removed dependency with CLI11 library.
    - Optional disable of executable build.
    - CLI help console output.
    - Removed platform handling in user API.
- **Client 2.0.0:**
  - **Add**
    - POSIX transport with based on timeout instad of polling.
    - Client to Agent ping feature without a session
    - Continuos fragment mode
    - FreeRTOS+TCP transport support
    - Zephyr RTOS time functions support
    - Custom transports API
    - DDS-XRCE best effort examples
    - `uxr_run_session_until_data` functionality
    - `uxr_create_session_retries` functionality

        * `uxr_buffer_topic` functionality
    - **Fix / Modify**
        * Update session creating timing to linear approach
        * Modified `uxr_prepare_output_stream` API return code
        * Removed `client.config` file in favor of CMake arguments.
        * Removed platform handling in user API.
        * Bugfix #156 request/reply lenght management.
        * Bugfix #167 reliable fragment slots management.
        * Bugfix #175 reliable fragment size management.
        * Bugfix #176 discovery message deserialization.
- **Micro-CDR 1.2.1:**
    - **Fix / Modify**
        * Bugfix #53 fix in ucdr_reset_buffer function
        * Bugfix #54 fix alignment zero-length sequence bug
        * Bugfix #55 fix asymmetric fragmentation buffers

5.21 Version 1.3.0

Agent 1.4.0 | Client 1.2.3

This version includes the following changes in both Agent and Client:

- **Agent 1.4.0:**
    - **Add**
        * FastDDS middleware (compatible with ROS 2 Foxy).
    - **Fix**
        * TermiosAgent's baudrate setting.
- **Client 1.2.3:**
    - **Modify**
        * Examples installation.
    - **Fix**
        * Minor Windows visibility function fixes.

5.22 Previous Versions

5.22.1 Version 1.2.0

Agent 1.3.0 | Client 1.2.1

This version includes the following changes in both Agent and Client:

- **Agent 1.3.0**
  - **Add**
    - IPv6 support.
    - Requester/Replier support.
    - Packaging compatibility with colcon.
    - Isolated installation option.
    - Raspberry Pi support.
  - **Change**
    - Serial transport.
- **Client 1.2.1**
  - **Add**
    - IPv6 support.
    - Requester/Replier support.
    - Packaging compatibility with colcon.
    - Isolated installation option.

5.22.2 Version 1.1.0

Agent 1.1.0 | Client 1.1.1

This version includes the following changes in both Agent and Client:

- **Agent 1.1.0:**
  - **Add**
    - Message fragmentation.
    - P2P communication.
    - API.
    - Time synchronization.
    - Windows discovery support.
    - New unitary tests.
    - API documentation.
    - Logger.
    - Command Line Interface.
    - Centralized middleware.

        * Remove Asio dependency.
    - **Change**
        * CMake approach.
        * Server's thread pattern.
        * Fast RTPS version upgraded to 1.8.0.
    - **Fix**
        * Serial transport.
- **Client 1.1.1:**
    - **Add**
        * Message fragmentation.
        * Time synchronization.
        * Windows discovery support.
        * New unitary tests.
        * API documentation.
        * Raspberry Pi support.
    - **Change**
        * Memory usage improvement.
        * CMake approach.
        * Discovery API.
        * Examples usage.
    - **Fix**
        * Acknack reading.
        * User data bad alignment.

5.22.3 Version 1.0.3

Agent 1.0.3 | Client 1.0.2

This version includes the following changes in both Agent and Client:

- **Agent 1.0.3:**
    - Fast RTPS version upgraded to 1.7.2.
    - Baud rate support improvements.
    - Bugfixes.
- **Client 1.0.2:**
    - Uses new Fast RTPS 1.7.2 XML format.
    - Add Raspberry Pi toolchain.
    - Fix bugs.

5.22.4 Version 1.0.2

Agent 1.0.2 | Client 1.0.1

This version includes the following changes in the Agent:

- **Agent 1.0.2:**
  - Fast RTPS version upgraded to 1.7.0.
  - Added dockerfile.
  - Documentation fixes.

5.22.5 Version 1.0.1

Agent 1.0.1 | Client 1.0.1

This release includes the following changes in both Agent and Client:

- **Agent 1.0.1:**
  - Fixed Windows installation.
  - Fast CDR version upgraded.
  - Simplified CMake code.
  - Bug fixes.
- **Client 1.0.1:**
  - Fixed Windows configuration.
  - MicroCDR version upgraded.
  - Cleaned unused code.
  - Fixed documentation.
  - Bug fixes.

5.22.6 Version 1.0.0

This release includes the following features:

- Extended C topic code generation tool (strings, sequences, and n-dimensional arrays).
- Discovery profile.
- Native write access profile (without using *eProsima Micro XRCE-DDS Gen*)
- Creation and configuration by XML.
- Creation by reference.
- Added *REUSE* flag at entities creation.
- Added prefix to functions.
- Transport stack modification.
- More tests.
- Reorganized project.

- Bug fixes.
- API changes.

5.22.7 Version 1.0.0Beta2

This release includes the following features:

- Reliability.
- Stream concept (best-effort, reliable).
- Multiples streams of the same type.
- Configurable data delivery control.
- C Topic example code generation tool.