DDS Router Documentation

Release ..

eProsima

Mar 26, 2024

INTRODUCTION

1 Contacts and Commercial support **3**

2 Contributing to the documentation **5**

3 Structure of the documentation **7**

3.1 Overview . . . 7
3.2 Contacts and Commercial support . . . 8
3.3 Contributing to the documentation . . . 8
3.4 Structure of the documentation . . . 9
3.5 DDS Router on Windows . . . 9
3.6 DDS Router on Linux . . . 9
3.7 Docker image . . . 9
3.8 Project Overview . . . 10
3.9 Example of usage . . . 12
3.10 User Interface . . . 17
3.11 DDS Router Participant . . . 21
3.12 Configuration . . . 30
3.13 WAN Configuration . . . 50
3.14 YAML Validator . . . 54
3.15 Nomenclature . . . 55
3.16 Echo Example . . . 56
3.17 Change Domain Example . . . 59
3.18 ROS2 Discovery Server Example . . . 61
3.19 WAN Example . . . 63
3.20 Repeater Example . . . 66
3.21 XML Participant Configuration . . . 68
3.22 Forwarding Routes . . . 70
3.23 WAN communication over TCP . . . 75
3.24 ROS 2 and Kubernetes . . . 82
3.25 Repeater DDS Router . . . 89
3.26 Linux installation from sources . . . 90
3.27 Windows installation from sources . . . 96
3.28 CMake options . . . 103
3.29 Version v2.2.0 . . . 104
3.30 Previous Versions . . . 105
3.31 Glossary . . . 111

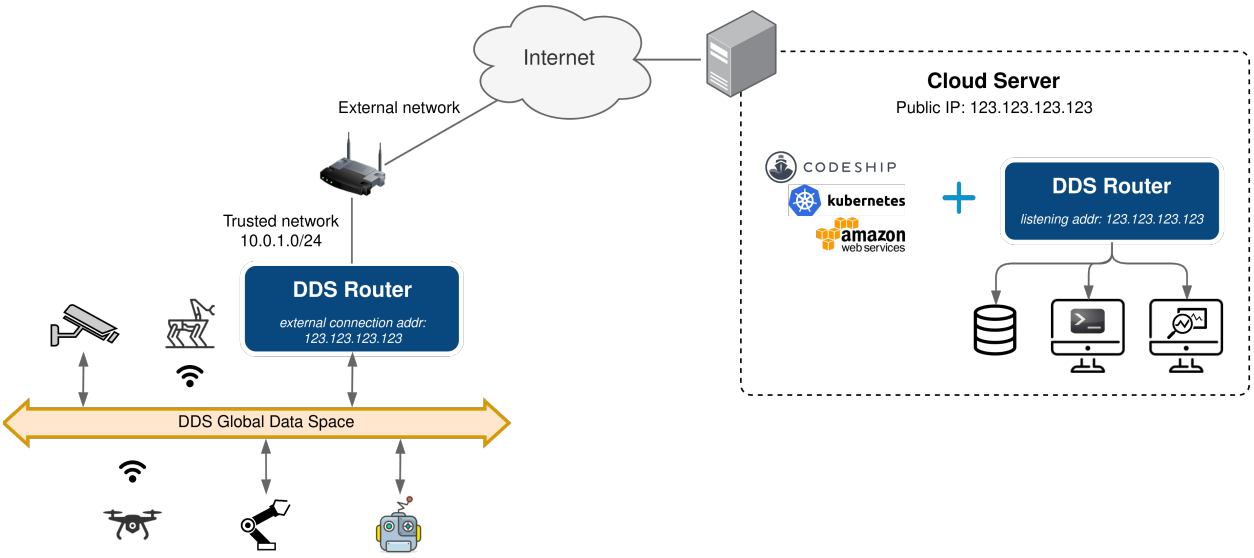
Index **113**

eProsima DDS Router is an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of *eProsima DDS Router*. This is achieved by deploying a *DDS Router* on an edge device of each local network so that the *DDS Router* routes DDS traffic from one network to the other through WAN communication.

Furthermore, *DDS Router* is a software designed for various forms of distributed networks, such as mesh networks in which nodes are deployed in different private local networks that are auto-discovered without any centralized network node, or cloud-based networks where there is a data processing cloud and multiple geographically distributed edge devices.

Following are some of the key features of *eProsima DDS Router*:

1. **WAN communication over TCP**: it supports WAN over TCP communication to establish DDS communications over the Internet.
2. **Distributed nature**: the user may deploy intermediate *DDS Router* nodes to discover new entities that enter and leave the network dynamically.
3. **Efficient data routing**: *DDS Router* avoids data introspection achieving a zero-copy system in data forwarding.
4. **Easy deployment**: it is based on an easily configurable modular system for users with no knowledge of computer networks.
5. **Topic allowlisting**: it is possible to configure a *DDS Router* to forward just the user data belonging to a topic specified by the user.
6. **Dynamic topic discovery**: the user does not need to fully specify over which topics to communicate (i.e. provide concrete topic names and types). The discovery of topics matching the allowlisting rules automatically triggers the creation of all entities required for communication.
7. **Quality of Service preservation**: DDS Router uses the QoS set in the user's DDS network and keeps the reliability and durability of the communication for each topic. These QoS are also manually configurable.


Internet
External network
Trusted network
10.0.1.0/24
DDS Router
external connection addr:
123.123.123.123
DDS Global Data Space
Cloud Server
Public IP: 123.123.123.123
CODESHIP
kubernetes
amazon
web services
DDS Router
listening addr: 123.123.123.123

CHAPTER

ONE

CONTACTS AND COMMERCIAL SUPPORT

Find more about us at eProsima's webpage.

Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48

CHAPTER

TWO

CONTRIBUTING TO THE DOCUMENTATION

DDS Router Documentation is an open source project, and as such all contributions, both in the form of feedback and content generation, are most welcomed. To make such contributions, please refer to the Contribution Guidelines hosted in our GitHub repository.

CHAPTER

THREE

STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the sections below.

- *Installation Manual*
- *Getting Started*
- *User Manual*
- *Examples*
- *Use cases*
- *Developer Manual*
- *Release Notes*

eProsima DDS Router is an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of *eProsima DDS Router*. This is achieved by deploying a *DDS Router* on an edge device of each local network so that the *DDS Router* routes DDS traffic from one network to the other through WAN communication.

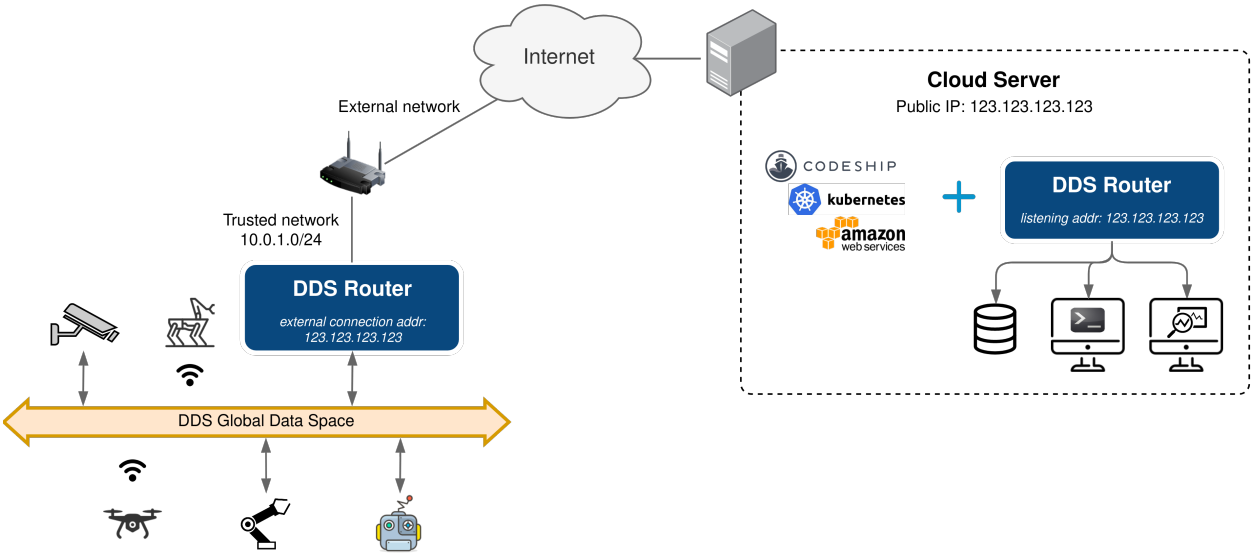
Furthermore, *DDS Router* is a software designed for various forms of distributed networks, such as mesh networks in which nodes are deployed in different private local networks that are auto-discovered without any centralized network node, or cloud-based networks where there is a data processing cloud and multiple geographically distributed edge devices.

3.1 Overview

Following are some of the key features of *eProsima DDS Router*:

1. **WAN communication over TCP**: it supports WAN over TCP communication to establish DDS communications over the Internet.
2. **Distributed nature**: the user may deploy intermediate *DDS Router* nodes to discover new entities that enter and leave the network dynamically.

3. **Efficient data routing**: *DDS Router* avoids data introspection achieving a zero-copy system in data forwarding.
4. **Easy deployment**: it is based on an easily configurable modular system for users with no knowledge of computer networks.
5. **Topic allowlisting**: it is possible to configure a *DDS Router* to forward just the user data belonging to a topic specified by the user.
6. **Dynamic topic discovery**: the user does not need to fully specify over which topics to communicate (i.e. provide concrete topic names and types). The discovery of topics matching the allowlisting rules automatically triggers the creation of all entities required for communication.
7. **Quality of Service preservation**: DDS Router uses the QoS set in the user's DDS network and keeps the reliability and durability of the communication for each topic. These QoS are also manually configurable.



3.2 Contacts and Commercial support

Find more about us at eProsima's webpage.

Support available at:

- Email: support@eprosima.com
- Phone: +34 91 804 34 48

3.3 Contributing to the documentation

DDS Router Documentation is an open source project, and as such all contributions, both in the form of feedback and content generation, are most welcomed. To make such contributions, please refer to the Contribution Guidelines hosted in our GitHub repository.

3.4 Structure of the documentation

This documentation is organized into the sections below.

- *Installation Manual*
- *Getting Started*
- *User Manual*
- *Examples*
- *Use cases*
- *Developer Manual*
- *Release Notes*

3.5 DDS Router on Windows

Warning: The current version of *DDS Router* does not have installers for Windows platforms. Please refer to the *Windows installation from sources* section to learn how to build *DDS Router* on Windows from sources.

3.6 DDS Router on Linux

Warning: The current version of *DDS Router* does not have installers for Linux platforms. Please refer to the *Linux installation from sources* section to learn how to build *DDS Router* on Linux from sources.

3.7 Docker image

eProsima distributes a Docker image of *DDS Router* with Ubuntu 22.04 as base image. This image launches an instance of *DDS Router* that is configured using a *YAML* configuration file provided by the user and shared with the Docker container. The steps to run *DDS Router* in a Docker container are explained below.

1. Download the compressed Docker image in `.tar` format from the eProsima Downloads website. It is strongly recommended to download the image corresponding to the latest version of *DDS Router*.

2. Extract the image by executing the following command:

   ```
   load ubuntu-ddsrouter:<version>.tar
   ```

   where `version` is the downloaded version of *DDS Router*.

3. Build a *DDS Router* configuration YAML file on the local machine. This will be the *DDS Router* configuration file that runs inside the Docker container. To continue this installation manual, let's use one of the configuration files provided in the *Examples* section. Open your preferred text editor and copy a full example from the *Examples* section into the `/<dds_router_ws>/DDS_ROUTER_CONFIGURATION.yaml` file, where `dds_router_ws` is the

path of the configuration file. To make this accessible from the Docker container we will create a shared volume containing just this file. This is explained in next point.

4. Run the Docker container executing the following command:

```
docker run -it \
    --net=host \
    -v /<dds_router_ws>/DDS_ROUTER_CONFIGURATION.yaml:/root/DDS_ROUTER_
↪CONFIGURATION.yaml \
    ubuntu-ddsrouter:v0.3.0
```

It is important to mention that both the path to the configuration file hosted in the local machine and the one created in the Docker container must be absolute paths in order to share just one single file as a shared volume.

After executing the previous command you should be able to see the initialization traces from the *DDS Router* running in the Docker container. If you want to terminate the application gracefully, just press `Ctrl+C` to stop the execution of *DDS Router*.

3.8 Project Overview

DDS Router is a cross-platform non-graphical application developed by eProsima and powered by Fast DDS that allows users to create a communication bridge that connects two DDS networks that otherwise would be isolated. The main use case of the *DDS Router* is to communicate two DDS networks that are physically or virtually separated and belong to different LANs, allowing the entities of each network to publish and subscribe to local and remote topics indistinctly.

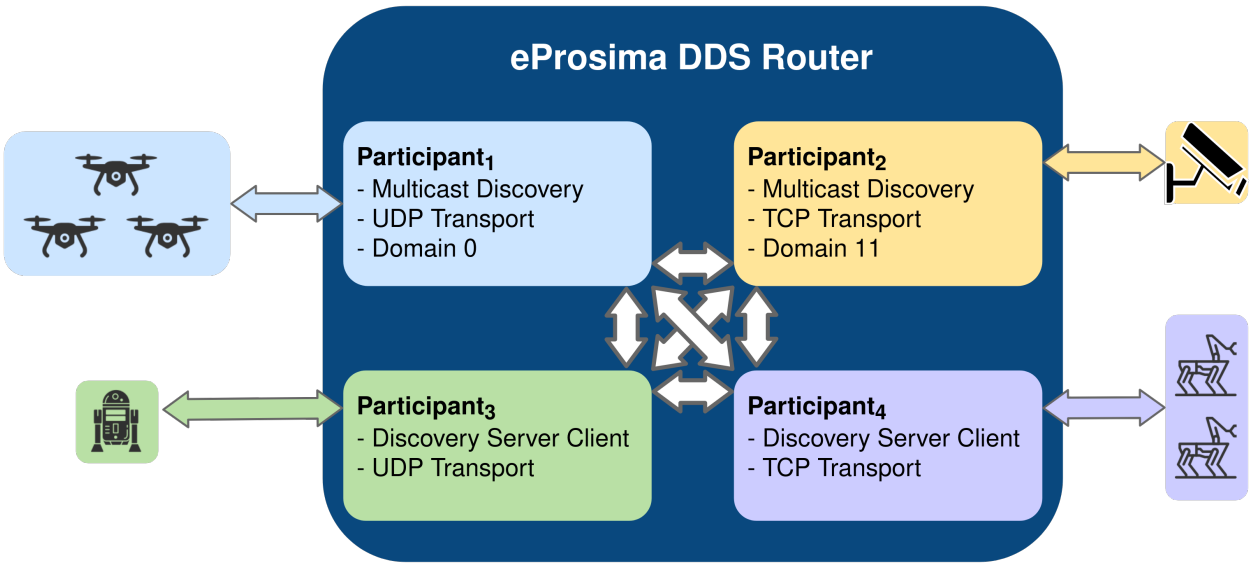
DDS Router is an application that internally runs *Participants*, which are an abstraction of DDS *DomainParticipants*. Each one of these Participants is an communication interface, a "door" to a specific DDS network configuration. These Participants allow the application to connect to different DDS networks at the same time. Every time one of these Participants receives a message from the DDS network to which they are connected, they will forward the data and the source of this message through the other Participants. The *DDS Router* configuration and the topics in which it operates depends on the initial *DDS Router configuration*.

The following schema represents a *DDS Router* local use case. This scenario presents different DDS networks that are isolated one to each other due to the Transport Protocol (UDP, TCP, etc.), the Discovery Protocol (Simple, Discovery Server, etc.) or the DDS *Domain Id* used by each DDS entity. Configuring the *DDS Router* to have 4 different Participants, each of them configured for one isolated DDS network, will create internally 4 Participants. All the data that arrives to one of the Participants will be forwarded through the others, allowing all the machines to connect to each other independently of their different configurations. This data transmission will be accomplished without copying the data, as all participants will share the pointer to the allocated data, successfully achieving a **zero-copy** communication mechanism.

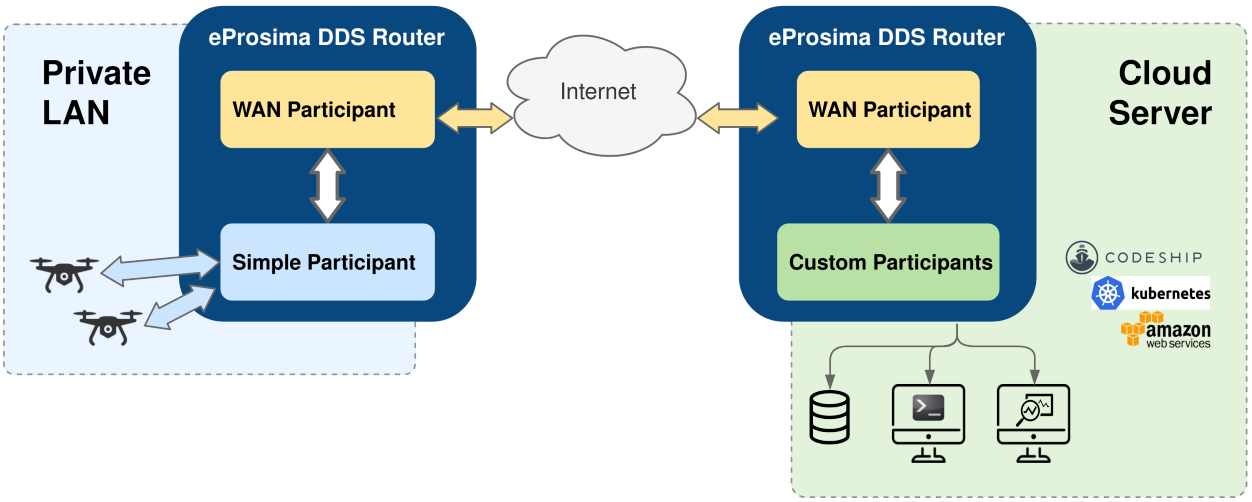
3.8.1 WAN Communication

Achieving a WAN communication of two networks that work in different LANs requires a running *DDS Router* application on each LAN. The *DDS Router* deployed will communicate with each network using DDS over WAN, and it will reroute every message received in LAN to the remote *DDS Router*. Once the remote Router receives data, it will transmit it to the local networks to which it is connected. This way, both DDS networks will behave as if they belonged to the same LAN.

Another important feature is that WAN communications are not limited to a single pair of *DDS Router*. The WAN communication may be performed using the eProsima Discovery Server discovery mechanism (dynamic discovery over non-multicast networks). Thus, any DDS Router connected to the same Discovery Servers will work as a standard DDS



node, publishing and subscribing on the shared DDS topics. This create an unlimited and highly scalable decentralized and distributed DDS network.



3.8.2 Usage Description

The *DDS Router* is a terminal (non-graphical) application that creates the DDS bridge as long as it is running. The configuration in *YAML* format is very intuitive and human-readable. The whole application has been thought to be user-friendly, following a user-oriented design.

- **RUN**: In order to run a *DDS Router* application, just a *YAML* configuration file is required with the specific configurations (see section *DDS Router Configuration* to check how to configure a DDS Router and see section *Application Arguments* to check the application supported arguments).
- **INTERACT**: Once the *DDS Router* application is running, the topics involved in this communication can be changed in runtime by just changing the *YAML* configuration file (see section *Reload Topics* for more details about re-configuring a running *DDS Router*).
- **STOP**: To stop the *DDS Router* just send a ^*C* signal to the process, and it will gracefully close the whole application (see section *Close Application* for more details on how to close the application).

3.8.3 Common Use Cases

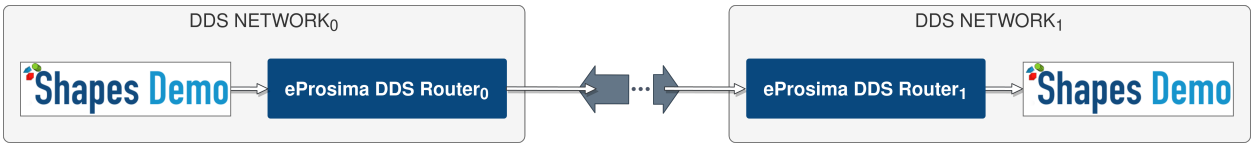
The different cases where the *DDS Router* could be applied are very varied, and would increase as new Participant Kinds will be added in future releases. These are most common use cases so far:

Use Case	Example
Communicating two different DDS Domain Ids	*Change Domain Example*
Communicating ROS 2 Discovery Server executions	*ROS2 Discovery Server Example*
WAN Communication	*WAN Example*

3.9 Example of usage

This example will serve as a hands-on tutorial, aimed at introducing some of the key concepts and features that *eProsima DDS Router* has to offer.

Two disjoint DDS networks will be bridged by a pair of routers to connect the endpoints hosted at each of the networks. In particular, two ShapesDemo instances will establish communication after proper configuration and deployment of the two aforementioned routers.



Note: This example applies to both LAN and WAN scenarios. For the WAN case, make sure that public IP addresses are used instead of private ones, and that the provided ports are reachable by properly configuring port forwarding in your Internet router devices.

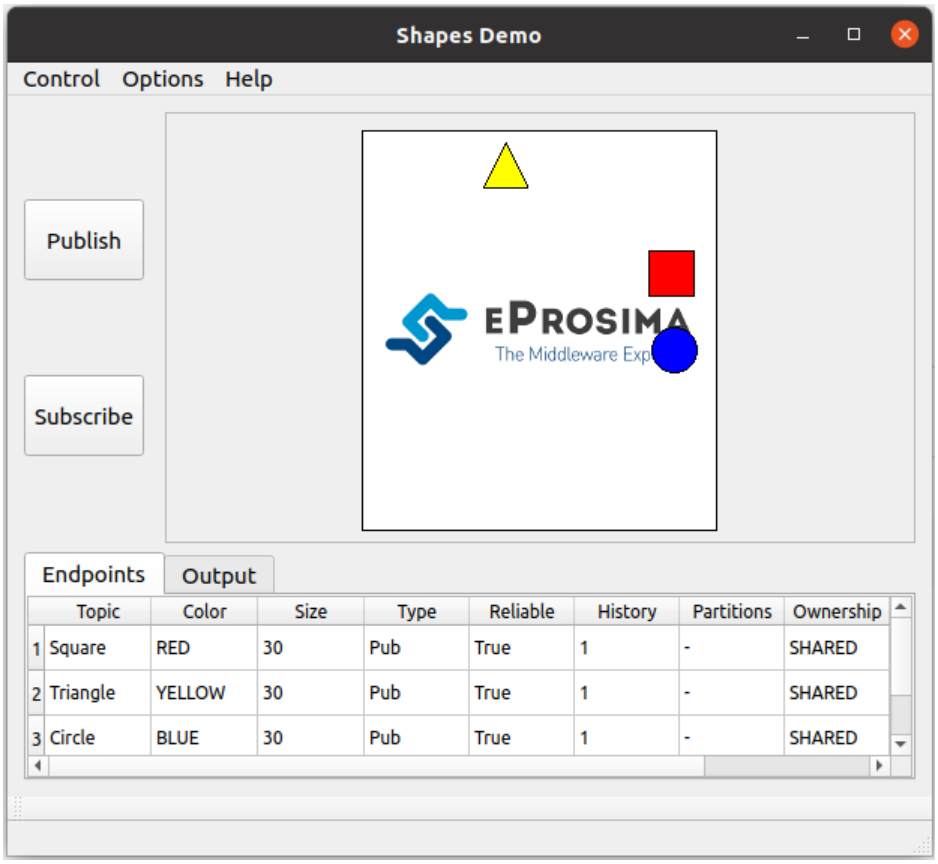
3.9.1 Launching ShapesDemo

ShapesDemo is an application that publishes and subscribes to shapes of different colors and sizes moving on a board. This is nothing more than a graphical tool to test the correctness of a specific DDS protocol implementation, as well as to prove interoperability with other implementations.

Let us launch a ShapesDemo instance in one of the DDS networks, and start publishing in topics `Square`, `Circle` and `Triangle` with default settings.

Now, run another instance in the other network and subscribe to the same topics chosen in the publisher side (use default settings).

Note: If you are trying this example in the LAN scenario, make sure a different DDS domain id is used in one of the ShapesDemo instances in order to avoid direct communication between them.



3.9.2 Router configuration

A configuration file is all that is required in order to run a *DDS Router* instance. In a nutshell, each router will forward messages if their associated topics match the filters contained in its `allowlist`. A `blocklist` may also be specified, on its own or in addition to an `allowlist`, but we will not be covering this here.

Let us first add only the `Square` topic:

```
allowlist:
  - name: Square
```

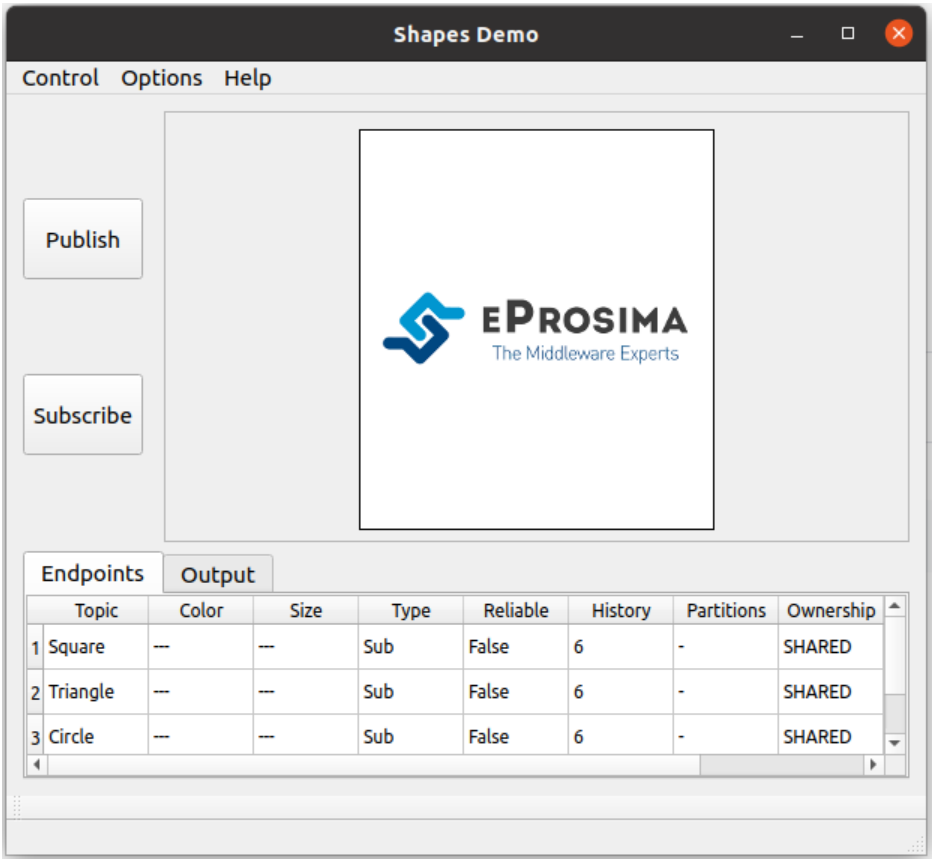
Apart from selecting on which topics we wish to send/receive data, we must configure as well the participants that will ultimately perform communication. Each router instance will contain a *simple* and a *WAN* participants. In brief, the simple participants will be in charge of locally communicating with the corresponding ShapesDemo application, while the WAN participants will be the ones bridging the connection between the two DDS networks.

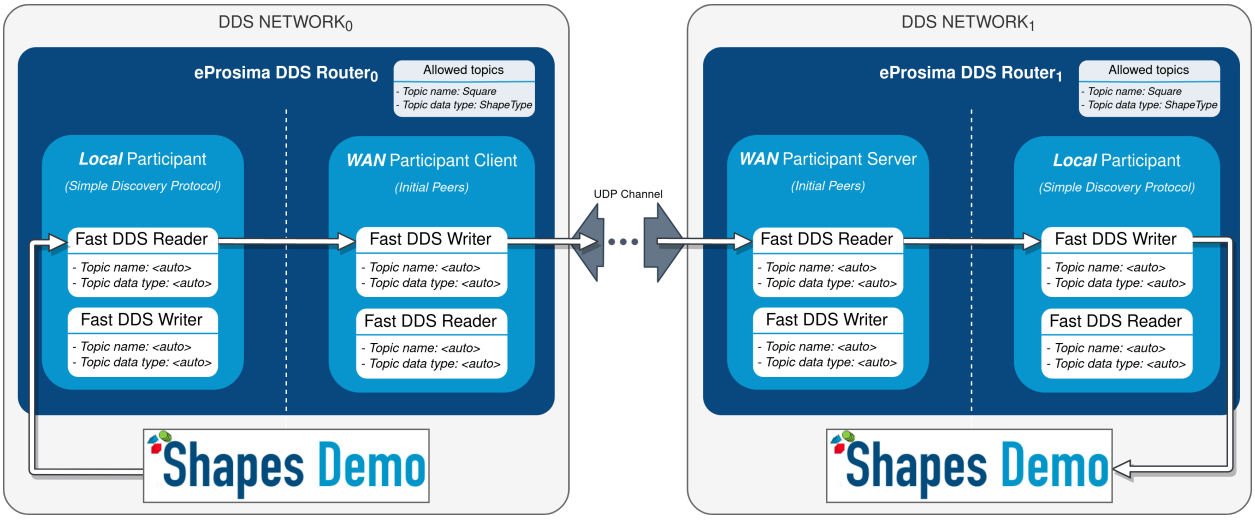
The only configuration required for simple participants is the DDS `domain` identifier.

```
  - name: SimpleParticipant
    kind: local
    domain: 0
```

If launching the two routers in the same LAN, set a different `domain` id in one of the two configuration files (same as the one previously set for ShapesDemo).

The configuration of WAN participants is more complex, and we will not be covering it here in detail. In short, in this example both WAN participants will communicate through UDP, with one being the client and the other one having


Shapes Demo
Control
Options
Help
Publish
Subscribe
eProsima
The Middleware Experts
Endpoints
Output
Topic
Color
Size
Type
Reliable
History
Partitions
Ownership
1 Square --- --- Sub False 6 - SHARED
2 Triangle --- --- Sub False 6 - SHARED
3 Circle --- --- Sub False 6 - SHARED


DDS NETWORK0
eProsima DDS Router0
Allowed topics
- Topic name: Square
- Topic data type: ShapeType
Local Participant
(Simple Discovery Protocol)
Fast DDS Reader
- Topic name: <auto>
- Topic data type: <auto>
Fast DDS Writer
- Topic name: <auto>
- Topic data type: <auto>
WAN Participant Client
(Initial Peers)
Fast DDS Writer
- Topic name: <auto>
- Topic data type: <auto>
Fast DDS Reader
- Topic name: <auto>
- Topic data type: <auto>
Shapes Demo
UDP Channel
DDS NETWORK1
eProsima DDS Router1
Allowed topics
- Topic name: Square
- Topic data type: ShapeType
WAN Participant Server
(Initial Peers)
Fast DDS Reader
- Topic name: <auto>
- Topic data type: <auto>
Fast DDS Writer
- Topic name: <auto>
- Topic data type: <auto>
Local Participant
(Simple Discovery Protocol)
Fast DDS Writer
- Topic name: <auto>
- Topic data type: <auto>
Fast DDS Reader
- Topic name: <auto>
- Topic data type: <auto>
Shapes Demo

the server role. Both participants are required to have a `listening address` (for the UDP case) where they will expect to receive traffic, and a `connection address` in the case of the client, which points at the server's `listening address`. Refer to *WAN Participant* and *WAN Configuration* for more information. You may also have a look at *WAN Example* for a detailed explanation on how to configure this kind of participant.

Following is an example of client and server configuration files:

```yaml
# client-ddsrouter.yaml

version: v4.0

allowlist:
  - name: Square

participants:

  - name: SimpleParticipant
    kind: local
    domain: 0

  - name: ClientWAN
    kind: wan
    connection-addresses:
      - ip: 192.168.1.8
        port: 11800
        transport: udp
    listening-addresses:
      - ip: 192.168.1.4
        port: 11008
        transport: udp
```

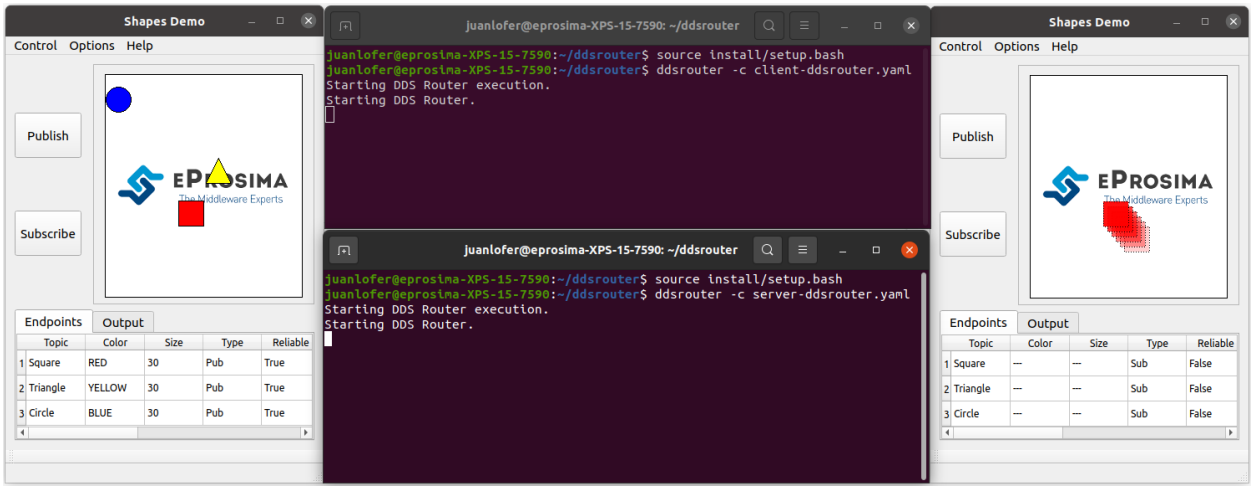
```yaml
# server-ddsrouter.yaml

version: v4.0

allowlist:
  - name: Square

participants:

  - name: SimpleParticipant
    kind: local
    domain: 1

  - name: ServerWAN
    kind: wan
    listening-addresses:
      - ip: 192.168.1.8
        port: 11800
        transport: udp
```

3.9.3 Router execution

Now, with the configuration files ready, launching a *DDS Router* instance is as easy as executing the following command:

```
ddsrouter -c config-file.yaml
```

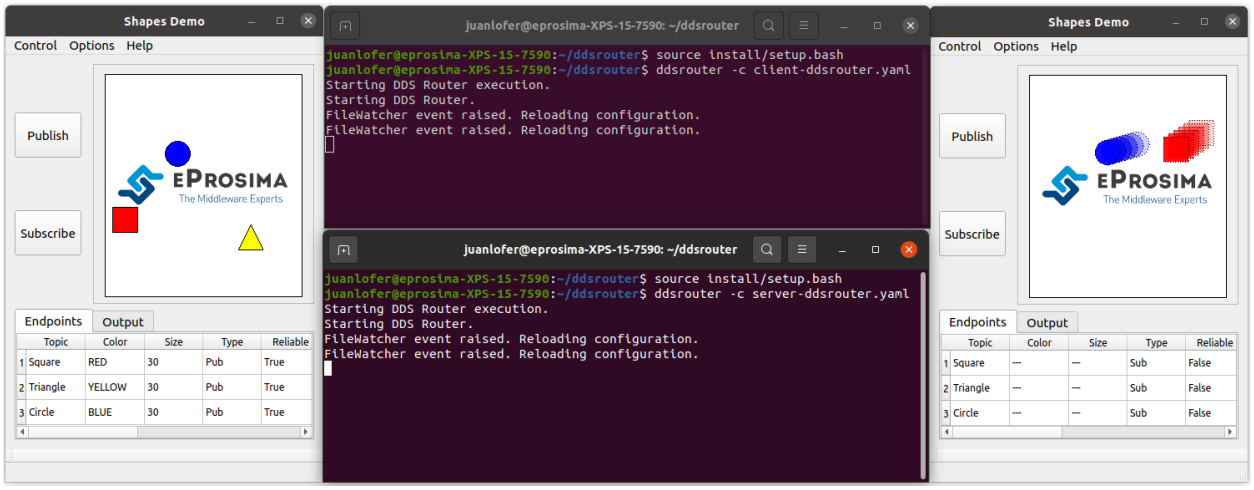
After setting up both routers, communication between the two ShapesDemo instances should have been established so the square shape is now visible in the subscriber's panel.



DDS Router supports the dynamic addition/deletion of topics at runtime (see *Reload Topics*). Let us test this feature by adding the circle topic to the allowlist of both routers. Also, by removing the square topic (removing this topic from one of the routers' allowlist will suffice) the square data should stop reaching the subscriber. Alternatively, the square topic may be added to the `blocklist`, achieving the same effect. See *Topic Filtering* for more details on allowlisting.

```
allowlist:
  - name: Circle
```

After applying these changes, the square should no longer be updated in the subscriber's side (appearing visible but frozen), while the circle should.



Please feel free to explore sections *Examples* and *Use Cases* for more information on how to configure and set up a router, as well as to discover multiple scenarios where *DDS Router* may serve as a useful tool.

3.10 User Interface

eProsima DDS Router is a user application executed from the command line and configured through a YAML configuration file.

- *Source Dependency Libraries*
- *Application Arguments*
- *Configuration File*
- *Reload Topics*
- *Log*
- *Close Application*

3.10.1 Source Dependency Libraries

eProsima DDS Router depends on *Fast DDS* `fastrtps` and `fastcdr` libraries. In order to correctly execute the Router, make sure that `fastrtps` and `fastcdr` are properly sourced.

```
source <path-to-fastdds-installation>/install/setup.bash
```

Note: If Fast DDS has been installed in the system, these libraries would be sourced by default.

3.10.2 Application Arguments

The *DDS Router* application supports several input arguments:

Command	Option	Long option	Value	Default Value
Help Argument	`-h`	`--help`		
Version Argument	`-v`	`--version`		
Configuration File Argument	`-c`	`--config-path`	Readable File Path	`./DDS_ROUTER_CONFIGURATION.yaml`
Reload Time Argument	`-r`	`--reload-time`	Unsigned Integer	`0`
Debug Argument	`-d`	`--debug`		
Log Verbosity Argument		`--log-verbosity`	`info` `warning` `error`	`warning`
Log Filter Argument		`--log-filter`	String	`"DDSROUTER"`

Help Argument

It shows the usage information of the application.

```
Usage: Fast DDS Router
Connect different DDS networks via DDS through LAN or WAN.
It will build a communication bridge between the different Participants included in the␣
↪provided configuration file.
To stop the execution gracefully use SIGINT (C^) or SIGTERM (kill) signals.
General options:

Application help and information.
-h --help           Print this help message.
-v --version        Print version, branch and commit hash.

Application parameters
-c --config-path    Path to the Configuration File (yaml format) [Default: ./DDS_ROUTER_
↪CONFIGURATION.yaml].
-r --reload-time    Time period in seconds to reload configuration file. This is needed␣
↪when FileWatcher functionality is not available (e.g. config file is a symbolic link).␣
↪Value 0 does not reload file. [Default: 0].
-t --timeout        Set a maximum time in seconds for the Router to run. Value 0 does␣
↪not set maximum. [Default: 0].

Debug options
-d --debug          Set log verbosity to Info (Using this option with --log-filter and/
↪or --log-verbosity will head to undefined behaviour).
    --log-filter     Set a Regex Filter to filter by category the info and warning log␣
↪entries. [Default = "DDSROUTER"].
    --log-verbosity  Set a Log Verbosity Level higher or equal the one given. (Values␣
↪accepted: "info","warning","error" no Case Sensitive) [Default = "warning"].
```

Version Argument

It shows the current version of the DDS Router and the hash of the last commit of the compiled code.

Configuration File Argument

Please refer to *Configuration File* for more information on how to build this configuration file.

Reload Time Argument

Set the *Reload Timer* in **seconds**.

Timeout Argument

This argument allow to set a maximum time while the application will be running. Setting this argument will set the number of seconds the application will run until it is killed. While the application is waiting for timeout, it is still possible to kill it via signal. Default value `0` means that the application will run forever (until kill via signal).

Debug Argument

This argument enables the *DDS Router* logs so the execution can be followed by internal debugging information. This argument sets *Log Verbosity Argument* to `info` and *Log Filter Argument* to `DDSROUTER`. For more information about debugging options, refer to *Log*.

Note: If this argument is used with any of the other arguments of debugging, the behavior depends on the order of parser of the arguments.

Log Verbosity Argument

Set the verbosity level so only log messages with equal or higher importance level are shown.

Log Filter Argument

Set a regex string as filter. Only log messages with a category that matches this regex will be printed (`ERROR` messages will be always shown unless *Log Verbosity Argument* is set to `ERROR`).

3.10.3 Configuration File

A *DDS Router* **requires** one and only one *YAML* configuration file as the operation of this application is configured via this *YAML* configuration file. Please refer to *Configuration* for more information on how to build this configuration file.

This *YAML* configuration file must be passed as argument to the *DDS Router* when executed. If no configuration file is provided as argument, the *DDS Router* will attempt to load a file named `DDS_ROUTER_CONFIGURATION.yaml` that must be in the same directory where the application is executed. If no configuration file is passed as argument, and the default configuration file does not exist in the current directory, the application will fail.

3.10.4 Reload Topics

The topics that the *DDS Router* is routing could be changed at runtime. Including topics in configuration's `allowlist` will create new *Writers* and *Readers* for each Participant in the Router. Removing a topic from `allowlist` will disable this topic, and so it will stop routing data in such topic. Be aware that disabling a topic does not eliminate the entities of that topic. So, if a topic has been active before, the Writers and Readers will still be present in the *DDS Router* and will still receive data.

There exist two methods to reload the list of allowed topics, an active and a passive one. Both methods work over the same configuration file with which the *DDS Router* has been initialized.

File Watcher

A File Watcher is a process that runs in the background and tracks changes in the *DDS Router* configuration file. Every time the file is changed, the OS sends a notification, and the File Watcher listens such notification and interacts with the *DDS Router* in order to reload the topics. This event occurs every time the configuration file is saved.

FileWatcher is used in every *DDS Router* execution by default. However, this method does not work properly in specific scenarios where the file being watched is not a real file but a link (e.g. Kubernetes executions).

Reload Timer

A timer can be set in order to periodically reload the configuration file. The configuration file will be automatically reloaded according to the specified time period.

3.10.5 Log

Log module of *DDS Router* uses the *Fast DDS* logging module. This log has 3 severity levels: `INFO`, `WARNING` and `ERROR`. Every log has also a category associated. This is how a log looks like:

```
Date                       Category         Severity     Log message                 ␣
↪Function
2022-11-16 14:58:13.375 [MODULE_SUBMODULE Error] It has happen ... because of ... ->␣
↪Function main
```

Every log entry has several parts:

- **Date**: format: `year-month-day hour::minute::second::millisecond` with millisecond accuracy. This is the time when the log was added to the log queue, not when it is printed.
- **Category**: Reference to the module where the log was raised. It is used to filter logs.
- **Severity**: Could be `Info`, `Warning` or `Error`.
- **Log message**: The actual log message.
- **Function**: Name of the function or method that has produced this log entry.

Note: For `INFO` logs to be compiled, the *DDS Router* must have been compiled with CMake option `CMAKE_BUILD_TYPE=Debug`, or compiled with CMake option `LOG_INFO=ON`.

If Fast DDS has been compiled in debug mode, it will print the logs of the DDS Router and Fast DDS mixed. In order to skip Fast DDS logs, compile `fastrtps` library with CMake option `-DLOG_NO_INFO=ON` or `CMAKE_BUILD_TYPE` different to `Debug`, or use the argument ``

3.10.6 Close Application

In order to stop a *DDS Router* application, use one of the following OS signals:

SIGINT

Send an interruption `SIGINT | ^C` signal *(signal value 2)* to the process. Press `Ctrl + C` in the terminal where the process is running.

SIGTERM

Send an interruption `SIGTERM` signal *(signal value 15)* to the process by executing the command `kill <pid>` in a different terminal, where `<pid>` is the id of the process running the *DDS Router*. Use the `ps` or `top` programs to check the processes' ids.

TIMEOUT

Setting a maximum amount of seconds that the application will work using argument `--timeout` will close the application once the time has expired.

3.11 DDS Router Participant

DDS Router *Participant* is a *DDS Router* entity that works as an interface between a network and the core of the router. Participants are the main elements inside the *DDS Router* functionality.

- *Participant*
  - *Participant Name*
  - *Participant Kind*
- *Participant creation*
- *Participant kinds*

3.11.1 Participant

A Participant is an abstraction over the DDS *DomainParticipant*. This entity manages the dynamic discovery of DDS entities on a specific network or interface. Each Participant is uniquely identified by a *Participant Name* in a *DDS Router* execution and has a predefined *Participant Kind* that specifies the internal general functionality of the Participant.

Participant Name

It is an alphanumeric string that uniquely identifies a Participant in a *DDS Router* execution.

Participant Kind

It specifies the kind of the Participant. There are several Participant kinds already defined, which will specify in general terms how the Participant behaves.

3.11.2 Participant creation

Each participant configuration is specified as a different item of `participants` array, and each of these configurations has a unique Participant Name that should not be repeated in a *DDS Router* execution.

Each Participant Kind is associated with one or several names or aliases that represent it. In order to use a Participant of a specific kind, use `kind` tag in the yaml configuration file. If the kind is not any of the valid aliases, the Participant will not be created and the execution will fail.

Note: There could be as many Participants as required, and their kinds could be repeated, but all names must be unique.

Below are some examples on how to configure a Participant:

```
- name: participant_1  # New Participant with Name = 'participant_1'
  kind: simple         # 'participant_1' will be created of kind 'simple'
  extra_configuration: ...
```

3.11.3 Participant kinds

Below is the list with all the available Participant Kinds.

Participant Kind	Aliases	Specific configuration tags	Description
Echo Participant	`echo`	`discovery data verbose`	Print in *stdout* all user and/or discovery data received.
Simple Participant	`simple local`	`domain`	Simple DDS DomainParticipant.
Local Discovery Server Participant	`discovery-server local-ds ds`	`guid listening-addresses connection-addresses tls`	Discovery Server DDS DomainParticipant for local communication.
Discovery Server WAN Participant	`wan-discovery-server wan-ds`	`guid listening-addresses connection-addresses tls`	Discovery Server DDS DomainParticipant for WAN communication.
WAN Participant	`wan router initial-peers`	`guid listening-addresses connection-addresses tls`	Initial Peers DDS DomainParticipant for WAN communication.
XML Participant	`xml XML`	`profile`	XML DDS DomainParticipant for custom configuration.

Echo Participant

This *Participant* prints in `stdout` all the discovery information and/or user data that is received by the *DDS Router*.

In the case of discovery traces, messages such as the following will be displayed:

```
New endpoint discovered: Endpoint{<endpoint_guid>;<endpoint_kind>;<topic>}.
```

For data reception messages, the traces show the following information:

```
Received data in Participant: <participant_id> in topic: <topic>.
```

These logs contain the *Participant Name* of the participant that has originally received the message, and the *Topic* where this message has been received. Additionally, extra information such as the data *Payload* (in hexadecimal format) and source *Endpoint Guid* is displayed in verbose mode:

```
In Endpoint: <endpoint_guid> from Participant: <participant_id> in topic: <topic>␣
↪payload received: <payload> with specific qos: <specific_qos>.
```

Notice that this Payload is the same that a standard DDS *DataReader* will receive if it is connected to one of the Participants of the *DDS Router*.

Note: This Participant does not perform any discovery or data reception functionality.

Use case

Use this Participant in order to see in `stdout` the data that is being relayed by the router, as well as information regarding discovery events. All the data received by any of the Participants of the router will be printed (if `data` is `true`) with its topic and source guid, along with the payload (in `verbose` mode).

Kind aliases

- `echo`

Configuration

The Echo Participant accepts three different **optional** parameters:

- `discovery`: Whether to echo information regarding discovery events. Defaults to **true**.
- `data`: Whether to echo information regarding user data reception. Defaults to **false**.
- `verbose`: Display detailed information about the user data received (if `data` set to `true`). Defaults to **false**.

Configuration Example

```yaml
- name: echo_participant     # Participant Name = echo_participant
  kind: echo
  data: true                 # Print a trace with every arrival of user data
  verbose: true              # Show detailed information on user data reception
  discovery: false           # Do not print traces regarding discovery events
```

Simple Participant

This kind of *Participant* refers to a Simple DDS *DomainParticipant*. This Participant will discover all Participants deployed in its own local network in the same domain via multicast communication, and will communicate with those that share publication or subscription topics.

Use case

Use this Participant in order to communicate an internal standard DDS network, such as a ROS 2 or Fast DDS network in the same LAN.

Kind aliases

- `simple`
- `local`

Configuration

The only configuration required to start a Simple Participant is the *Domain Id* on which it will listen for DDS communications. Check *Configuration section* for further details.

Configuration Example

```yaml
- name: simple_participant     # Participant Name = simple_participant
  kind: simple
  domain: 2                    # Domain Id = 2
```

Local Discovery Server Participant

This kind of *Participant* refers to a *Discovery Server DomainParticipant*. This Participant will work as discovery broker for those Participants that connect to it (clients or servers). It could also connect to one or multiple Discovery Servers to create a Discovery Server Network.

Use case

Use this Participant in order to communicate an internal DDS network using Discovery Server. This is highly useful in networks that do not support multicast communication; or to reduce the number of meta-traffic packets exchanged in discovery, reducing the network traffic in the discovery process.

Kind aliases

- `discovery-server`
- `local-ds`
- `ds`
- `local-discovery-server`

Configuration

The Local Discovery Server Participant allows users to configure the standard attributes of a Discovery Server.

- To configure the Discovery Server *GuidPrefix*, check the *Domain Id* configuration section.
- To configure the Discovery Server listening addresses, check the *Listening Addresses* configuration section.
- To configure the Discovery Server connection addresses to connect to other Discovery Servers, check the *Discovery Server Connection Addresses* configuration section.

Note: The network addresses set in *listening-addresses* and *connection-addresses* use `UDP` transport by default if the transport is not specified in the address configuration.

Configuration Example

Configure a Local Discovery Server setting the GuidPrefix used for ROS 2 deployments with id `2` (`44.53.02.5f.45.50.52.4f.53.49.4d.41`). It listens for clients in *localhost* in ports `11600` in `UDP` and `11601` in `TCP`. This example connects the local Discovery Server Participant with a remote Discovery Server listening in IPv6 address `2001:4860:4860::8888` and port `11666` and configured with `01.0f.04.00.00.00.00.00.00.00.ca.fe` Discovery Server GuidPrefix.

```
- name: local_discovery_server_participant        # Participant Name = local_discovery_
↪server_participant

  kind: discovery-server

  discovery-server-guid:
    id: 2
    ros-discovery-server: true                    # ROS Discovery Server id =>␣
↪GuidPrefix = 44.53.02.5f.45.50.52.4f.53.49.4d.41

  listening-addresses:                            # Local Discovery Server Listening␣
↪Addresses
    - ip: 127.0.0.1                               # Use UDP by default
```

(continues on next page)

(continued from previous page)

```
      port: 11600
    - ip: 127.0.0.1
      port: 11601
      transport: tcp                                # Use TCP transport

  connection-addresses:                             # External Discovery Server Listening␣
↪Addresses
    - discovery-server-guid:
        id: 4                                       # External Discovery Server id =>␣
↪GuidPrefix = 01.0f.04.00.00.00.00.00.00.00.ca.fe
      addresses:
        - ip: 2001:4860:4860::8888                  # Use UDP by default
          port: 11666
```

Discovery Server WAN Participant

This type of *Participant* refers to a *Discovery Server DomainParticipant* that communicates with other **Discovery Server WAN** Participants in different networks. This Participant will work as bridge for every Participant working locally in the LAN and any other LAN that has a *DDS Router* with an active Discovery Server WAN Participant.

Warning: Do not try to communicate a Discovery Server WAN Participant with any other kind of Participant that is not of type Discovery Server WAN Participant.

Use case

Use this Participant to communicate an internal DDS network with other LANs through a WAN communication. Each of the networks to be connected require a running *DDS Router*, and the messages will be relay from one to another depending on the topics filtered by each of them.

Kind aliases

- `wan-discovery-server`
- `wan-ds`

Configuration

The WAN Discovery Server Participant allows users to configure the standard attributes of a Discovery Server.

- To configure the Discovery Server *GuidPrefix*, check the *Domain Id* configuration section.
- To configure the Discovery Server listening addresses, check the *Listening Addresses* configuration section.
- To configure the Discovery Server connection addresses to connect to other Discovery Servers, check the *Discovery Server Connection Addresses* configuration section.

Note: The network addresses set in *listening-addresses* and *connection-addresses* use `UDP` transport by default if the transport is not specified in the address configuration.

WAN Configuration

Refer to section *WAN Configuration* for a detailed explanation on how to correctly configure the *DDS Router* for WAN communication.

Configuration Example

Configure a WAN Discovery Server with GuidPrefix id `2` (`01.0f.02.00.00.00.00.00.00.00.ca.fe`). It listens for clients in public IP `82.0.0.1` in port `11600` in `TCP`. It connects with a remote WAN Participant in IPv6 address `2001:4860:4860::8888` and port `11666` which Discovery Server GuidPrefix is `01.0f.04.00.00.00.00.00.00.00.ca.fe` using `UDP` transport.

```yaml
- name: wan_participant                          # Participant Name = wan_participant

  kind: wan-discovery-server

  discovery-server-guid:
    id: 2                                        # GuidPrefix = 01.0f.02.00.00.00.00.00.00.
↪00.ca.fe

  listening-addresses:                           # WAN Discovery Server Listening Addresses
    - ip: 82.0.0.1                               # Use UDP by default
      port: 11600

  connection-addresses:                          # Another WAN Participant Listening␣
↪Addresses
    - discovery-server-guid:
        id: 4                                    # External Discovery Server id =>␣
↪GuidPrefix = 01.0f.04.00.00.00.00.00.00.00.ca.fe
      addresses:
        - ip: 2001:4860:4860::8888
          port: 11666
          transport: udp                         # Use UDP transport
```

WAN Participant

This type of *Participant* refers to a *Initial Peers DomainParticipant* that communicates with other **WAN** Participants in different networks. This Participant will work as bridge for every Participant working locally in the LAN and any other LAN that has a *DDS Router* with an active WAN Participant.

Warning: Do not try to communicate a WAN Participant with any other kind of Participant that is not of type WAN Participant.

Use case

Use this Participant to communicate an internal DDS network with other LANs through a WAN communication. Each of the networks to be connected require a running *DDS Router*, and the messages will be relayed from one to another depending on the topics filtered by each of them.

Kind aliases

- `wan`
- `router`
- `initial-peers`

Configuration

The WAN Participant allows users to configure the listening and connection addresses (whether it should locate or be located by other remote Participants):

- To configure the listening addresses, check the *Listening Addresses* configuration section.
- To configure the connection addresses to connect with other Wan Participants, check the *Initial Peers Connection Addresses* configuration section.

Note: The network addresses set in *listening-addresses* and *connection-addresses* use `UDP` transport by default if the transport is not specified in the address configuration.

Repeater

This Participant allows a tag `repeater` to be used as a *TURN* server. Please refer to section *Repeater DDS Router* for more information.

WAN Configuration

Refer to section *WAN Configuration* for a detailed explanation on how to correctly configure the *DDS Router* for WAN communication.

Configuration Example

Configure a WAN Participant. It listens for clients in public IP `82.0.0.1` in port `11600` in `TCP`. It connects with a remote Participant in IPv6 address `2001:4860:4860::8888` and port `11666` using `TCP` transport.

```yaml
- name: wan_participant                     # Participant Name = wan_participant

  kind: wan

  listening-addresses:                      # WAN Participant Listening Addresses
    - ip: 82.0.0.1                          # Use UDP by default
      port: 11600
```

(continues on next page)

(continued from previous page)

```
  connection-addresses:                     # Another WAN Participant Listening␣
↪Addresses
    - ip: 2001:4860:4860::8888
      port: 11666
      transport: tcp                        # Use UDP transport
```

XML Participant

This type of *Participant* refers to a *DomainParticipant* that uses QoS profiles loaded from XML files to be configured.

Fast DDS supports XML to fully configure a DomainParticipant. Using XML configuration, users have whole access to the full configuration of a DDS DomainParticipant. Check the following *documentation <https://fast-dds.docs.eprosima.com/en/latest/fastdds/xml_configuration/domainparticipant.html>* for further information on how to configure a DDS DomainParticipant with XML. For further information regarding how to load XML configuration files to the *DDS Router*, check the *Load XML Configuration* section.

Note: This kind of Participant is meant for advanced users as XML profiles will overwrite the default internal settings of the DDS Router.

Use case

Use this Participant to fully configure a DomainParticipant, its discovery methods, transport options, DDS QoS, etc. The main use case for this Participant is using **DDS Security**, which requires XML configuration from the user's side.

Warning: This Participant kind does not support *RPC*. Thus services and actions of ROS 2 will not work correctly.

Kind aliases

- `xml`
- `XML`

Configuration

The XML Participant allows setting a profile name for the internal DomainParticipant of the *DDS Router*. Such profile name will be used as the QoS profile when creating the internal DomainParticipant.

Create a Fast DDS XML Participant profile

The whole DomainParticipant configuration settings must be configured via XML, *DDS Router* will not configure any attribute or QoS for it. To configure the profile, check the *Profile* configuration section.

However, there are specific QoS that will affect the performance of the *DDS Router* and that are advisable for the user to set them. Notice that not setting such QoS will not affect the correct functionality of the application, but may affect its performance.

- `ignore_local_endpoints` avoid local matching for this participant's endpoints:

```
<participant profile_name="ignore_local_endpoints_domainparticipant_xml_
↪profile">
    <rtps>
        <propertiesPolicy>
            <properties>
                <property>
                    <name>fastdds.ignore_local_endpoints</name>
                    <value>true</value>
                </property>
            </properties>
        </propertiesPolicy>
    </rtps>
</participant>
```

Configuration Example

Configure a XML Participant that gets all of its QoS from XML profile named `custom_participant_configuration`. This XML profile must be previously loaded. Use *Fast DDS* or *DDS Router* support to load XML configuration files as explained in *this section*.

```
- name: xml_participant                        # Participant Name = xml_participant

  kind: xml

  profile: custom_participant_configuration    # Configure participant with this profile
```

3.12 Configuration

A *DDS Router* is configured by a *.yaml* configuration file. This *.yaml* file contains all the information regarding the *DDS Router* configuration, such as topics filtering and *Participants* configurations.

- *Configuration version*
- *Load XML Configuration*
  - *Load XML Files*
  - *Raw XML*
- *Topics Configuration*
  - *Built-in Topics*

    - *Topic Filtering*
    - *Topic QoS*
    - *Manual Topics*
- *Specs Configuration*
    - *Number of Threads*
    - *Remove Unused Entities*
    - *Discovery Trigger*
    - *QoS*
    - *Logging*
    - *Monitor*
- *Participant Configuration*
    - *Domain Id*
    - *Ignore Participant Flags*
    - *Custom Transport Descriptors*
    - *Interface Whitelist*
    - *Repeater Participant*
    - *Network Address*
    - *External Port*
    - *Discovery Server GuidPrefix*
    - *Listening Addresses*
    - *Initial Peers Connection Addresses*
    - *Discovery Server Connection Addresses*
    - *Profile*
    - *QoS*
- *Forwarding Routes*
    - *Generic Routes*
    - *Topic Routes*
- *General Example*

3.12.1 Configuration version

The YAML Configuration supports a `version` value to identify the configuration version to parse the file. In future releases the YAML format (some key words, fields, etc.) may change. This value allows users to keep using the same YAML file with an old configuration format, maintaining compatibility with future releases.

The current configuration version is v4.0. This is the configuration version that is described along this page.

Note: The current default version when the tag `version` is not set is *v4.0*.

Warning: **Deprecation warning**. Update to version *v4.0* since previous *v3.1* is no longer supported.

3.12.2 Load XML Configuration

Fast DDS supports configuration of its internal entities (*DomainParticipant*, *DataWriter*, etc.) via XML Profiles. These XML files contain different profiles that set specific QoS, and entities can be created following such profiles. These XML files can be loaded in the process by their *default file name* or by an environment variable. Check the Fast DDS documentation for more information.

Another way of loading these XML configurations is using the *DDS Router* yaml configuration. The YAML Configuration supports a `xml` **optional** tag that contains certain options to load Fast DDS XML configurations. XML configurations are then used to configure an *XML Participant*.

Load XML Files

Under the **optional** tag `files`, a list can be set with the names of files to load XML from.

Raw XML

Under the **optional** tag `raw`, an XML configuration (with the same format as an XML file) can be set as a string to be loaded.

```yaml
xml:

  files:
      - "./xml_configuration.xml"

  raw: |
      <?xml version="1.0" encoding="UTF-8" ?>
      <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
          <participant profile_name="custom_participant_configuration">
              <domainId>1</domainId>
              <rtps></rtps>
          </participant>
      </profiles>
```

Note: The *DDS Router* does not modify any XML configuration in a *XML Participant*. However, there are some QoS that can affect performance. These QoS should be configured by the user explicitly. Check *Create a Fast DDS XML*

Participant profile.

3.12.3 Topics Configuration

Built-in Topics

The discovery phase can be accelerated by listing topics under the `builtin-topics` tag. The *DDS Router* will create the DataWriters and DataReaders for these topics in the *DDS Router* initialization. The *Topic QoS* for these topics can be manually configured with a *Manual Topic*, a *Participant Topic QoS*, and a *Specs Topic QoS*; if a *Topic QoS* is not configured, it will take its default value.

The `builtin-topics` must specify a `name` and `type` without wildcard characters.

```
builtin-topics:
  - name: HelloWorldTopic
    type: HelloWorld
```

Topic Filtering

The *DDS Router* automatically detects the topics that are being used in a DDS Network. The *DDS Router* then creates internal DDS *Writers* and *Readers* for each participant in each topic, and forwards the data published on each topic. The *DDS Router* allows filtering DDS *Topics* to allow users to configure the DDS *Topics* that must be forwarded. These data filtering rules can be configured under the `allowlist` and `blocklist` tags. If the `allowlist` and `blocklist` are not configured, the *DDS Router* will forward all the data published on the topics it discovers. If both the `allowlist` and `blocklist` are configured and a topic appears in both of them, the `blocklist` has priority and the topic will be blocked.

Topics are determined by the tags `name` (required) and `type`, both of which accept wildcard characters.

Note: Placing quotation marks around values in a YAML file is generally optional, but values containing wildcard characters do require single or double quotation marks.

Consider the following example:

```
allowlist:
  - name: AllowedTopic1
    type: Allowed

  - name: AllowedTopic2
    type: "*"

  - name: HelloWorldTopic
    type: HelloWorld

blocklist:
  - name: "*"
    type: HelloWorld
```

In this example, the data in the topic `AllowedTopic1` with type `Allowed` and the data in the topic `AllowedTopic2` with any type will be forwarded by the *DDS Router*. The data in the topic `HelloWorldTopic` with type `HelloWorld` will be blocked, since the `blocklist` is blocking all topics with any name and with type `HelloWorld`.

Topic QoS

The following is the set of QoS that are configurable for a topic. For more information on topics, please read the Fast DDS Topic section.

Quality of Service	Yaml tag	Data type	Default value	QoS set
Reliability	`reliability`	*bool*	`false`	`RELIABLE` / `BEST_EFFORT`
Durability	`durability`	*bool*	`false`	`TRANSIENT_LOCAL` / `VOLATILE`
Ownership	`ownership`	*bool*	`false`	`EXCLUSIVE_OWNERSHIP_QOS` / `SHARED_OWNERSHIP_QOS`
Partitions	`partitions`	*bool*	`false`	Topic with / without partitions
Key	`keyed`	*bool*	`false`	Topic with / without key
History Depth	`history-depth`	*unsigned integer*	`5000`	*History Depth*
Max Transmission Rate	`max-tx-rate`	*float*	`0` (unlimited)	*Max Transmission Rate*
Max Reception Rate	`max-rx-rate`	*float*	`0` (unlimited)	*Max Reception Rate*
Downsampling	`downsampling`	*unsigned integer*	1	*Downsampling*

Warning: Manually configuring `TRANSIENT_LOCAL` durability may lead to incompatibility issues when the discovered reliability is `BEST_EFFORT`. Please ensure to always configure the `reliability` when configuring the `durability` to avoid the issue.

History Depth

The `history-depth` tag configures the history depth of the Fast DDS internal entities. By default, the depth of every RTPS History instance is `5000`, which sets a constraint on the maximum number of samples a *DDS Router* instance can deliver to late joiner Readers configured with `TRANSIENT_LOCAL` DurabilityQosPolicyKind. Its value should be decreased when the sample size and/or number of created endpoints (increasing with the number of topics and *DDS Router* participants) are big enough to cause memory exhaustion issues. If enough memory is available, however, the `history-depth` could be increased to deliver a greater number of samples to late joiners.

Max Transmission Rate

The `max-tx-rate` tag limits the frequency [Hz] at which samples are sent by discarding messages transmitted before `1/max-tx-rate` seconds have passed since the last sent message. It only accepts non-negative numbers. By default it is set to `0`; it sends samples at an unlimited transmission rate.

Max Reception Rate

The `max-rx-rate` tag limits the frequency [Hz] at which samples are processed by discarding messages received before `1/max-rx-rate` seconds have passed since the last processed message. It only accepts non-negative numbers. By default it is set to `0`; it processes samples at an unlimited reception rate.

Downsampling

The `downsampling` tag reduces the sampling rate of the received data by only keeping *1* out of every *n* samples received (per topic), where *n* is the value specified under the `downsampling` tag. When the `max-rx-rate` tag is also set, downsampling only applies to messages that have passed the `max-rx-rate` filter. It only accepts positive integers. By default it is set to `1`; it accepts every message.

Manual Topics

A subset of *Topic QoS* can be manually configured for a specific topic under the tag `topics`. The tag `topics` has a required `name` tag that accepts wildcard characters. It also has three optional tags: a `type` tag that accepts wildcard characters, a `qos` tag with the *Topic QoS* that the user wants to manually configure, and a `participants` tag that lists the participants to which the configuration applies. If a `qos` is not manually configured, it will get its value by discovery; if the `participants` tag is empty or non-existent, the configuration will apply to all participants.

Example of usage

```yaml
topics:
  - name: "temperature/*"
    type: "temperature/types/*"
    qos:
      max-tx-rate: 15
      downsampling: 2
    participants:
      - Participant0
      - Participant1
```

Note: The *Topic QoS* configured in the Manual Topics take precedence over the *Participant Topic QoS* and the *Specs Topic QoS*.

3.12.4 Specs Configuration

The YAML Configuration supports a `specs` **optional** tag that contains certain options related with the overall configuration of the DDS Router instance to run. The values available to configure are:

Number of Threads

`specs` supports a `threads` **optional** value that allows the user to set a maximum number of threads for the internal `ThreadPool`. This ThreadPool allows to limit the number of threads spawned by the application. This improves the performance of the data transmission between Participants.

This value should be set by each user depending on each system's characteristics. In case this value is not set, the default number of threads used is `12`.

Remove Unused Entities

`specs` supports a `remove-unused-entities` **optional** value that configures the deletion of unused internal entities in the *DDS Router*. By default, unused internal entities are *not* removed. Thus, when the *DDS Router* discovers a Subscriber (by default; see *Discovery Trigger*), the *DDS Router* creates entities in all of its participants, and these entities stay up even after the Subscriber disconnects.

At times it can be useful to remove the internal entities that are not being used. Consider the following example. Two *DDS Routers* are communicating through a WAN connection, when the last of the external Subscribers to which they are forwarding data disconnects. By default, the internal entities of the *DDS Routers* would *not* be removed, so the *DDS Routers* would keep consuming bandwidth, even though the data is never read. By setting the `remove-unused-entities` option to `true`, the internal entities of the *DDS Router* would be removed, and the *DDS Routers* would stop communicating and free up the bandwidth.

Warning: At the time being, the removal of unused entities is incompatible with the Transient-Local Durability QoS.

Warning: At the time being, `remove-unused-entities: true` is only compatible with a *discovery-trigger* set to `reader`.

Note: The `remove-unused-entities` option doesn't apply to *Built-in Topics* since they are created before being discovered by a *Participant*.

Discovery Trigger

`specs` supports a `discovery-trigger` **optional** value that configures what type of external entity triggers the creation/removal of entities in the *DDS Router*. The possible values for the `discovery-trigger` are:

Value	Tag	Description
Reader	`reader`	The creation/removal of readers triggers the creation/removal of internal entities.
Writer	`writer`	The creation/removal of writers triggers the creation/removal of internal entities.
Any	`any`	The creation/removal of readers or writers triggers the creation/removal of internal entities.
None	`none`	The creation/removal of external readers or writers doesn't trigger the creation/removal of internal entities.

Warning: When the *DDS Router* creates internal entities triggered by the discovery of a writer (i.e. the `discovery-trigger` is either `writer` or `any`), the *DDS Router* will create its internal entities with the writer's *Topic QoS*, and, therefore, the QoS of the communication between the external entities and the *DDS Router* may differ from the QoS of the communication without the *DDS Router*.

Warning: At the time being, *remove-unused-entities* set to `true` is only compatible with a `discovery-trigger: reader`.

QoS

`specs` supports a `qos` **optional** tag to configure the default values of the *Topic QoS*.

Note: The *Topic QoS* configured in `specs` can be overwritten by the *Participant Topic QoS* and the *Manual Topics*.

Logging

`specs` supports a `logging` **optional** tag to configure the *DDS Router* logs. Under the `logging` tag, users can configure the type of logs to display and filter the logs based on their content and category. When configuring the verbosity to `info`, all types of logs, including informational messages, warnings, and errors, will be displayed. Conversely, setting it to `warning` will only show warnings and errors, while choosing `error` will exclusively display errors. By default, the filter allows all errors to be displayed, while selectively permitting warning and informational messages from `DDSROUTER` category.

```yaml
logging:
  verbosity: info
  filter:
    error: "DDSPIPE|DDSROUTER"
    warning: "DDSPIPE|DDSROUTER"
    info: "DDSROUTER"
```

Note: Configuring the logs via the Command-Line is still active and takes precedence over YAML configuration when both methods are used simultaneously.

Logging	Yaml tag	Description	Data type	Default value	Possible values
Verbosity	`verbosity`	Show messages of equal or higher importance.	*enum*	`error`	`info` / `warning` / `error`
Filter	`filter`	Regex to filter the category or message of the logs.	*string*	info : `DDSROUTER` warning : `DDSROUTER` error : `""`	Regex string

Note: For the logs to function properly, the `-DLOG_INFO=ON` compilation flag is required.

The *DDS Router* prints the logs by default (warnings and errors in the standard error and infos in the standard output). The *DDS Router*, however, can also publish the logs in a DDS topic. To publish the logs, under the tag `publish`, set `enable: true` and set a `domain` and a `topic-name`. The type of the logs published is defined as follows:

LogEntry.idl

```
const long UNDEFINED = 0x10000000;
const long SAMPLE_LOST = 0x10000001;
const long TOPIC_MISMATCH_TYPE = 0x10000002;
const long TOPIC_MISMATCH_QOS = 0x10000003;

enum Kind {
  Info,
  Warning,
  Error
};

struct LogEntry {
  @key long event;
  Kind kind;
  string category;
  string message;
  string timestamp;
};
```

Note: The type of the logs can be published by setting `publish-type: true`.

Example of usage

```
logging:
  verbosity: info
  filter:
    error: "DDSPIPE|DDSROUTER"
    warning: "DDSPIPE|DDSROUTER"
    info: "DDSROUTER"
  publish:
    enable: true
    domain: 84
    topic-name: "DdsRouterLogs"
    publish-type: false
  stdout: true
```

Monitor

`specs` supports a `monitor` **optional** tag to publish internal data from the *DDS Router*. If the monitor is enabled, it publishes (and logs under the `MONITOR_DATA` *log filter*) the *DDS Router's* internal data on a `domain`, under a `topic-name`, once every `period` (in milliseconds). If the monitor is not enabled, the *DDS Router* will not collect or publish any data.

Note: The data published is relative to each period. The *DDS Router* will reset its tracked data after publishing it.

In particular, the *DDS Router* will track the number of messages lost, received, and the message reception rate [Hz] of each topic. The type of the data published is defined as follows:

MonitoringTopics.idl

```
struct DdsTopicData
{
    string participant_id;
    unsigned long msgs_lost;
    unsigned long msgs_received;
    double msg_rx_rate;
};

struct DdsTopic
{
    string name;
    string type_name;
    boolean type_discovered;
    boolean type_mismatch;
    boolean qos_mismatch;
    sequence<DdsTopicData> data;
};

struct MonitoringTopics
{
    sequence<DdsTopic> topics;
};
```

Example of usage

```
monitor:
  topics:
    enable: true
    period: 1000
    domain: 10
    topic-name: "DdsRouterTopicData"
```

3.12.5 Participant Configuration

At the yaml base level, along with the `builtin-topics` tag, there will be the tag `participants` to handle an array of *Participant* configurations. Each Participant is identified by a unique *Participant Name* and requires to set the `kind` of the Participant. There can be any number of Participants, and Participant kinds can be repeated.

Each Participant has its specific configuration. Please, refer to *Participant kinds* in order to see each of the *Participant Kinds* requirements.

> **Warning:** Do not configure two Participants in a way that they can communicate to each other (e.g. two Simple participants in the same domain). This will lead to an infinite feedback loop between each other.

In the following configuration example, the *DDS Router* will create two *Simple Participants*, one for `domain 0` and one for `domain 1`. This is a typical use case of *DDS Domain bridge*. The topics allowed in the two domains will start communicating to each other. Note that the communication is not P2P performed between the end-user DDS entities, i.e. the data must reach the *DDS Router* and this will forward the data.

```yaml
participants:               # Tag to introduce the participants configurations array

################

  - name: Participant0      # Participant Name = Participant0
    kind: local             # Participant Kind = simple
    domain: 0               # DomainId = 0

################

  - name: my_custom_part    # Participant Name = my_custom_part
    kind: simple            # Participant Kind = echo
    domain: 1               # DomainId = 1
```

The first Participant *Participant0* has Participant Name *Participant0* and is configured to be of the *simple* Participant Kind, and to communicate locally in domain 0. The second Participant has Participant Name *simple* and it is configured to be of the *simple* kind and to communicate locally with domain 1.

Domain Id

Tag `domain` configures the *Domain Id* of a specific Participant. Be aware that some Participants (e.g. Discovery Servers) do not need a Domain Id configuration.

```yaml
domain: 101
```

Ignore Participant Flags

A set of discovery traffic filters can be defined for *Simple Participants* in order to add an extra level of isolation. This configuration option can be set through the `ignore-participant-flags` tag:

```yaml
ignore-participant-flags: no_filter                          # No filter (default)
# or
ignore-participant-flags: filter_different_host              # Discovery traffic from␣
↪another host is discarded
# or
ignore-participant-flags: filter_different_process           # Discovery traffic from␣
↪another process on same host is discarded
# or
ignore-participant-flags: filter_same_process                # Discovery traffic from␣
↪own process is discarded
# or
ignore-participant-flags: filter_different_and_same_process  # Discovery traffic from␣
↪own host is discarded
```

See Ignore Participant Flags for more information.

Custom Transport Descriptors

By default, *Simple Participants* are created with enabled UDP and Shared Memory transport descriptors. The use of one or the other for communication will depend on the specific scenario, and whenever both are viable candidates, the most efficient one (Shared Memory Transport) is automatically selected. However, a user may desire to force the use of one of the two, which can be accomplished via the `transport` configuration tag in *Simple Participants*.

```
transport: builtin    # UDP & SHM (default)
# or
transport: udp        # UDP only
# or
transport: shm        # SHM only
```

Warning: Participants configured with `transport: shm` will only communicate with applications using Shared Memory Transport exclusively (with disabled UDP transport).

Interface Whitelist

Optional tag `whitelist-interfaces` allows to limit the network interfaces used by UDP and TCP transport. This may be useful to only allow communication within the host (note: same can be done with *Ignore Participant Flags*), or in the WAN scenario one may choose to only communicate through the Ethernet or WiFi interface (when both available). Example:

```
whitelist-interfaces:
  - "127.0.0.1"    # Localhost only
```

See Interface Whitelist for more information.

Repeater Participant

The optional tag `repeater` configures a *WAN Participant* as a *Repeater* point. This means that this Participant will forward all the information received from its Readers to its Writers.

Check the *Repeater DDS Router* use case to see how the `repeater` Participant attribute is used.

```
repeater: true
```

Note: This tag is only supported in configuration versions above v2.0.

Network Address

Network Addresses are elements that can be configured for specific Participants. An Address is defined by:

- *IP*: IP of the host (public IP in case of WAN communication).
- *Port*: Port where the Participant is listening.
- *External Port*: Public port accessible for external entities (only for TCP).
- *Transport Protocol*: `UDP` or `TCP`. If it is not set, it would be chosen by default depending on the Participant Kind.

- *IP version*: `v4` or `v6`. If it is not set, it would be chosen depending on the *IP* string format.
- *Domain Name*: Domain's unique name to ask the DNS server for the related IP. This field is ignored if `ip` is specified.

```
ip: 127.0.0.1
port: 11666
external-port: 11670
transport: tcp
ip-version: v4

################

ip: 2001:4860:4860::8844      # Recognized as IPv6
port: 1616
transport: udp

################

domain: localhost             # DNS call return value = 127.0.0.1
port: 33333                   # Uses UDP by default
```

External Port

External port is used to configure a Server (Discovery Server or Initial Peers) that runs under a NAT and uses TCP transport. This value could be set in a TCP listening address to differentiate the public and the internal port. **In case this value is not set, the external port is considered to be the same as the internal one.** If both ports coincide, external and internal port in the network router port forwarding rules must coincide. For more information, check section *External port*.

Discovery Server GuidPrefix

A *Discovery Server* requires a DDS *GuidPrefix* in order for other Participants to connect to it. Under the `discovery-server-guid` tag, there are several possibilities for configuring a GuidPrefix.

Discovery Server GuidPrefix by string

The GuidPrefix of the Discovery Server can be configured using `guid` tag. Be aware of using the correct format for GuidPrefix. That is, 12 hexadecimal numbers (lower than `ff`) separated with `.`.

```
discovery-server-guid:
  guid: "1.f.1.0.0.0.0.0.0.0.ca.fe"       # GuidPrefix = 01.0f.01.00.00.00.00.00.00.00.
↪ca.fe
```

Discovery Server GuidPrefix by Id

Using tag `id`, the GuidPrefix will be calculated arbitrarily using a default *DDS Router* GuidPrefix. This default GuidPrefix is `01.0f.<id>.00.00.00.00.00.00.00.ca.fe`. Default value for `id` is `0`. This entry is ignored if `guid` is specified.

```
discovery-server-guid:
  id: 13                                  # GuidPrefix = 01.0f.0d.00.00.00.00.00.00.00.
↪ca.fe
```

Note: In the current version of the *DDS Router* only ids in the range 0 to 256 are allowed. In future releases it would be implemented to allow a wider range of ids.

ROS Discovery Server GuidPrefix

There is a specific GuidPrefix for ROS 2 executions which can be used with Fast DDS CLI and ROS 2 `ROS_DISCOVERY_SERVER` environment variable (https://fast-dds.docs.eprosima.com/en/v2.4.1/fastdds/ros2/discovery_server/ros2_discovery_server.html).

The ROS 2 Discovery Server GuidPrefix is set by default to `44.53.<id>.5f.45.50.52.4f.53.49.4d.41` where `<id>` is the specific id of the Server. This GuidPrefix also allows an `id` value to specify which id is used in the GuidPrefix. The default value for `id` is `0`.

```
discovery-server-guid:
  ros-discovery-server: true              # GuidPrefix = 44.53.x.5f.45.50.52.4f.53.49.4d.
↪41
  id: 13                                  # GuidPrefix = 44.53.0d.5f.45.50.52.4f.53.49.
↪4d.41
```

Listening Addresses

Tag `listening-addresses` configures the network addresses where this Participant is going to listen for remote Participants. `listening-addresses` is *key* for an array of *Network Addresses*.

```
listening-addresses:
  - ip: 127.0.0.1                 # UDP by default
    port: 11667
  - ip: 2001:4860:4860::8844      # Recognized as IPv6
    port: 11666
    external-port: 11668
    transport: tcp
```

Initial Peers Connection Addresses

Tag `connection-addresses` configure a connection with one or multiple remote WAN Participants. `connection-addresses` is *key* for an array of *Network Addresses*.

```
connection-addresses:
  - ip: 127.0.0.1
    port: 11666
  - ip: 2001:4860:4860::8844
    port: 11668
    transport: tcp
```

Discovery Server Connection Addresses

Tag `connection-addresses` configure a connection with one or multiple remote Discovery Servers. `connection-addresses` is the *key* for an array in which each element has a GuidPrefix referencing the Discovery Server to connect with; and a tag `addresses` configuring the addresses of such Discovery Server. Each element inside `addresses` must follow the configuration for *Network Address*.

```
connection-addresses:
  - discovery-server-guid:
      guid: 44.53.0d.5f.45.50.52.4f.53.49.4d.41
    addresses:
      - ip: 127.0.0.1
        port: 11666
  - discovery-server-guid:
      id: 4
    addresses:
      - ip: 2001:4860:4860::8888
        port: 11667
        transport: tcp
      - ip: 2001:4860:4860::8844
        port: 11668
        transport: tcp
```

Profile

Tag `profile` set the *QoS Profile* to create a specific Participant. This profile must match with an existent profile loaded by XML. It will use such profile for configuring the Participant.

```
profile: participant_custom_configuration
```

QoS

Participants support a `qos` **optional** tag to manually configure their *Topic QoS*.

Note: The *Topic QoS* configured for a Participant can be overwritten by the *Manual Topics* but take precedence over the *Specs Topic QoS*.

3.12.6 Forwarding Routes

The *DDS Router* is capable of establishing different internal routes between its participants. This feature enables users to only forward sensitive data to a set of participants.

Note: By default, when the tag `routes` is not set, every participant forwards the data it receives to every other participant.

Generic Routes

To configure a custom set of forwarding routes, use the tag `routes` followed by the destination participants for each source participant.

Note: If a participant is not listed as a source, it will forward the data it receives to every other participant.

Note: If a participant is listed as a source but it is not given any destination participants, it will not forward the data it receives to any participant.

Consider the following example with three participants: `Participant0`, `Participant1`, and `Participant2`.

```
routes:
  - src: Participant0
    dst:
      - Participant2

  - src: Participant1
```

- Participant `Participant0` will only forward the data it receives to participant `Participant2`.
- Participant `Participant1` will not forward the data it receives to any participant, since it does not have any destination participants.
- Participant `Participant2` will forward the data it receives to every participant (`Participant0` and `Participant1`), since it does not have a forwarding route.

Warning: A repeater participant with a route defined must add itself to its route's destinations.

Topic Routes

Besides the generic routes just described, custom routes can also be configured for a specific topic (determined by a `name` and `type` pair). To configure a custom set of forwarding routes for a specific topic, use the tag `topic-routes`.

> **Warning:** Topic routes take precedence over generic routes. This means that when data is published on a topic with a topic route configured, the generic routes are ignored and the topic route is used.

Again, consider the following example with three participants: `Participant0`, `Participant1`, and `Participant2`.

```yaml
topic-routes:
  - name: HelloWorld
    type: HelloWorld
    routes:
      - src: Participant1
        dst:
          - Participant0

      - src: Participant2
```

When data is published in the topic `HelloWorld` with type `HelloWorld`:

- Participant `Participant0` will forward the data it receives to every participant, since it does not have a forwarding route.
- Participant `Participant1` will only forward the data it receives to participant `Participant0`.
- Participant `Participant2` will not forward the data it receives to any participant, since it does not have any destination participants.

3.12.7 General Example

A complete example of all the configurations described on this page can be found below.

```yaml
# Version Latest
version: v4.0

# Specifications
specs:
  threads: 10
  remove-unused-entities: false
  discovery-trigger: reader

  qos:
    history-depth: 1000
    max-tx-rate: 0
    max-rx-rate: 20
    downsampling: 3

  logging:
    verbosity: info
    filter:
      error: "DDSPIPE|DDSROUTER"
```

(continues on next page)

(continued from previous page)

```
      warning: "DDSPIPE|DDSROUTER"
      info: "DDSROUTER"
    publish:
        enable: true
        domain: 84
        topic-name: "DdsRouterLogs"
        publish-type: false
    stdout: true

  monitor:
    topics:
      enable: true
      period: 1000
      domain: 10
      topic-name: "DdsRouterTopicStatistics"

# XML configurations to load
xml:

  # Load this file as Fast DDS XML configuration
  files:
      - "./xml_configuration.xml"

  # Load text as Fast DDS XML configuration
  raw: |
      <?xml version="1.0" encoding="UTF-8" ?>
      <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
          <participant profile_name="custom_participant_configuration">
              <domainId>1</domainId>
              <rtps></rtps>
          </participant>
      </profiles>

# Relay topic rt/chatter and type std_msgs::msg::dds_::String_
# Relay topic HelloWorldTopic and type HelloWorld

builtin-topics:

  - name: rt/chatter
    type: std_msgs::msg::dds_::String_

  - name: HelloWorldTopic
    type: HelloWorld

# Manually configure Topic QoS for a set of participants on a topic

topics:

  - name: "temperature/*"
    type: "temperature/types/*"
    qos:
      max-tx-rate: 15
```

(continues on next page)

(continued from previous page)

```
      downsampling: 2
    participants:
      - Participant0
      - Participant1

# Do not allow ROS2 services

blocklist:
  - name: "rr/*"
  - name: "rq/*"


participants:

####################

# Simple DDS Participant in domain 3

  - name: Participant0              # Participant Name = Participant0

    kind: local                     # Participant Kind = local (= simple)

    domain: 3                       # DomainId = 3

    qos:

      max-rx-rate: 0                  # Max Reception Rate = 0 (unlimited)

      downsampling: 1                 # Downsampling = 1

####################

# Simple DDS Participant in domain 7

  - name: Participant1              # Participant Name = Participant1

    kind: local                     # Participant Kind = local (= simple)

    domain: 7                       # DomainId = 7

    qos:

      max-rx-rate: 15                 # Max Reception Rate = 15

####################

# Discovery Server DDS Participant with ROS GuidPrefix so a local ROS 2 Client could␣
↪connect to it
# This Discovery Server will listen in ports 11600 and 11601 in localhost

  - name: ServerROS2                # Participant Name = ServerROS2
```

(continues on next page)

(continued from previous page)

```
    kind: local-discovery-server    # Participant Kind = local-discovery-server

    discovery-server-guid:
      id: 1
      ros-discovery-server: true    # ROS Discovery Server id => GuidPrefix = 44.53.01.
↪5f.45.50.52.4f.53.49.4d.41

    listening-addresses:            # Local Discovery Server Listening Addresses
      - ip: 127.0.0.1               # IP = localhost ; Transport = UDP (by default)
        port: 11600                 # Port = 11600
      - ip: 127.0.0.1               # IP = localhost
        port: 11601                 # Port = 11601
        external-port: 11602        # External Port = 11602
        transport: tcp              # Transport = TCP

    connection-addresses:
      - discovery-server-guid:
          id: 2
          ros-discovery-server: true
        addresses:
          - domain: "localhost"
            port: 22000

####################

# Participant that will communicate with a DDS Router in a different LAN.
# This Participant will work as the remote DDS Router Client, so it sets the connection␣
↪address of the remote one.

  - name: Wan                       # Participant Name = Wan

    kind: wan-ds                     # Participant Kind = Discovery Server WAN

    discovery-server-guid:
      id: 2                         # Internal WAN Discovery Server id => GuidPrefix =␣
↪01.0f.02.00.00.00.00.00.00.00.ca.fe

    connection-addresses:           # WAN Discovery Server Connection Addresses
      - discovery-server-guid:
          id: 4                     # External WAN Discovery Server id => GuidPrefix =␣
↪01.0f.04.00.00.00.00.00.00.00.ca.fe
        addresses:
          - ip: 8.8.8.8             # IP = 8.8.8.8
            port: 11666             # Port = 11666
            transport: udp          # Transport = UDP


####################

# Participant that will use a user set configuration via QoS Profile.

  - name: xml_participant                       # Participant Name = xml_participant
```

(continues on next page)

(continued from previous page)

```
    kind: xml

    profile: custom_participant_configuration   # Configure participant with this profile

# Custom generic forwarding route.

routes:

  - src: Participant0
    dst:
      - Participant1

# Custom topic forwarding route.

topic-routes:

  - name: HelloWorld
    type: HelloWorld
    routes:
      - src: Participant1
        dst:
          - Participant0
```

3.13 WAN Configuration

In order to communicate a *DDS Router* via *WAN*, some configurations may be required.

3.13.1 NAT Traversal

If the *DDS Router* is under a *NAT*, a remote *DDS Router* in a different *LAN* will not be able to reach it. Thus, NAT traversal methods will be required. The most common method that we recommend is configuring the network router so it forwards a specific port from the Internet to a specific host.

Note: NAT Traversal communication only affects to IPv4 communication. Using IPv6 would not create NAT under network routers so every device could be accessed externally. Thus, configurations explained in this section do not apply to IPv6 deployments.

Port Forwarding

This is the easiest way to achieve NAT traversal. Most network routers support a graphical interface where port forwarding could be easily set.

External port

In order to configure the *DDS Router* to connect under a NAT, two ports must be taken into account. The internal port (a.k.a. `port`) is the one that the host of the *DDS Router* will use to open a socket and to receive information. The external port (`external-port`) references the public port meant for other entities to be able to locate this *DDS Router*. Setting the external port is useful so the network router port forwarding could redirect from a public port to a different value of internal host port.

Note: External port configuration is not mandatory. If not set the internal and the external port must coincide in the network router port forwarding rules.

Warning: External port is only available for TCP communication. In UDP communication the internal and the external port must coincide in the network router port forwarding rules.

3.13.2 TCP vs UDP

TCP and *UDP* are two well known network transport protocols. Both have their advantages and disadvantages depending on the scenario. The following comparison is meant to help users choose between one or the other.

Feature	UDP	TCP
Communication speed	Faster.	Slower.
Reliability	No Transport Layer reliability (but it can have DDS reliability).	Transport Layer reliability (duplicated if DDS reliability is enabled).
Port Forwarding	Both sides of the communication must have their ports forwarded from the router. Internal and external ports must coincide.	Only the server side of the communication must have its ports forwarded from the router.

Note: DDS is thought to work over UDP and has its own reliability mechanisms. Thus, the *DDS Router* uses UDP transport by default for every address that has not explicitly specified a transport in the configuration file.

3.13.3 TLS

eProsima DDS Router also supports TLS over TCP, and its configuration can be set per participant for types WAN Discovery Server and WAN. The following is a list of the accepted entries under the `tls` tag:

Tag	Requirements	Description	Example
`ca`	Mandatory for TLS clients if `peer_verification` is active.	Path to the CA (Certification-Authority) file.	`ca.crt`
`password`	Optional for TLS servers.	Password of the `private_key` file.	`<private_key_file_password>`
`private_key`	Mandatory for TLS servers.	Path to the private key certificate file.	`ddsrouter.key`
`cert`	Mandatory for TLS servers.	Path to the public certificate chain file.	`ddsrouter.crt`
`dh_params`	Mandatory for TLS servers.	Path to the Diffie-Hellman parameters file.	`dh_params.pem`
`peer_verification`	Optional for clients.	Whether to verify the server. (Default true).	`true`
`sni_host`	Optional for clients if using SNI.	Name of the server to connect with.	`my_server.com`

Note: Although in principle only required for TLS clients (with peer verification), the CA (Certification- Authority) file may also be provided for TLS servers when willing to connect them to other participants configured as servers.

3.13.4 Examples

TCP Port Forwarding Example

Let there be a scenario where user *A* with host H_A has a private IP `192.168.1.2` given by the network router R_A with public IP `1.1.1.1`. Let user *B* with host H_B have a private IP `192.168.2.2` given by the network router R_B with public IP `2.2.2.2`. *A* will act as the server of the TCP communication and *B* will act as the client.

User *A* should set a port forwarding rule in router R_A as `11666 -> 192.168.1.2:11667`. That is, every datagram that arrives to IP `1.1.1.1:11666` will be forwarded to `192.168.1.2:11667`. User *A* should set its *listening-addresses* as follows:

```
- name: WANServerParticipant_userA
  kind: wan

  listening-addresses:
    - ip: 1.1.1.1                    # Public IP of host Ha
      port: 11667                    # Physical port used for the dds router host
      external-port: 11666           # Port forwarded router Ra
      transport: tcp                 # Transport protocol
```

User *B* should set *connection-addresses* to connect to H_A as follows:

```
- name: WANClientParticipant_userB
  kind: wan
```

(continues on next page)

(continued from previous page)

```
  connection-addresses:
    - ip: 1.1.1.1                    # Public IP of Ha
      port: 11666                    # Port forwarded in Ra
      transport: tcp                 # Transport protocol
```

This way, B will connect to A. A will be able to receive the message because R_A will forward the message to H_A. Once A has received the message, a TCP channel will be set, and the communication will travel both ways without requiring to traverse any other NAT.

UDP Port Forwarding Example

Let there be a scenario where user A with host H_A has a private IP `192.168.1.2` given by the network router R_A with public IP `1.1.1.1`. Let user B with host H_B have a private IP `192.168.2.2` given by the network router R_B with public IP `2.2.2.2`. A and B will communicate via UDP, so there is no need to set a client and a server. It does not matter whether A knows B address, B knows A, or both know each other. In this example, B will know A address, and not the other way around.

User A should set a port forwarding rule in router R_A as `11666 -> 192.168.1.2:11666`. That is, every datagram that arrives to IP `1.1.1.1:11666` will be forwarded to `192.168.1.2:11666`. User A should set its *listening-addresses* as follows:

```
- name: WANServerParticipant_userA
  kind: wan

  listening-addresses:
    - ip: 1.1.1.1                    # Public IP of host Ha
      port: 11666                    # Internal and External port
```

User B should set a port forwarding rule in router R_B as `11777 -> 192.168.2.2:11777`. This is, every datagram that arrives to IP `2.2.2.2:11777` will be forwarded to `192.168.2.2:11777`. User B should set its *listening-addresses* and *connection-addresses* as follows:

```
- name: WANClientParticipant_userB
  kind: wan

  listening-addresses:
    - ip: 2.2.2.2                    # Public IP of host Hb
      port: 11777                    # Internal and External port
  connection-addresses:
    - ip: 1.1.1.1                    # Public IP of Ha
      port: 11666                    # Port forwarded in Ra
```

This way, B will connect to A. Once A receives the message from B, it will communicate with it via address `2.2.2.2:11777`. B will continue communicating with A via address `1.1.1.1:11666`.

TLS Configuration Example

Below is an example on how to configure a WAN participant as a TLS server and client:

```
- name: TLS_Server
  kind: wan

  listening-addresses:
    - ip: 1.1.1.1
      port: 11666
      transport: tcp

  tls:
    ca: ca.crt
    password: ddsrouterpass
    private_key: ddsrouter.key
    cert: ddsrouter.crt
    dh_params: dh_params.pem
```

```
- name: TLS_Client
  kind: wan

  connection-addresses:
    - ip: 1.1.1.1
      port: 11666
      transport: tcp

  tls:
    ca: ca.crt
```

You may also have a look at the <path/to/ddsrouter_tool>/share/resources/configurations/security/ directory, which contains examples of key and certificate files as well as a script with the commands used to generate them.

3.14 YAML Validator

Configuration files used to launch a DDS-Router instance need to follow a specific structure, which is extensively described along section *Configuration*. The *YAML Validator tool* has been developed for the sole purpose of validating user-defined configuration files in an easy manner.

Note: Yaml Validator tool is supported for `3.0` configuration version only.

After having sourced the *DDS Router* workspace, execute the following command in order to validate a YAML configuration file:

```
ddsrouter_yaml_validator --config-file ddsrouter-config.yaml
```

Alternatively, the user may choose to validate against a different schema, by using instead the command below:

```
ddsrouter_yaml_validator --config-file ddsrouter-config.yaml --schema schema.json
```

3.15 Nomenclature

3.15.1 DDS Router nomenclature

Payload Raw data (no format specified) that is received and sent forward from the DDS Router.

Participant nomenclature

Participant DDS Router communication Interface. It is an abstraction of DDS DomainParticipant.

This term is explained *here*.

Participant Name Unique identifier of a Participant.

This term is explained *here*.

Participant Kind Element that identifies a Participant kind. It can be set as a *string* that references an alias of an existing Participant Kind.

This term is explained *here*.

3.15.2 DDS nomenclature

DataReader DDS element that subscribes to a specific Topic. It belongs to one and only one Participant, and it is uniquely identified by a Guid.

See Fast DDS documentation for further information.

DataWriter DDS entity that publishes data in a specific Topic. It belongs to one and only one Participant, and it is uniquely identified by a Guid.

See Fast DDS documentation for further information.

Discovery Server Discovery Server Discovery Protocol is a Fast DDS feature that enables a new Discovery mechanism based on a Server that filters and distribute the discovery information. This is highly recommended in networks where multicast is not available (e.g. WAN).

See Fast DDS documentation for further information.

Domain Id The Domain Id is a virtual partition for DDS networks. Only DomainParticipants with the same Domain Id would be able to communicate to each other. DomainParticipants in different Domains will not even discover each other.

See Fast DDS documentation for further information.

DomainParticipant A DomainParticipant is the entry point of the application to a DDS Domain. Every DomainParticipant is linked to a single domain from its creation, and cannot change such domain. It also acts as a factory for Publisher, Subscriber and Topic.

See Fast DDS documentation for further information.

Endpoint DDS element that publish or subscribes in a specific Topic. Endpoint kinds are *DataWriter* or *DataReader*.

Guid Global Unique Identifier. It contains a GuidPrefix and an EntityId. The EntityId uniquely identifies sub-entities inside a Participant. Identifies uniquely a DDS entity (DomainParticipant, DataWriter or DataReader).

GuidPrefix Global Unique Identifier shared by a Participant and all its sub-entities. Identifies uniquely a DDS Participant.

Initial Peers It is a Fast DDS Discovery Protocol that allows the router to send the discovery information directly to the Participants configured. This is highly recommended for static networks where multicast is not available (e.g. WAN).

See Fast DDS documentation for further information.

Topic DDS isolation abstraction to encapsulate subscriptions and publications. Each Topic is uniquely identified by a topic name and a topic type name (name of the data type it transmits).

See Fast DDS documentation for further information.

3.16 Echo Example

The following YAML configuration file configures a DDS Router to create a *Simple Participant* in *Domain Id* `0` and an *Echo Participant* that will print in `stdout` every message get in Domain `0`, as well as information regarding discovery events.

```yaml
###################################
# CONFIGURATION VERSION
version: v4.0                                                    # 0

###################################
# ALLOWED TOPICS
# Allowing FastDDS and ROS2 HelloWorld demo examples topics

allowlist:
  - name: HelloWorldTopic                                        # 1
    type: HelloWorld                                             # 1
  - name: rt/chatter                                             # 2
    type: std_msgs::msg::dds_::String_

###################################
# PARTICIPANTS
participants:

###################################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in allowlist in specific domain and listen␣
↪every message published there

  - name: SimpleParticipant                                      # 3
    kind: local                                                  # 4
    domain: 0                                                    # 5

###################################
# ECHO PARTICIPANT
# This Participant will print in stdout every message received by the other Participants,
↪ as well as discovery information

  - name: EchoParticipant                                        # 6
    kind: echo                                                   # 7
    discovery: true                                              # 8
    data: true                                                   # 9
```

(continues on next page)

(continued from previous page)

```
    verbose: true                                                    # 10
```

3.16.1 Configuration

Allowed Topics

This section lists the *Topics* that the DDS Router will route from one Participant to the other. Topic `HelloWorldTopic` with datatype `HelloWorld`, and ROS 2 topic `rt/chatter` with datatype `std_msgs::msg::dds_::String_` will be forwarded from `SimpleParticipant` to `EchoParticipant`, that will print the message in `stdout`.

```
allowlist:
  - name: HelloWorldTopic                                            # 1
    type: HelloWorld                                                 # 1
  - name: rt/chatter                                                 # 2
    type: std_msgs::msg::dds_::String_
```

Simple Participant

This Participant is configured with a name, a kind and the Domain Id, which is `0` in this case.

```
  - name: SimpleParticipant                                          # 3
    kind: local                                                      # 4
    domain: 0                                                        # 5
```

Echo Participant

This Participant is configured to display information regarding messages received, as well as discovery events. See *Echo Participant Configuration* for more details.

```
  - name: EchoParticipant                                            # 6
    kind: echo                                                       # 7
    discovery: true                                                  # 8
    data: true                                                       # 9
    verbose: true                                                    # 10
```

3.16.2 Execute example

For a detailed explanation on how to execute the *DDS Router*, refer to this *section*.

Note: Internal entities for a specific topic are only created once a data receiver (Reader/Subscriber) is discovered. Hence, for these example to work, either substitute `allowlist` for *builtin-topics* in the configuration file, or launch a subscriber/listener in the same domain (`0`).

Execute with Fast DDS HelloWorld Example

Execute a Fast DDS HelloWorld example:

```
./DDSHelloWorldExample publisher
```

Execute *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/echo.yaml`). The expected output from the DDS Router, printed by the `Echo Participant` is:

```
New endpoint discovered: Endpoint{01.0f.b8.d9.81.30.3d.a7.01.00.00.00|0.0.1.3;writer;
↪DdsTopic{HelloWorldTopic;HelloWorld;Fuzzy{Level(20) TopicQoS{TRANSIENT_LOCAL;RELIABLE;
↪SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};Active;
↪ParticipantId{SimpleParticipant}}.
In Endpoint: 01.0f.b8.d9.81.30.3d.a7.01.00.00.00|0.0.1.3 from Participant: ParticipantId
↪{SimpleParticipant} in topic: DdsTopic{HelloWorldTopic;HelloWorld;Fuzzy{Level(20)␣
↪TopicQoS{VOLATILE;BEST_EFFORT;SHARED;depth(5000)}}} payload received: Payload{00 01 00␣
↪00 01 00 00 00 0b 00 00 00 48 65 6c 6c 6f 57 6f 72 6c 64 00 00} with specific qos:␣
↪SpecificEndpointQoS{Partitions{};OwnershipStrength{0}}.
In Endpoint: 01.0f.b8.d9.81.30.3d.a7.01.00.00.00|0.0.1.3 from Participant: ParticipantId
↪{SimpleParticipant} in topic: DdsTopic{HelloWorldTopic;HelloWorld;Fuzzy{Level(20)␣
↪TopicQoS{VOLATILE;BEST_EFFORT;SHARED;depth(5000)}}} payload received: Payload{00 01 00␣
↪00 02 00 00 00 0b 00 00 00 48 65 6c 6c 6f 57 6f 72 6c 64 00 00} with specific qos:␣
↪SpecificEndpointQoS{Partitions{};OwnershipStrength{0}}.
...
In Endpoint: 01.0f.b8.d9.81.30.3d.a7.01.00.00.00|0.0.1.3 from Participant: ParticipantId
↪{SimpleParticipant} in topic: DdsTopic{HelloWorldTopic;HelloWorld;Fuzzy{Level(20)␣
↪TopicQoS{VOLATILE;BEST_EFFORT;SHARED;depth(5000)}}} payload received: Payload{00 01 00␣
↪00 0a 00 00 00 0b 00 00 00 48 65 6c 6c 6f 57 6f 72 6c 64 00 00} with specific qos:␣
↪SpecificEndpointQoS{Partitions{};OwnershipStrength{0}}.
```

Execute with ROS 2 demo nodes

Execute a ROS 2 `demo_nodes_cpp` *talker* in default domain `0`:

```
ros2 run demo_nodes_cpp talker
```

Execute *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/echo.yaml`). The expected output from the DDS Router, printed by the `Echo Participant` is:

```
New endpoint discovered: Endpoint{01.0f.b8.d9.b6.3a.7d.95.01.00.00.00|0.0.1.3;writer;
↪DdsTopic{ros_discovery_info;rmw_dds_common::msg::dds_::ParticipantEntitiesInfo_;Fuzzy
↪{Level(20) TopicQoS{TRANSIENT_LOCAL;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS
↪{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant}}.
New endpoint discovered: Endpoint{01.0f.b8.d9.b6.3a.7d.95.01.00.00.00|0.0.2.4;reader;
↪DdsTopic{ros_discovery_info;rmw_dds_common::msg::dds_::ParticipantEntitiesInfo_;Fuzzy
↪{Level(20) TopicQoS{TRANSIENT_LOCAL;RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS
↪{Partitions{};OwnershipStrength{0}};Active;ParticipantId{SimpleParticipant}}.
...
New endpoint discovered: Endpoint{01.0f.b8.d9.b6.3a.7d.95.01.00.00.00|0.0.12.3;writer;
↪DdsTopic{rt/chatter;std_msgs::msg::dds_::String_;Fuzzy{Level(20) TopicQoS{VOLATILE;
↪RELIABLE;SHARED;depth(5000)}}};SpecificEndpointQoS{Partitions{};OwnershipStrength{0}};
↪Active;ParticipantId{SimpleParticipant}}.
```

(continues on next page)

(continued from previous page)

```
In Endpoint: 01.0f.b8.d9.b6.3a.7d.95.01.00.00.00|0.0.12.3 from Participant: ParticipantId
↪{SimpleParticipant} in topic: DdsTopic{rt/chatter;std_msgs::msg::dds_::String_;Fuzzy
↪{Level(0) TopicQoS{VOLATILE;BEST_EFFORT;SHARED;depth(1000)}}} payload received: Payload
↪{00 01 00 00 0f 00 00 00 48 65 6c 6c 6f 20 57 6f 72 6c 64 3a 20 31 00 00} with␣
↪specific qos: SpecificEndpointQoS{Partitions{};OwnershipStrength{0}}.
In Endpoint: 01.0f.b8.d9.b6.3a.7d.95.01.00.00.00|0.0.12.3 from Participant: ParticipantId
↪{SimpleParticipant} in topic: DdsTopic{rt/chatter;std_msgs::msg::dds_::String_;Fuzzy
↪{Level(0) TopicQoS{VOLATILE;BEST_EFFORT;SHARED;depth(1000)}}} payload received: Payload
↪{00 01 00 00 0f 00 00 00 48 65 6c 6c 6f 20 57 6f 72 6c 64 3a 20 32 00 00} with␣
↪specific qos: SpecificEndpointQoS{Partitions{};OwnershipStrength{0}}.
...
```

3.17 Change Domain Example

The following YAML configuration file configures a DDS Router to create two *Simple Participants*, one in domain `0` and another in domain `1`.

```yaml
###################################
# CONFIGURATION VERSION
version: v4.0                                                    # 0

###################################
# ALLOWED TOPICS
# Allowing FastDDS and ROS2 HelloWorld demo examples topics

allowlist:
  - name: HelloWorldTopic                                        # 1
    type: HelloWorld                                             # 1
  - name: rt/chatter                                             # 2
    type: std_msgs::msg::dds_::String_

###################################
# PARTICIPANTS
participants:

###################################
# SIMPLE PARTICIPANT DOMAIN 0
# This participant subscribes to allowlist topics in DDS Domain 0 and listen every␣
↪message published in such DDS Domain

  - name: SimpleParticipant_domain0                              # 3
    kind: local                                                  # 4
    domain: 0                                                    # 5

###################################
# SIMPLE PARTICIPANT DOMAIN 1
# This participant subscribes to allowlist topics in DDS Domain 1 and listen every␣
↪message published in such DDS Domain

  - name: SimpleParticipant_domain1                              # 6
```

(continues on next page)

(continued from previous page)

```
    kind: local                                                    # 7
    domain: 1                                                      # 8
```

3.17.1 Configuration

Allowed Topics

This section lists the *Topics* that the DDS Router will route from one Participant to the other. Topic `HelloWorldTopic` with datatype `HelloWorld`, and ROS 2 topic `rt/chatter` with datatype `std_msgs::msg::dds_::String_` will be forwarded from one domain to the other, allowing different DDS domains to interact with each other.

```
allowlist:
  - name: HelloWorldTopic                                          # 1
    type: HelloWorld                                               # 1
  - name: rt/chatter                                               # 2
    type: std_msgs::msg::dds_::String_
```

Simple Participant Domain 0

This Participant is configured with a name, a kind, and a Domain Id, which is `0` in this case.

```
  - name: SimpleParticipant_domain0                                # 3
    kind: local                                                    # 4
    domain: 0                                                      # 5
```

Simple Participant Domain 1

This Participant is configured with a name, a kind, and a Domain Id, which is `1` in this case.

```
  - name: SimpleParticipant_domain1                                # 6
    kind: local                                                    # 7
    domain: 1                                                      # 8
```

3.17.2 Execute example

Please refer to section *User Interface* for a detailed explanation on how to execute the *DDS Router*.

Execute with ROS 2 demo nodes

Execute a ROS 2 `demo_nodes_cpp` *talker* in domain `0`:

```
ROS_DOMAIN_ID=0 ros2 run demo_nodes_cpp talker
```

Execute a ROS 2 `demo_nodes_cpp` *listener* in domain 1:

```
ROS_DOMAIN_ID=1 ros2 run demo_nodes_cpp listener
```

Execute the *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/change_domain_allowlist.yaml`). Once the *DDS Router* is running, messages from *talker* in domain 0 will be forwarded by the Router to the *listener* in domain 1 that will print them in `stdout`.

There is also available an example without `allowlist` (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/change_domain.yaml`). In this case, the topics of the DDS network are dynamically discovered by the DDS Router. Since there is no allowlist, the data from all the topics found are forwarded.

3.18 ROS2 Discovery Server Example

In the following snippet we see a yaml file to configure a DDS Router to create a *Simple Participant* in domain `0` and a *Local Discovery Server* with ROS 2 configuration.

```yaml
#####################################
# CONFIGURATION VERSION
version: v4.0                                   # 0

#####################################
# ALLOWED TOPICS
# Allowing ROS2 HelloWorld demo_nodes topic

allowlist:
  - name: rt/chatter                            # 1
    type: std_msgs::msg::dds_::String_          # 1

#####################################
# PARTICIPANTS
participants:

#####################################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in allowlist in domain 0 and listen every␣
↪message published there

  - name: SimpleROS2                            # 2
    kind: local                                 # 3
    domain: 0                                   # 4

#####################################
# ROS DISCOVERY SERVER
# This participant will subscribe to topics in allowlist using Discovery Server protocol␣
↪as Server

  - name: ServerROS2                            # 5
    kind: local-discovery-server                # 6
    discovery-server-guid:
      ros-discovery-server: true                # 7
      id: 1                                     # 8
    listening-addresses:                        # 9
      - domain: localhost                       # 10
        port: 11888                             # 11
```

3.18.1 Configuration

Allowed Topics

In this section are the *Topics* that the DDS Router will route from one Participant to the other. Topic `HelloWorldTopic` with datatype `HelloWorld`, and ROS 2 topic `rt/chatter` with datatype `std_msgs::msg::dds_::String_` will be forwarded from one domain to the other, allowing different DDS domains to interact to each other.

```
allowlist:
  - name: rt/chatter                          # 1
    type: std_msgs::msg::dds_::String_        # 1
```

Simple Participant

This Participant is configured by a name, a kind and the Domain Id, in this case `0`.

```
  - name: SimpleROS2                          # 2
    kind: local                               # 3
    domain: 0                                 # 4
```

Discovery Server Participant

This Participant is configured by a name, a kind and a listening addresses where Discovery Server will expect metatraffic data from clients.

```
  - name: ServerROS2                          # 5
    kind: local-discovery-server              # 6
    discovery-server-guid:
      ros-discovery-server: true              # 7
      id: 1                                   # 8
    listening-addresses:                      # 9
      - domain: localhost                     # 10
        port: 11888                           # 11
```

3.18.2 Execute example

For a detailed explanation on how to execute the *DDS Router*, refer to this *section*. Execute a ROS 2 `demo_nodes_cpp` *talker* in domain `0`:

```
ROS_DOMAIN_ID=0 ros2 run demo_nodes_cpp talker
```

Execute a ROS 2 `demo_nodes_cpp` *listener* using Discovery Server as Discovery Protocol:

```
ROS_DISCOVERY_SERVER=";127.0.0.1:11888" ros2 run demo_nodes_cpp listener
```

Execute *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/ros_discovery_server.yaml`). Once the *DDS Router* is running, messages from *talker* in domain 0 will be forwarded by the Router to the *listener* using Discovery Server, that will print them in `stdout`.

3.19 WAN Example

In the following snippet we see a yaml file to configure a DDS Router to create a *Simple Participant* in domain `0` and a *WAN Participant*.

```yaml
###################################
# CONFIGURATION VERSION
version: v4.0                                                    # 0

###################################
# ALLOWED TOPICS
# Allowing FastDDS and ROS2 HelloWorld demo examples topics

allowlist:
  - name: HelloWorldTopic                                        # 1
    type: HelloWorld                                             # 1
  - name: rt/chatter                                             # 2
    type: std_msgs::msg::dds_::String_

###################################
# PARTICIPANTS
participants:

###################################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in allowlist in domain 0 and listen every␣
↪message published there

  - name: SimpleParticipant                                      # 3
    kind: local                                                  # 4
    domain: 0                                                    # 5

###################################
# WAN SERVER
# This participant will subscribe to topics in allowlist and connect to clients through␣
↪Initial Peers.

  - name: WANServer                                              # 6
    kind: wan                                                    # 7
    listening-addresses:                                         # 8
      - ip: 1.1.1.1                                              # 9
        port: 11666                                              # 10
        transport: udp                                           # 11
```

3.19.1 Configuration

Allowed Topics

In this section are the *Topics* that the DDS Router will route from one Participant to the other. Topic `HelloWorldTopic` with datatype `HelloWorld`, and ROS 2 topic `rt/chatter` with datatype `std_msgs::msg::dds_::String_` will be forwarded from one domain to the other, allowing different DDS domains to interact to each other.

```yaml
allowlist:
  - name: HelloWorldTopic                                    # 1
    type: HelloWorld                                         # 1
  - name: rt/chatter                                         # 2
    type: std_msgs::msg::dds_::String_
```

Simple Participant

This Participant is configured by a name, a kind and the Domain Id, in this case `0`.

```yaml
  - name: SimpleParticipant                                  # 3
    kind: local                                              # 4
    domain: 0                                                # 5
```

WAN Participant Server

This Participant is configured with a name, a kind and the listening addresses where it will expect data from other remote WAN Participant Clients. This Participant act as a Server only to receive the discovery data from other WAN Participants. Once the connection has been established, the communication will be symmetrical (except in TCP case, in which case this Participant will work as TCP Server).

```yaml
  - name: WANServer                                          # 6
    kind: wan                                                # 7
    listening-addresses:                                     # 8
      - ip: 1.1.1.1                                          # 9
        port: 11666                                          # 10
        transport: udp                                       # 11
```

WAN Participant Client

In order to create a WAN Participant Client, check the configuration file `<path/to/ddsrouter_tool>/share/resources/configurations/examples/wan_client.yaml`

```yaml
  - name: WANClient                                          # 6
    kind: wan                                                # 7
    connection-addresses:                                    # 8
      - ip: 1.1.1.1                                          # 9
        port: 11666
    listening-addresses:                                     # 10
      - ip: 2.2.2.2                                          # 11
        port: 11670                                          # 12
        transport: udp                                       # 13
```

3.19.2 Execute example

In order to run this example, there must be two different hosts located in different local networks:

- host H_A with private IP `192.168.1.2` connected to network router R_A with public IP `1.1.1.1`.
- host H_B with private IP `192.168.2.2` connected to network router R_B with public IP `2.2.2.2`.

This example could be run in localhost or with two hosts in the same LAN, but it will not use the WAN communication features of the *DDS Router*.

Host H_A

This host runs the *DDS Router* WAN Server, which will wait for other WAN Clients to connect to it. Execute *DDS Router* using file `<path/to/ddsrouter_tool>/share/resources/configurations/examples/wan_server.yaml`. Remember to change the IP and port on the configuration file to the actual public IP of R_A, and be sure that the port forwarding rules are configured in R_A so H_A is accessible from the outside. Check the following *section* for further information about how to configure WAN in *DDS Router*. Refer to this *section* for a detailed explanation on how to execute the *DDS Router*.

First of all, execute a ROS 2 `demo_nodes_cpp` *listener* in domain `0`. This listener will discover the Simple Participant in the *DDS Router*, but will not receive any data yet.

```
ROS_DOMAIN_ID=0 ros2 run demo_nodes_cpp listener
```

Host H_B

This host runs the *DDS Router* WAN Client, which will connect to the previously launched WAN Server. Execute *DDS Router* using file `<path/to/ddsrouter_tool>/share/resources/configurations/examples/wan_client.yaml`. Remember to change the IPs and ports on the configuration file to the actual public IPs of R_A and R_B, and be sure that port forwarding is configured in R_B so H_B is accessible from the outside.

In this case, the Simple Participant is configured to use the Domain Id 1, so execute a ROS 2 `demo_nodes_cpp` *talker* in domain 1.

```
ROS_DOMAIN_ID=1 ros2 run demo_nodes_cpp talker
```

Result

After executing both *DDS Router* applications in both hosts, and *talker* and *listener* applications, the *listener* in H_A will start receiving and printing data from the *talker* in H_B. **You are communicating DDS via WAN**.

Remember that the Participants in every *DDS Router* could be configured as any *Participant Kind*, allowing to use local Discovery Server, connect to several domains in the same LAN, connect to several WANs, etc. Endless Possibilities. Just remember uncle Ben's words: *with great power comes great responsibility*.

3.20 Repeater Example

A *DDS Router* could work as a *TURN* **Repeater** to forward messages by the same Participant. If you are interested in understanding the use case of a Repeater please refer to the following section *Repeater DDS Router*.

3.20.1 Configuration

Version

The version attribute is required, as the `repeater` tag is only supported from v4.0 configuration version.

```yaml
version: v4.0                                                    # 0
```

Allowed Topics

This section lists the *Topics* that the DDS Router will route from one Participant to the other. Topic `HelloWorldTopic` with datatype `HelloWorld`, and ROS 2 topic `rt/chatter` with datatype `std_msgs::msg::dds_::String_` will be forwarded from one domain to the other, allowing different DDS domains to interact with each other.

```yaml
allowlist:
  - name: HelloWorldTopic                                        # 1
    type: HelloWorld                                             # 1
  - name: rt/chatter                                             # 2
    type: std_msgs::msg::dds_::String_
```

Repeater Participant

The **Repeater Participant** is the one that will be used to forward messages by the same Participant. It must be a `WAN` Participant and be configured as so with *WAN configuration*.

```yaml
  - name: RepeaterParticipant                                    # 3
    kind: wan                                                    # 4
    repeater: true                                               # 5
    listening-addresses:
      - domain: "server.domain.com"
        port: 11666
        transport: tcp
```

Client Participants

Every *Client* can connect to a Repeater Participant as if it was a normal `WAN` Participant. The Repeater admits as many edge *DDS Router* connections as desired, so several *DDS Router* can use the same or similar configuration to communicate across it.

```yaml
  - name: Client                                                 # 5
    kind: wan                                                    # 6
    connection-addresses:
      - domain: "server.domain.com"
```

(continues on next page)

(continued from previous page)

```
        port: 11666
        transport: tcp
```

3.20.2 Execute example

Please refer to this *section* for a detailed explanation on how to execute the *DDS Router*.

Execute with ROS 2 demo nodes

In order to run this example, there must be three different hosts located in different local networks:

- host H_A with private IP `192.168.1.2` connected to network router R_A with public IP `1.1.1.1`.
- host H_B with private IP `192.168.2.2` connected to network router R_B with public IP `2.2.2.2`.
- host H_C with private IP `192.168.2.3` connected to network router R_C with public IP `3.3.3.3`.

This example could be run in localhost or with two hosts in the same LAN, but it will not use the WAN communication features of the *DDS Router*.

Host H_A

This host runs the *DDS Router* Repeater Server, which will receive data from both edges and redirect the messages between them. Execute *DDS Router* using file `<path/to/ddsrouter_tool>/share/resources/configurations/examples/repeater.yaml`. Remember to change the IP and port on the configuration file to the actual public IP of R_A, and be sure that the port forwarding rules are configured in R_A so H_A is accessible from the outside.

Hosts H_B and H_C

These hosts run the *DDS Router* WAN Clients, which will connect to the previously launched Repeater Server. Execute *DDS Router* using file `<path/to/ddsrouter_tool>/share/resources/configurations/examples/wan_client.yaml`. Remember to change the IPs and ports on the configuration file to the actual public IPs of R_A and R_B. **In this example the port forwarding is not required, as the Repeater will allow the communication through it, and TCP protocol is being used.**.

Both clients can execute ROS 2 demo nodes, which will publish and subscribe in topic `rt/chatter`. Execute a `talker` in one of them and a `listener` in the other. It is recommended to use different `ROS_DOMAIN_ID` in each node, so in case both nodes are accessible (under same network) no loop is created. In order to do so, change the YAML configuration files to use different domains, and use the following ROS2 commands:

```
# Hb
ROS_DOMAIN_ID=0 ros2 run demo_nodes_cpp talker

# Hc
ROS_DOMAIN_ID=1 ros2 run demo_nodes_cpp listener
```

Result

After executing the three *DDS Router* applications in hosts, and *talker* and *listener* applications, the *listener* in H_C will start receiving and printing data from the *talker* in H_B. **You are communicating DDS via WAN**.

Note: If H_B can access H_C due to port forwarding or because both are under the same network, the `listener` will receive duplicated messages, as one of them will arrive from H_B and the other from H_A.

3.21 XML Participant Configuration

The following YAML configuration file configures a DDS Router to create a *Simple Participant* in domain `0`, and another participant configured by *XML* to communicate in domain `1`. The result of executing this configuration is similar to the one in *Change Domain Example*. It generates a bridge between two different domains (`0` & `1`).

```yaml
###################################
# CONFIGURATION VERSION
version: v4.0                                                    # 0

###################################
# CONFIGURATION VERSION
xml:                                                             # 1
  files:
    - "./xml_configuration.xml"                                  # 2
  raw: |                                                         # 3
    <?xml version="1.0" encoding="UTF-8" ?>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <participant profile_name="custom_participant_configuration">
            <domainId>1</domainId>
            <rtps></rtps>
        </participant>
    </profiles>

###################################
# PARTICIPANTS
participants:

###################################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in allowlist in domain 0 and listen every␣
↪message published there

  - name: SimpleParticipant                                      # 6
    kind: local                                                  # 7
    domain: 0                                                    # 8

###################################
# WAN SERVER
# This participant will subscribe to topics in allowlist and connect to clients through␣
↪Initial Peers.
```

(continues on next page)

(continued from previous page)

```yaml
  - name: XMLParticipant                                    # 9
    kind: xml                                               # 10
    profile: custom_participant_configuration               # 11
```

3.21.1 Configuration

Fast DDS XML Configuration

Fast DDS supports loading XML configuration files to load profiles. These profiles are used to configure different DomainParticipants using profile name. Loading an XML file or setting the raw xml file in the *DDS Router* YAML configuration file allows to load such profiles. Here there are the 2 ways to load them. For more information check *following section*.

```yaml
xml:                                                        # 1
  files:
    - "./xml_configuration.xml"                             # 2
  raw: |                                                    # 3
    <?xml version="1.0" encoding="UTF-8" ?>
    <profiles xmlns="http://www.eprosima.com/XMLSchemas/fastRTPS_Profiles" >
        <participant profile_name="custom_participant_configuration">
            <domainId>1</domainId>
            <rtps></rtps>
        </participant>
    </profiles>
```

Simple Participant Domain 0

This Participant is configured with a name, a kind and the Domain Id, which is `0` in this case.

```yaml
  - name: SimpleParticipant                                 # 6
    kind: local                                             # 7
    domain: 0                                               # 8
```

XML Participant Domain 1

This Participant is configured with a name, a kind and the XML profile tag that will be used to configure it. As the XML loaded in this same YAML file configures profile `custom_participant_configuration` as default Participant in domain 1, this Participant will work as a `local` participant in such domain.

```yaml
  - name: XMLParticipant                                    # 9
    kind: xml                                               # 10
    profile: custom_participant_configuration               # 11
```

3.21.2 Execute example

Please refer to this *section* for a detailed explanation on how to execute the *DDS Router*.

Execute with Fast DDS Basic Configuration Example

Execute a *Fast DDS* `BasicConfigurationExample` *publisher* in domain `0`:

```
./BasicConfigurationExample publisher --domain 0
```

Execute a *Fast DDS* `BasicConfigurationExample` *subscriber* in domain `1`:

```
./BasicConfigurationExample subscriber --domain 1
```

Execute *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/xml.yaml`). Once the *DDS Router* is running, messages from *publisher* in domain 0 will be forwarded by the Router to the *subscriber* in domain 1, that will print them in `stdout`.

Execute with ROS 2 demo nodes

Execute a ROS 2 `demo_nodes_cpp` *talker* in domain `0`:

```
ROS_DOMAIN_ID=0 ros2 run demo_nodes_cpp talker
```

Execute a ROS 2 `demo_nodes_cpp` *listener* in domain `1`:

```
ROS_DOMAIN_ID=1 ros2 run demo_nodes_cpp listener
```

Execute *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/xml.yaml`). Once the *DDS Router* is running, messages from *talker* in domain 0 will be forwarded by the Router to the *listener* in domain 1, that will print them in `stdout`.

3.22 Forwarding Routes

The following YAML configuration file configures a DDS Router to create two *Simple Participants* in domains `0` and `1`. It then establishes a generic forwarding route between them, and a topic forwarding route for the topic `Circle`.

Note: This configuration enables participants in different domains to publish and subscribe to topics, as in *Change Domain Example*. The difference is that in this example we will use forwarding routes to define the participants we want to send messages to.

```
########################
# CONFIGURATION VERSION
version: v4.0                                                    # 0

##############
# PARTICIPANTS
participants:                                                    # 1
```

(continues on next page)

(continued from previous page)

```yaml
####################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in domain 0.

  - name: SimpleParticipant_0                                    # 2
    kind: local                                                  # 3
    domain: 0                                                    # 4

####################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in domain 1.

  - name: SimpleParticipant_1
    kind: local
    domain: 1


#################
# GENERIC ROUTES
routes:                                                          # 5

#######
# ROUTE
# This participant will forward the data it receives to
# SimpleParticipant_1
  - src: SimpleParticipant_0                                     # 6
    dst:                                                         # 7
      - SimpleParticipant_1                                      # 8

#######
# ROUTE
# This participant will not forward the data it receives.
  - src: SimpleParticipant_1                                     # 10


###############
# TOPIC ROUTES
topic-routes:                                                    # 11

  - name: Circle                                                 # 12
    type: HelloWorld                                             # 13

    routes:                                                      # 14

#############
# TOPIC ROUTE
# This participant will forward the data it receives on
# topic Circle to SimpleParticipant_0.
      - src: SimpleParticipant_1                                 # 15
        dst:                                                     # 16
          - SimpleParticipant_0                                  # 17
```

(continues on next page)

(continued from previous page)

```
#############
# TOPIC ROUTE
# This participant will not forward the data it receives
# on topic Circle.
      - src: SimpleParticipant_0
```

3.22.1 Configuration

Simple Participants

These participants are configured with a name, a kind (`local`), and a domain id (`0` and `1`).

```
##############
# PARTICIPANTS
participants:                                                    # 1

####################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in domain 0.

  - name: SimpleParticipant_0                                    # 2
    kind: local                                                  # 3
    domain: 0                                                    # 4

####################
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in domain 1.

  - name: SimpleParticipant_1
    kind: local
    domain: 1
```

Generic Forwarding Routes

We define the generic forwarding routes under the tag `routes`. This route is configured so that `SimpleParticipant_1` subscribes to the data published by `SimpleParticipant_0`.

```
#######
# ROUTE
# This participant will forward the data it receives to
# SimpleParticipant_1
  - src: SimpleParticipant_0                                     # 6
    dst:                                                         # 7
      - SimpleParticipant_1                                      # 8
```

This route is configured so that `SimpleParticipant_1` does not publish the data it receives. Thus, a subscriber in domain `0` would not receive the data published in domain `1`.

```
#######
# ROUTE
```

(continues on next page)

(continued from previous page)

```
# This participant will not forward the data it receives.
  - src: SimpleParticipant_1                                    # 10
```

Topic Forwarding Routes

We define the topic forwarding routes under the tag `topic-routes` by declaring the topic's `name`, `type`, and `routes`.

```
##############
# TOPIC ROUTES
topic-routes:                                                   # 11

  - name: Circle                                                # 12
    type: HelloWorld                                            # 13

    routes:                                                     # 14
```

Note: The `type` tag is required. The topic forwarding route will not be set for a topic with the same `name` but a different `type`.

Then, we declare the route for each participant.

Warning: When there is not a topic forwarding route for a specific topic, the generic forwarding route will be completely ignored and the topic forwarding route will be used instead.

This route is configured so that `SimpleParticipant_0` will subscribe to the data published by `SimpleParticipant_1`.

```
#############
# TOPIC ROUTE
# This participant will forward the data it receives on
# topic Circle to SimpleParticipant_0.
      - src: SimpleParticipant_1                                # 15
        dst:                                                    # 16
          - SimpleParticipant_0                                 # 17
```

This route is configured so that `SimpleParticipant_0` does not forward the data it receives.

```
#############
# TOPIC ROUTE
# This participant will not forward the data it receives
# on topic Circle.
      - src: SimpleParticipant_0
```

3.22.2 Execute example

Please refer to the *User Interface* section for a detailed explanation on how to execute the *DDS Router*.

Execute with Fast DDS Basic Configuration Example

To check that the generic forwarding routes are working we are going to execute two examples. In the first one, we will set up a *publisher* in domain `0`, a *subscriber* in domain `1`, and check that the subscriber receives the publications. In the second one, we will set up a *publisher* in domain `1`, a *subscriber* in domain `0`, and check that the subscriber does not receive the publications, since there are no generic forwarding routes from domain `1` to domain `0`.

Publish in domain 0 and subscribe in domain 1

Execute a *Fast DDS* `BasicConfigurationExample` *publisher* in domain `0`:

```
./BasicConfigurationExample publisher --domain 0
```

Execute a *Fast DDS* `BasicConfigurationExample` *subscriber* in domain `1`:

```
./BasicConfigurationExample subscriber --domain 1
```

Execute the *DDS Router* with the configuration file available at `<path/to/ddsrouter_tool>/share/resources/configurations/examples/forwarding_routes.yaml`. Once the *DDS Router* is running, messages from the *publisher* in domain `0` will be forwarded by the *DDS Router* to the *subscriber* in domain `1`, that will print them in `stdout`.

Publish in domain 1 and subscribe in domain 0

Execute a *Fast DDS* `BasicConfigurationExample` *publisher* in domain `1`:

```
./BasicConfigurationExample publisher --domain 1
```

Execute a *Fast DDS* `BasicConfigurationExample` *subscriber* in domain `0`:

```
./BasicConfigurationExample subscriber --domain 0
```

Execute the *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/forwarding_routes.yaml`). Once the *DDS Router* is running, nothing should happen, since there are no generic forwarding routes from domain `1` to domain `0`.

Execute with Fast DDS Basic Configuration Example on topic Circle

To check that the topic forwarding routes are working we are going to execute two examples. In the first one, we will set up a *publisher* in domain `1`, a *subscriber* in domain `0`, and check that the subscriber receives the publications on topic `Circle`. In the second one, we will set up a *publisher* in domain `0`, a *subscriber* in domain `1`, and check that the subscriber does not receive the publications, since there are no topic forwarding routes from domain `0` to domain `1` on topic `Circle`.

Publish in domain 1 and subscribe in domain 0

Execute a *Fast DDS* `BasicConfigurationExample` *publisher* in domain `1` on topic `Circle`:

```
./BasicConfigurationExample publisher --domain 1 --topic Circle
```

Execute a *Fast DDS* `BasicConfigurationExample` *subscriber* in domain `0` on topic `Circle`:

```
./BasicConfigurationExample subscriber --domain 0 --topic Circle
```

Execute the *DDS Router* with the configuration file available at `<path/to/ddsrouter_tool>/share/resources/configurations/examples/forwarding_routes.yaml`. Once the *DDS Router* is running, messages from the *publisher* in domain `1` on topic `Circle` will be forwarded by the *DDS Router* to the *subscriber* in domain `0`, that will print them in `stdout`.

Publish in domain 0 and subscribe in domain 1

Execute a *Fast DDS* `BasicConfigurationExample` *publisher* in domain `0` on topic `Circle`:

```
./BasicConfigurationExample publisher --domain 0 --topic Circle
```

Execute a *Fast DDS* `BasicConfigurationExample` *subscriber* in domain `1` on topic `Circle`:

```
./BasicConfigurationExample subscriber --domain 1 --topic Circle
```

Execute the *DDS Router* with this configuration file (available in `<path/to/ddsrouter_tool>/share/resources/configurations/examples/forwarding_routes.yaml`). Once the *DDS Router* is running, nothing should happen, since there are no topic forwarding routes on topic `Circle` from domain `0` to domain `1`.

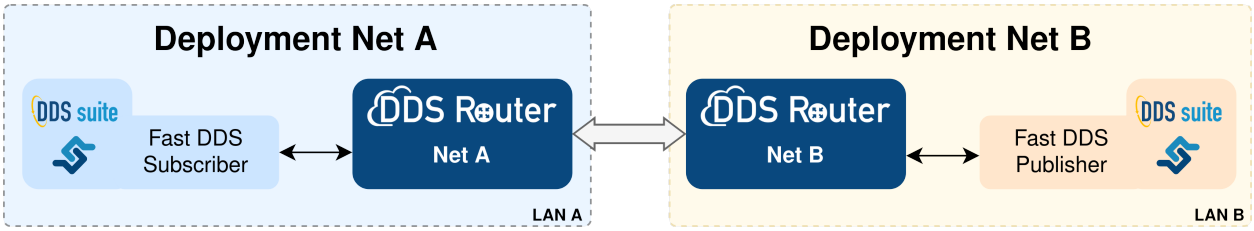
3.23 WAN communication over TCP

- *Prerequisites*
  - *Hardware requirements*
  - *Software requirements*
- *Deployment on Net A*
  - *Running Fast DDS Subscriber*
  - *Running DDS Router Net A*
- *Deployment on Net B*
  - *Running Fast DDS Publisher*
  - *Running the DDS Router Net B*
- *Results*

This tutorial covers the first steps in setting up a WAN DDS communication over TCP. More specifically, we will run a DDS Subscriber on a machine deployed on a local `network A` and a DDS Publisher on a machine deployed on a local `network B`. Both DDS entities will communicate over the Internet by means of the DDS Router.

Warning: This tutorial is intended for WAN communication. However, if there is only access to a LAN communication, it is possible to follow the tutorial by changing the DDS Domain Id so DDS entities in LAN A use default Domain (`0`) and those in LAN B use DDS Domain 1. This way the DDS entities are logically isolated and will not discovery other entities out of their DDS Domain.

The image below describes the scenario presented in this tutorial.



Several key elements can be observed in it:

1. **Fast DDS Subscriber** and **Fast DDS Publisher**. The application used for this tutorial is the *BasicHelloWorldExample*. The *BasicHelloWorldExample* is a Fast DDS application that implements a subscriber and a publisher, to which a configuration of basic DDS and QoS parameters such as DDS domain, transport, communication reliability, among others, can be applied.

1. **DDS Router**. *eProsima DDS Router* is an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscriber deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed in different geographic areas on their own dedicated local networks as if they were all on the same network through the use of *DDS Router*.

   This example presents two routers that enable Internet communication:

   - *DDS Router Net A*. This is the DDS Router that is deployed on *LAN A* and configured as TCP client. This way it is possible for the robot to communicate out-of-the-box with an external DDS Router configured as TCP server.
   - *DDS Router Net B*. It plays the server role in the communication. It will expose a public network address to which client DDS Routers connect to establish communication.

   It is important to mention that there is no correlation between DDS Publisher and Subscriber and TCP client and server. That is, the functionality of the DDS entity is independent of its behavior in TCP communication. Thus, although one DDS Router acts as a TCP client and another as a TCP server, both share information in both directions of communication.

3.23.1 Prerequisites

Hardware requirements

This tutorial will require two machines (*Deploymen Net A* and *Deploymen Net B*) deployed on different networks (*LAN A* and *LAN B*). It is possible to simulate the scenario by deploying everything needed on the same machine and two virtual networks but let's focus on the case of a real deployment.

Software requirements

You have two options:

1. Install all components individually on each of the machines. If you follow this option, you will need to install:
   - Fast DDS
   - Fast DDS BasicHelloWorldExample
   - *DDS Router*
2. Use the Fast DDS Suite. This is a publicly available Docker image that contains all the software needed for this tutorial. You can download this Docker image from the eProsima downloads website.

   To load this image into your Docker repository, from a terminal run

   ```
   $ docker load -i ubuntu-fastdds-suite:<FastDDS-Version>.tar
   ```

   You can run this Docker container as follows

   ```
   $ docker run --net=host -it ubuntu-fastdds-suite:<FastDDS-Version>
   ```

3.23.2 Deployment on Net A

First, let's deploy the *Fast DDS Subscriber* and *DDS Router Net A* on a machine on *LAN A*.

Running Fast DDS Subscriber

1. Run a Docker container using the Fast DDS Suite image.

   ```
   docker run --net=host -it ubuntu-fastdds-suite:<FastDDS-Version>
   ```

1. Run the Fast DDS Subscriber executing the following commands:

   ```
   goToExamples
   ./dds/BasicConfigurationExample/bin/BasicConfigurationExample subscriber --
   ↪transport udp
   ```

Running DDS Router Net A

As you know, DDS Router is configured through a YAML configuration file. Therefore, since we are going to run DDS Router from a Docker container, this file must be shared so that it is accessible within the container.

Let's create first the DDS Router configuration file. It will look like the one shown below.

```
version: v4.0

participants:

  - name: DDS_LAN_A
    kind: local

  - name: Router_Client
```

(continues on next page)

(continued from previous page)

```
    kind: wan
    connection-addresses:
      - ip: 123.123.123.123
        port: 45678
        transport: tcp
```

Next, it is briefly explained the most relevant aspects of this configuration file.

The `participants` are the interfaces of the DDS Router to communicate with other networks. In this case, we have two participants:

- `local`: this is a simple participant that communicates with all DDS entities it finds in the default DDS Domain, domain 0. For more information about this participant please refer to the *Simple Participant section* of this documentation.
- `wan`: it is a participant designed for WAN communication or the communication between two *DDS Routers*. It uses the Fast DDS Initial Peers configuration to establish a point-to-point communication between two DDS entities, two *DDS Routers* in this case. For now it is not necessary to know more about the details of this configuration as you only need to configure the connection address of the DDS Router.

  For the *DDS Router Net A*, a connection address shall be defined which must be the same as the one exposed by the *DDS Router Net B*. There are some relevant configurations within this connection address:

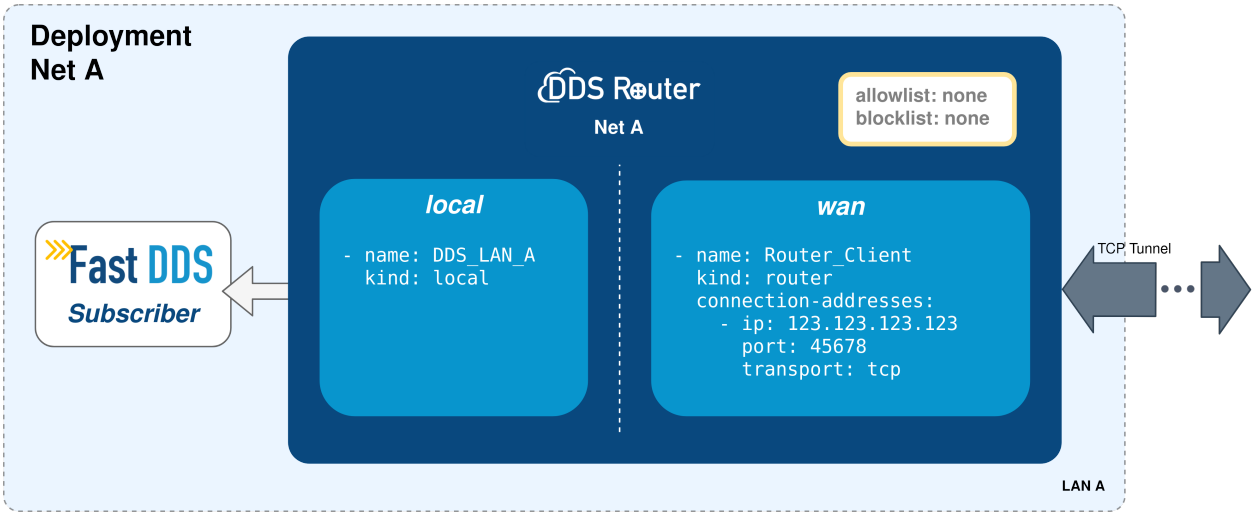
  - `connection-addresses`: defines the IP (`ip`) and port (`port`) of the network addresses to which it connects, and the transport protocol (`transport`) to be used in the communication, TCP in this case.

Note: In this case, the DDS Router will forward all topics found in the LAN A DDS network. However, it is important to mention that the DDS topics relayed by the DDS Router can be filtered by configuring the `allowlist` and `blocklist`. If this is the case please refer to the *Topic filtering documentation* for information on how to do this.

To finish this step, run the DDS Router with the configuration file created as an argument.

```
docker run --net=host -it -v <path/to/file>/ddsrouter_net_A.yaml:/ddsrouter_net_A.yaml␣
↪ubuntu-fastdds-suite:<FastDDS-Version>
ddsrouter -c /ddsrouter_net_A.yaml
```

The following figure summarizes the deployment on LAN A.



3.23.3 Deployment on Net B

Running Fast DDS Publisher

1. Run a Docker container using the Fast DDS Suite image.

```
docker run --net=host -it ubuntu-fastdds-suite:<FastDDS-Version>
```

1. Run the Fast DDS Publisher executing the following commands:

   WAN

```
goToExamples
./dds/BasicConfigurationExample/bin/BasicConfigurationExample publisher \
    --interval 1000 --transport udp
```

   LAN

```
goToExamples
./dds/BasicConfigurationExample/bin/BasicConfigurationExample publisher \
    --interval 1000 --transport udp --domain 1
```

Note: As stated *here*, change the DDS Domain Id if running the edge and cloud applications on the same LAN.

Running the DDS Router Net B

Configure transversal NAT on the network router

The first thing to do before starting to configure DDS Router is to configure the network router to allow a remote communication from the Internet to reach a specific device on the LAN, more specifically to expose an IP address and a port to the network that will be used by our DDS Router application.

This configuration will depend on your network router, but it should be similar to the one shown in the following image.

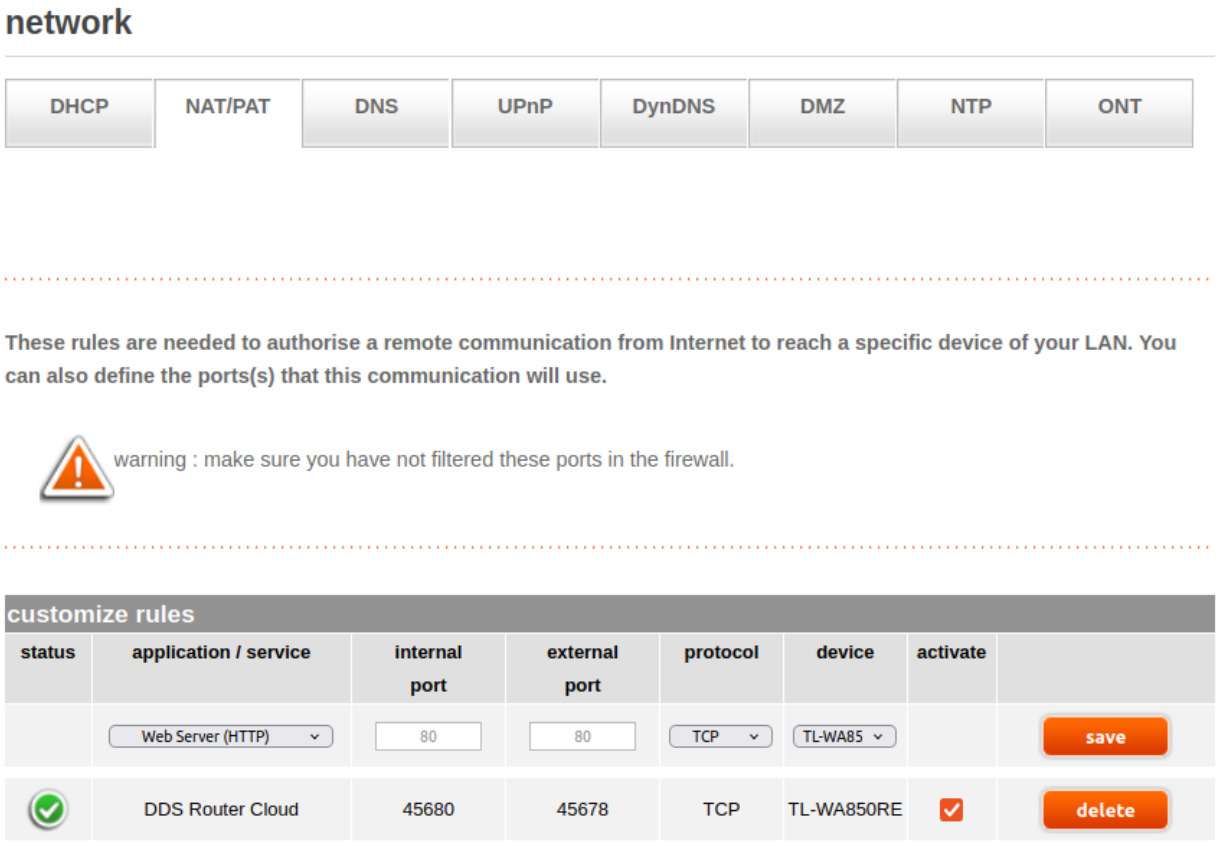
Configure the DDS Router Net B

The DDS Router Net B configuration file is quite similar to the DDS Router Net A configuration file, as can be seen below:

WAN

```
version: v4.0

participants:

  - name: DDS_LAN_B
    kind: local

  - name: Router_Server
    kind: wan
    listening-addresses:
```

(continues on next page)



(continued from previous page)

```
      - ip: 123.123.123.123
        port: 45680
        external-port: 45678
        transport: tcp
```

LAN

```
version: v4.0

participants:

  - name: DDS_LAN_B
    kind: local
    domain: 1

  - name: Router_Server
    kind: wan
    listening-addresses:
      - ip: 123.123.123.123
        port: 45678
        transport: tcp
```

Note: As stated *here*, set the DDS Domain Id on the `local` participant in order to discover the Fast DDS Publisher.

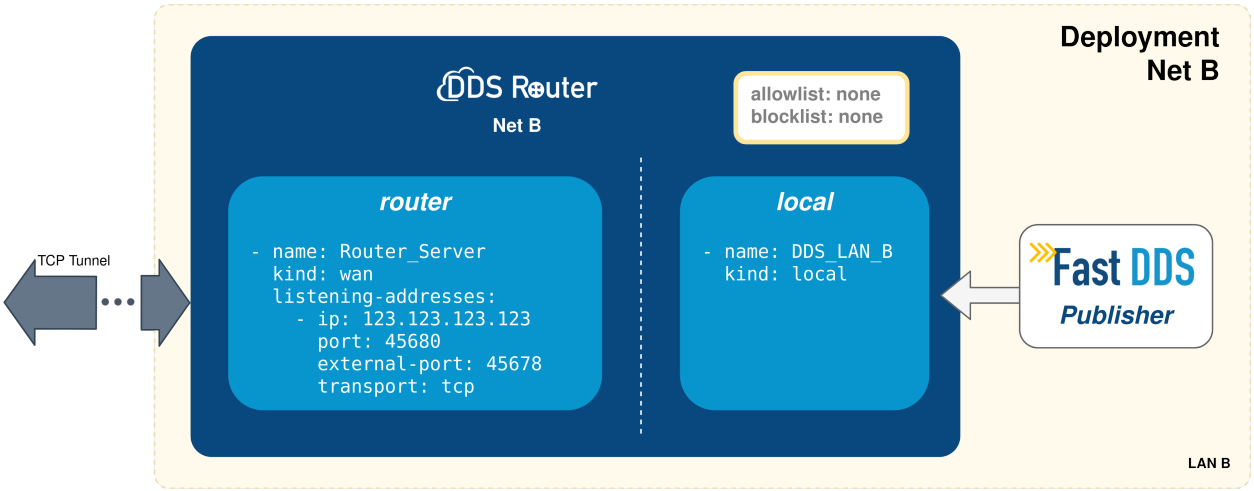
In this case there are also two participants, two communication interfaces for the DDS Router. The first one communicates the DDS Router with any DDS entity, while the second one enables to establish a communication channel with another DDS Router.

Even so there are some differences in the second participant that are worth mentioning. This participant sets a listening address (`listening-addresses`), rather than a connection address. This is because it is the participant that waits for incoming communications since it has this network address exposed and accessible from the Internet.

To finish, run the DDS Router Net B with the above configuration.

```
docker run --net=host -it -v <path/to/file>/ddsrouter_net_B.yaml:/ddsrouter_net_B.yaml␣
↪ubuntu-fastdds-suite:<FastDDS-Version>
ddsrouter -c /ddsrouter_net_B.yaml
```

The following figure summarizes the deployment on LAN B.



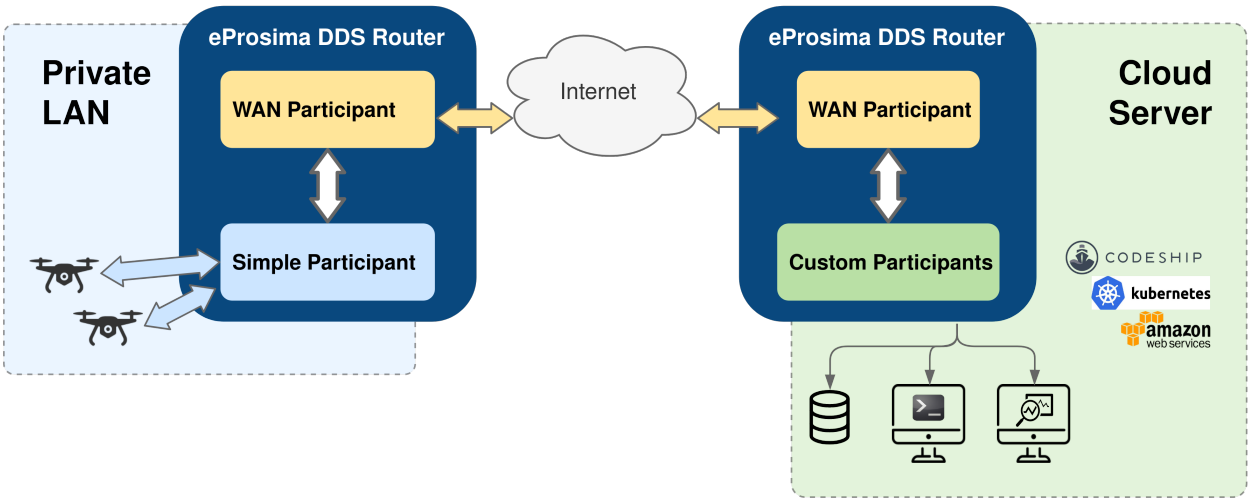
3.23.4 Results

If all the steps in this tutorial have been followed, the Fast DDS Subscriber on Deployment Net A should start receiving samples and prompting the following traces:

```
Message HelloWorld  10 RECEIVED
Message HelloWorld  11 RECEIVED
Message HelloWorld  12 RECEIVED
Message HelloWorld  13 RECEIVED
Message HelloWorld  14 RECEIVED
```

3.24 ROS 2 and Kubernetes

Apart from plain LAN-to-LAN communication, Cloud environments such as container-oriented platforms have also been present throughout the *DDS Router* design phase. In this walk-through example, we will set up both a *Kubernetes* (*K8s*) network and a local environment in order to establish communication between a pair of ROS nodes, one sending messages from a LAN (talker) and another one (listener) receiving them in the Cloud. This will be accomplished by having a *DDS Router* instance at each side of the communication.

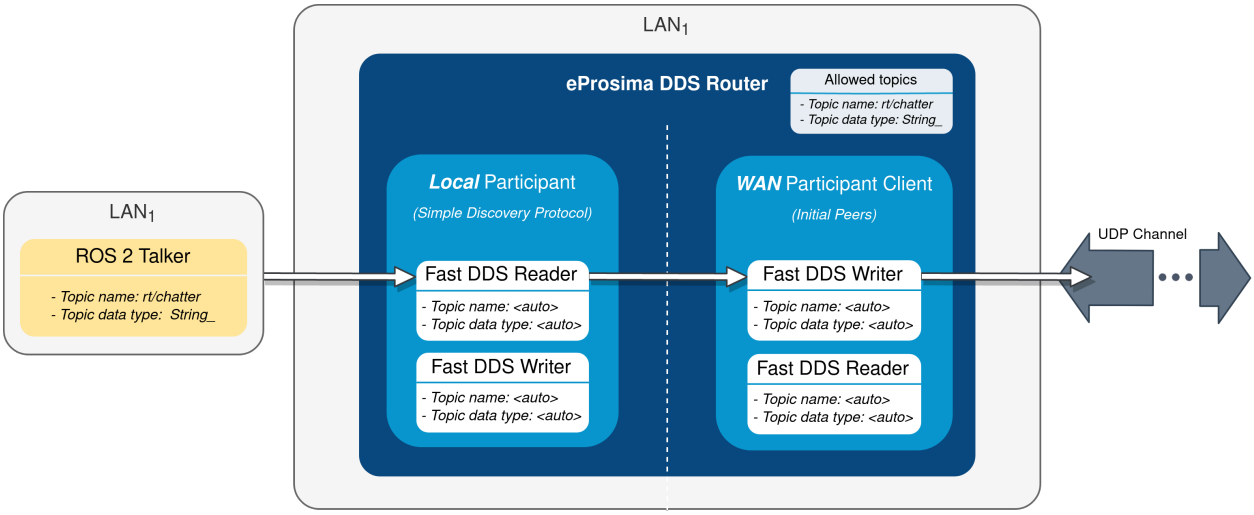


3.24.1 Local setup

The local instance of *DDS Router* (local router) only requires to have a *Simple Participant*, and a *WAN Participant* that will play the client role in the discovery process of remote participants (see *Initial Peers discovery mechanism*).

After having acknowledged each other's existence through Simple DDS discovery mechanism (multicast communication), the local participant will start receiving messages published by the ROS 2 talker node, and will then forward them to the WAN participant. Following, these messages will be sent to another participant hosted on a *K8s* cluster to which it connects via WAN communication over UDP/IP.

Following is a representation of the above-described scenario:



Local router

The configuration file used by the local router will be the following:

```
# local-ddsrouter.yaml

version: v4.0

allowlist:
  - name: rt/chatter
    type: std_msgs::msg::dds_::String_

participants:

  - name: SimpleParticipant
    kind: local
    domain: 0

  - name: LocalWAN
    kind: wan
    listening-addresses:      # Needed for UDP communication
      - ip: 3.3.3.3           # LAN public IP
        port: 30003
        transport: udp
    connection-addresses:
      - ip: 2.2.2.2           # Public IP exposed by the k8s cluster to reach the cloud␣
↪DDS-Router
        port: 30002
        transport: udp
```

Note that the simple participant will be receiving messages sent in DDS domain `0`. Also note that, due to the choice of UDP as transport protocol, a listening address with the LAN public IP address needs to be specified for the local WAN participant, even when behaving as client in the participant discovery process. Make sure that the given port is reachable from outside this local network by properly configuring port forwarding in your Internet router device. The connection address points to the remote WAN participant deployed in the *K8s* cluster. For further details on how to configure WAN communication, please have a look at *WAN Configuration* and *WAN Participant Configuration Example*.

Note: As an alternative, *TCP transport* may be used instead of UDP. This has the advantage of not requiring to set a listening address in the local router's WAN participant (TCP client), so there is no need to fiddle with the configuration of your Internet router device.

To launch the local router, execute:

```
ddsrouter --config-path local-ddsrouter.yaml
```

Talker

This example will make use of *ROS 2 galactic* with `demo-nodes-cpp` package installed. If not already present in your system, you may choose any of the available options to install ROS galactic, or even consider directly using a distributed Docker image. Just make sure the resulting environment is prepared to utilize *eProsima Fast DDS* as middleware (see Working with eProsima Fast DDS).

Once *ROS 2* is installed, start publishing messages in DDS domain `0` by executing:

```
RMW_IMPLEMENTATION=rmw_fastrtps_cpp ros2 run demo_nodes_cpp talker
```

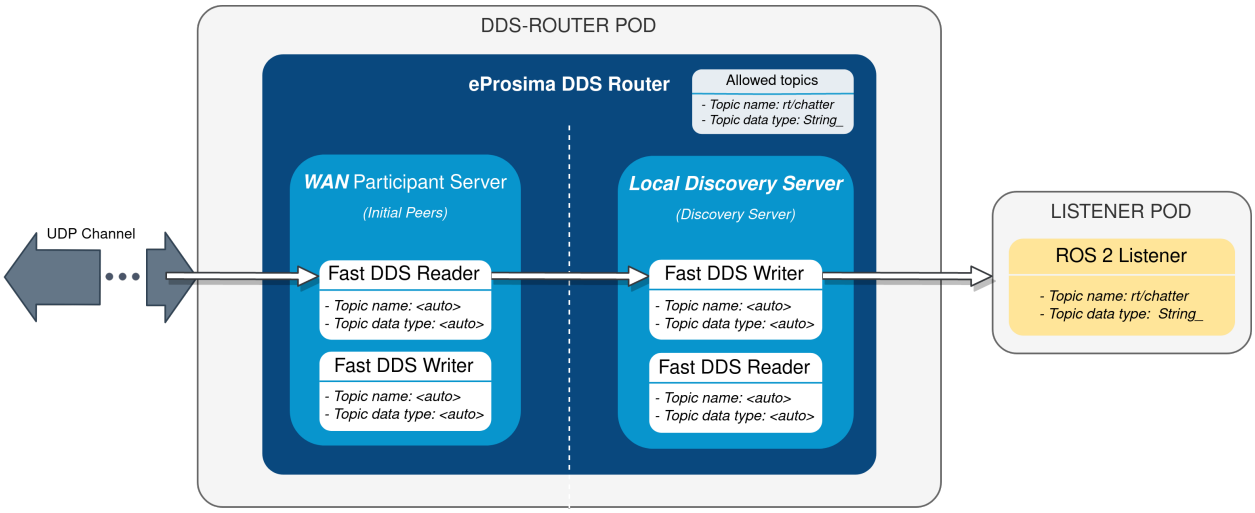
3.24.2 Kubernetes setup

Two different deployments will be used for this example, each in a different *K8s* pod. The *DDS Router* cloud instance (cloud router) consists of two participants:

- A *WAN Participant* that receives the messages coming from our LAN through the aforementioned UDP communication channel.
- A *Local Discovery Server* (local DS) that propagates them to a ROS 2 listener node hosted in a different *K8s* pod.

The choice of a Local Discovery Server instead of a Simple Participant to communicate with the listener has to do with the difficulty of enabling multicast routing in cloud environments.

The described scheme is represented in the following figure:



In addition to the two mentioned deployments, two *K8s* services are required in order to direct dataflow to each of the pods. A LoadBalancer will forward messages reaching the cluster to the WAN participant of the cloud router, and a ClusterIP service will be in charge of delivering messages from the local DS to the listener pod. Following are the settings needed to launch these services in *K8s*:

```
kind: Service
apiVersion: v1
metadata:
  name: ddsrouter
  labels:
    app: ddsrouter
```

(continues on next page)

(continued from previous page)

```
spec:
  ports:
    - name: UDP-30002
      protocol: UDP
      port: 30002
      targetPort: 30002
  selector:
    app: ddsrouter
  type: LoadBalancer
```

```
kind: Service
apiVersion: v1
metadata:
  name: local-ddsrouter
spec:
  ports:
    - name: UDP-30001
      protocol: UDP
      port: 30001
      targetPort: 30001
  selector:
    app: ddsrouter
  clusterIP: 192.168.1.11  # Private IP only reachable within the k8s cluster to␣
↪communicate with the ddsrouter application
  type: ClusterIP
```

Note: An Ingress needs to be configured for the LoadBalancer service to make it externally-reachable. In this example we consider the assigned public IP address to be `2.2.2.2`.

The configuration file used for the cloud router will be provided by setting up a ConfigMap:

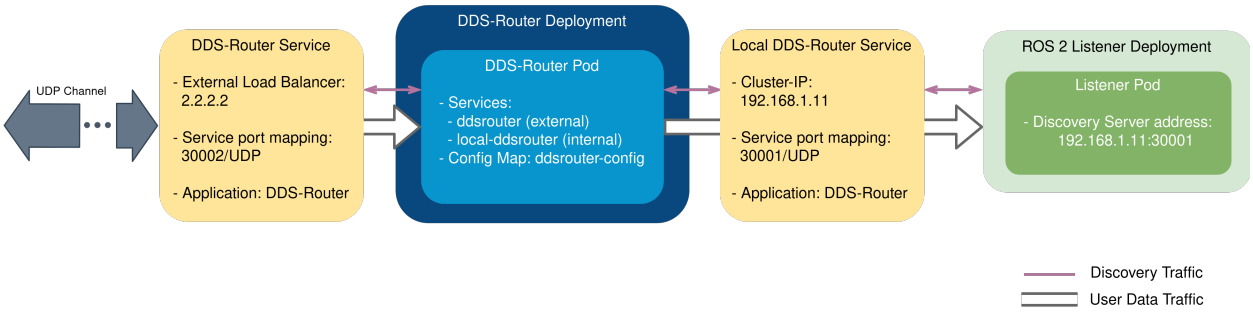
```
kind: ConfigMap
apiVersion: v1
metadata:
  name: ddsrouter-config
data:
  ddsrouter.config.file: |-
    version: v4.0

    allowlist:
      - name: rt/chatter
        type: std_msgs::msg::dds_::String_

    participants:

      - name: LocalDiscoveryServer
        kind: local-discovery-server
        discovery-server-guid:
          ros-discovery-server: true
          id: 1
```

(continues on next page)

(continued from previous page)

```
        listening-addresses:
          - ip: 192.168.1.11      # Private IP only reachable within the k8s cluster to␣
↪communicate with the ddsrouter application
            port: 30001
            transport: udp

      - name: CloudWAN
        kind: wan
        listening-addresses:
          - ip: 2.2.2.2           # Public IP exposed by the k8s cluster to reach the␣
↪cloud DDS-Router
            port: 30002
            transport: udp
```

Following is a representation of the overall *K8s* cluster configuration:



DDS-Router deployment

The cloud router is launched from within a Docker image, which uses as configuration file the one hosted in the previously set up ConfigMap. This Docker image needs to be built and made available to the *K8s* cluster for using *DDS Router*, which can be accomplished by providing the following `Dockerfile`. If willing to see log messages in `STDOUT`, use `Dockerfile` instead. Assuming the name of the generated Docker image is `ddsrouter:main`, the cloud router will then be deployed with the following settings:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: ddsrouter
  labels:
    app: ddsrouter
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ddsrouter
  template:
    metadata:
      labels:
        app: ddsrouter
    spec:
```

(continues on next page)

(continued from previous page)

```
      volumes:
        - name: config
          configMap:
            name: ddsrouter-config
            items:
              - key: ddsrouter.config.file
                path: DDSROUTER_CONFIGURATION.yaml
      containers:
        - name: ddsrouter
          image: ddsrouter:main
          ports:
            - containerPort: 30001
              protocol: UDP
            - containerPort: 30002
              protocol: UDP
          volumeMounts:
            - name: config
              mountPath: /ddsrouter/resources
      restartPolicy: Always
```

Listener deployment

A suitable Docker image must also be provided in the context of the cluster in order to use *ROS 2*. We will use `ros:galactic` as basis for this image, install `demo-nodes-cpp`, and include a parser that will allow us to specify the port and IP address of the local DS. This can be achieved by using the following Dockerfile and entrypoint:

```
FROM ros:galactic

SHELL ["/bin/bash", "-c"]

# Install demo-nodes-cpp
RUN source /opt/ros/$ROS_DISTRO/setup.bash && \
    apt update && \
    apt install -y ros-$ROS_DISTRO-rmw-fastrtps-cpp && \
    apt install -y ros-$ROS_DISTRO-demo-nodes-cpp

# Set Fast DDS as middleware
ENV RMW_IMPLEMENTATION=rmw_fastrtps_cpp

COPY ./run.bash /
RUN chmod +x /run.bash

# Setup entrypoint
ENTRYPOINT ["/run.bash"]
```

```
#!/bin/bash

if [[ $1 == "listener" ]]
then
    NODE="listener"
else
```

(continues on next page)

(continued from previous page)

```
    NODE="talker"
fi

SERVER_IP=$2
SERVER_PORT=$3

# Setup environment
source "/opt/ros/$ROS_DISTRO/setup.bash"

echo "Starting ${NODE} as client of Discovery Server ${SERVER_IP}:${SERVER_PORT}"
ROS_DISCOVERY_SERVER=";${SERVER_IP}:${SERVER_PORT}" ros2 run demo_nodes_cpp ${NODE}
```

Now, assuming the name of the built image is `ros2-demo-nodes:galactic`, the listener pod can be deployed by providing the following configuration:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: ros2-galactic-listener
  labels:
    app: ros2-galactic-listener
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ros2-galactic-listener
  template:
    metadata:
      labels:
        app: ros2-galactic-listener
    spec:
      containers:
        - name: ros2-demo-nodes
          image: ros2-demo-nodes:galactic
          args:
            - listener
            - 192.168.1.11
            - '30001'
      restartPolicy: Always
```

Once all these components are up and running, communication should have been established between talker and listener nodes, so that messages finally manage to reach the listener pod and get printed in its `STDOUT`.

Feel free to interchange the locations of the ROS nodes by slightly modifying the provided configuration files, hosting the talker in the *K8s* cluster while the listener runs in our LAN.

3.25 Repeater DDS Router

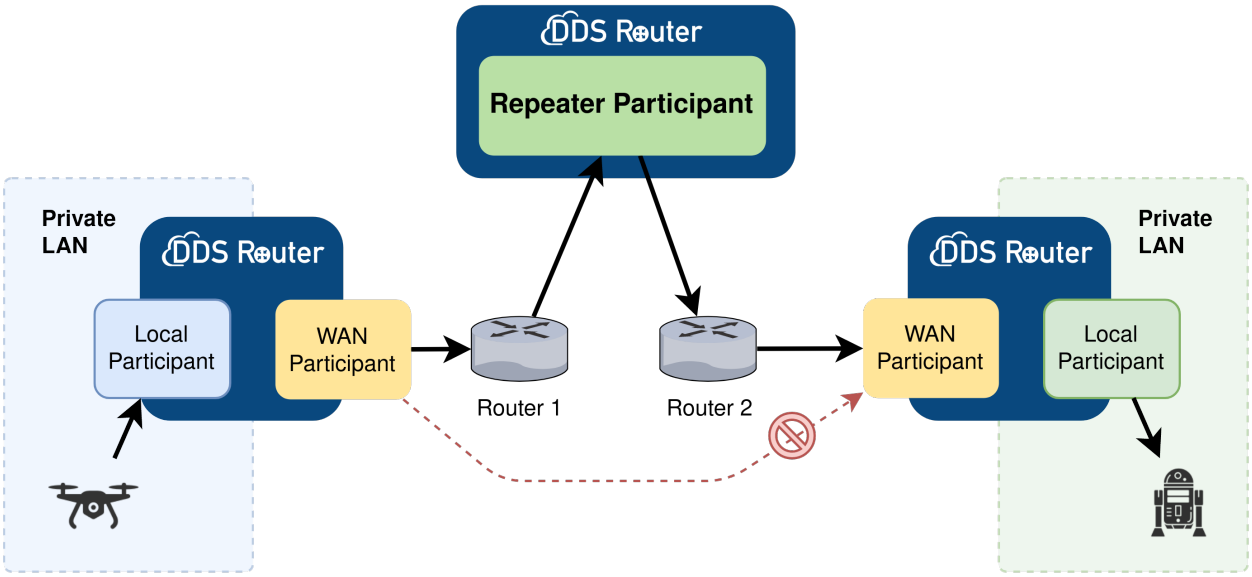
A *DDS Router* could work as a *TURN* **Repeater**. This means that a *DDS Router* can be used to repeat messages between networks.

3.25.1 Use case

The use of a *TURN* Server is very useful in the following scenarios:

- **NAT Traversal**: If the edge devices are under different NATs, they cannot access each other if no ports are opened in their respective internet access points.
- **Unreachable Network**: If edge devices work under different networks (e.g. using different transport protocols, connected in different private networks, etc.) cannot reach each other.

The following figure exemplifies these use cases. When the communication between edge routers is not possible, a Repeater can be set in the middle to forward data and make the communication possible.



3.25.2 How to configure

This *DDS Router* configuration is very simple, as all it needs is a *WAN Participant* and adding to its configuration the tag *repeater: true*. There could be more Participants in this *DDS Router* and topic filtering is also available. The Repeater Participant only works as other normal Participant, with the particularity that resend forward the data that receives.

In order to see an example of this configuration, access to the following example *Repeater Example*.

3.26 Linux installation from sources

The instructions for installing the *DDS Router* application from sources and its required dependencies are provided in this page. It is organized as follows:

- *Dependencies installation*
  - *Requirements*
  - *Dependencies*
- *Colcon installation*
- *CMake installation*
  - *Local installation*
  - *Global installation*
- *Run an application*

3.26.1 Dependencies installation

DDS Router depends on *eProsima Fast DDS* library and certain Debian packages. This section describes the instructions for installing *DDS Router* dependencies and requirements in a Linux environment from sources. The following packages will be installed:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocation library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of eProsima Fast DDS library.
- `cmake_utils`, an eProsima utils library for CMake.
- `cpp_utils`, an eProsima utils library for C++.

First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon* or the *CMake* installation instructions.

Requirements

The installation of *DDS Router* in a Linux environment from sources requires the following tools to be installed in the system:

- *CMake, g++, pip, wget and git*
- *Colcon* [optional]
- *Gtest* [for test only]
- *PyYAML* [for YAML Validator only]
- *jsonschema* [for YAML Validator only]

CMake, g++, pip, wget and git

These packages provide the tools required to install *DDS Router* and its dependencies from command line. Install CMake, g++, pip, wget and git using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install cmake g++ pip wget git
```

Colcon

colcon is a command line tool based on CMake aimed at building sets of software packages. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

```
pip3 install -U colcon-common-extensions vcstool
```

Note: If this fails due to an Environment Error, add the `--user` flag to the `pip3` installation command.

Gtest

Gtest is a unit testing library for C++. By default, *DDS Router* does not compile tests. It is possible to activate them with the opportune CMake options when calling colcon or CMake. For more details, please refer to the *CMake options* section. For a detailed description of the Gtest installation process, please refer to the Gtest Installation Guide.

It is also possible to clone the Gtest Github repository into the *DDS Router* workspace and compile it with colcon as a dependency package. Use the following command to download the code:

```
git clone --branch release-1.11.0 https://github.com/google/googletest src/googletest-
↪distribution
```

PyYAML

PyYAML is a YAML parser and emitter for Python.

It is used by the DDS-Router *YAML Validator* for loading the content of configuration files.

Install `pyyaml` by executing the following command:

```
pip3 install -U pyyaml
```

jsonschema

jsonschema is an implementation of the JSON Schema specification for Python.

It is used by the DDS-Router *YAML Validator* for performing validation of configuration files against a given JSON schema.

Install `jsonschema` by executing the following command:

```
pip3 install -U jsonschema
```

Dependencies

DDS Router has the following dependencies, when installed from sources in a Linux environment:

- *Asio and TinyXML2 libraries*
- *OpenSSL*
- *yaml-cpp*
- *eProsima dependencies*

Asio and TinyXML2 libraries

Asio is a cross-platform C++ library for network and low-level I/O programming, which provides a consistent asynchronous model. TinyXML2 is a simple, small and efficient C++ XML parser. Install these libraries using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libasio-dev libtinyxml2-dev
```

OpenSSL

OpenSSL is a robust toolkit for the TLS and SSL protocols and a general-purpose cryptography library. Install OpenSSL using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libssl-dev
```

yaml-cpp

yaml-cpp is a YAML parser and emitter in C++ matching the YAML 1.2 spec, and is used by *DDS Router* application to parse the provided configuration files. Install yaml-cpp using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install libyaml-cpp-dev
```

eProsima dependencies

If it already exists in the system an installation of *Fast DDS* library with version greater than *2.4.0*, just source this library when building the *DDS Router* application by using the command:

```
source <fastdds-installation-path>/install/setup.bash
```

In other case, just download *Fast DDS* project from sources and build it together with *DDS Router* using colcon as it is explained in section *Colcon installation*.

3.26.2 Colcon installation

1. Create a `DDS-Router` directory and download the `.repos` file that will be used to install *DDS Router* and its dependencies:

```
mkdir -p ~/DDS-Router/src
cd ~/DDS-Router
wget https://raw.githubusercontent.com/eProsima/DDS-Router/main/ddsrouter.repos
vcs import src < ddsrouter.repos
```

> **Note:** In case there is already a *Fast DDS* installation in the system it is not required to download and build every dependency in the `.repos` file. It is just needed to download and build the *DDS Router* project having sourced its dependencies. Refer to section *eProsima dependencies* in order to check how to source *Fast DDS* library.

2. Build the packages:

```
colcon build
```

Note: Being based on CMake, it is possible to pass the CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual.

3.26.3 CMake installation

This section explains how to compile *DDS Router* with CMake, either *locally* or *globally*.

Local installation

1. Create a `DDS-Router` directory where to download and build *DDS Router* and its dependencies:

```
mkdir -p ~/DDS-Router/src
mkdir -p ~/DDS-Router/build
cd ~/DDS-Router
wget https://raw.githubusercontent.com/eProsima/DDS-Router/main/ddsrouter.repos
vcs import src < ddsrouter.repos
```

2. Compile all dependencies using CMake.

   - Foonathan memory

```
cd ~/DDS-Router
mkdir build/foonathan_memory_vendor
cd build/foonathan_memory_vendor
cmake ~/DDS-Router/src/foonathan_memory_vendor -DCMAKE_INSTALL_PREFIX=~/
↪DDS-Router/install -DBUILD_SHARED_LIBS=ON
cmake --build . --target install
```

   - Fast CDR

```
cd ~/DDS-Router
mkdir build/fastcdr
cd build/fastcdr
cmake ~/DDS-Router/src/fastcdr -DCMAKE_INSTALL_PREFIX=~/DDS-Router/
↪install
cmake --build . --target install
```

- Fast DDS

```
cd ~/DDS-Router
mkdir build/fastdds
cd build/fastdds
cmake ~/DDS-Router/src/fastdds -DCMAKE_INSTALL_PREFIX=~/DDS-Router/
↪install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install
```

- Dev Utils

```
# CMake Utils
cd ~/DDS-Router
mkdir build/cmake_utils
cd build/cmake_utils
cmake ~/DDS-Router/src/dev-utils/cmake_utils -DCMAKE_INSTALL_PREFIX=~/
↪DDS-Router/install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install

# C++ Utils
cd ~/DDS-Router
mkdir build/cpp_utils
cd build/cpp_utils
cmake ~/DDS-Router/src/dev-utils/cpp_utils -DCMAKE_INSTALL_PREFIX=~/DDS-
↪Router/install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install
```

- DDS Pipe

```
# ddspipe_core
cd ~/DDS-Router
mkdir build/ddspipe_core
cd build/ddspipe_core
cmake ~/DDS-Router/src/ddspipe/ddspipe_core -DCMAKE_INSTALL_PREFIX=~/DDS-
↪Router/install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install

# ddspipe_participants
cd ~/DDS-Router
mkdir build/ddspipe_participants
cd build/ddspipe_participants
cmake ~/DDS-Router/src/ddspipe/ddspipe_participants -DCMAKE_INSTALL_
↪PREFIX=~/DDS-Router/install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install

# ddspipe_yaml
```

(continues on next page)

(continued from previous page)

```
cd ~/DDS-Router
mkdir build/ddspipe_yaml
cd build/ddspipe_yaml
cmake ~/DDS-Router/src/ddspipe/ddspipe_yaml -DCMAKE_INSTALL_PREFIX=~/DDS-
↪Router/install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install
```

3. Once all dependencies are installed, install *DDS Router*:

```
# ddsrouter_core
cd ~/DDS-Router
mkdir build/ddsrouter_core
cd build/ddsrouter_core
cmake ~/DDS-Router/src/ddsrouter/ddsrouter_core -DCMAKE_INSTALL_PREFIX=~/DDS-Router/
↪install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install

# ddsrouter_yaml
cd ~/DDS-Router
mkdir build/ddsrouter_yaml
cd build/ddsrouter_yaml
cmake ~/DDS-Router/src/ddsrouter/ddsrouter_yaml -DCMAKE_INSTALL_PREFIX=~/DDS-Router/
↪install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install

# ddsrouter_tool
cd ~/DDS-Router
mkdir build/ddsrouter_tool
cd build/ddsrouter_tool
cmake ~/DDS-Router/src/ddsrouter/tools/ddsrouter_tool -DCMAKE_INSTALL_PREFIX=~/DDS-
↪Router/install -DCMAKE_PREFIX_PATH=~/DDS-Router/install
cmake --build . --target install
```

Note: By default, *DDS Router* does not compile tests. However, they can be activated by downloading and installing Gtest and building with CMake option `-DBUILD_TESTS=ON`.

Global installation

To install *DDS Router* system-wide instead of locally, remove all the flags that appear in the configuration steps of `Fast-CDR`, `Fast-DDS`, and `DDS-Router`, and change the first in the configuration step of `foonathan_memory_vendor` to the following:

```
-DCMAKE_INSTALL_PREFIX=/usr/local/ -DBUILD_SHARED_LIBS=ON
```

3.26.4 Run an application

To run the *DDS Router* application, source the *Fast DDS* library and execute the executable file that has been installed in `<install-path>/ddsrouter_tool/bin/ddsrouter`:

```
# If built has been done using colcon, all projects could be sourced as follows
source install/setup.bash
./<install-path>/ddsrouter_tool/bin/ddsrouter
```

Be sure that this executable has execute permissions.

3.27 Windows installation from sources

The instructions for installing the *DDS Router* application from sources and its required dependencies are provided in this page. It is organized as follows:

- *Dependencies installation*
  - *Requirements*
  - *Dependencies*
- *Colcon installation*
- *CMake installation*
  - *Local installation*
  - *Global installation*
- *Run an application*

3.27.1 Dependencies installation

DDS Router depends on *eProsima Fast DDS* library and certain Debian packages. This section describes the instructions for installing *DDS Router* dependencies and requirements in a Windows environment from sources. The following packages will be installed:

- `foonathan_memory_vendor`, an STL compatible C++ memory allocation library.
- `fastcdr`, a C++ library that serializes according to the standard CDR serialization mechanism.
- `fastrtps`, the core library of eProsima Fast DDS library.
- `cmake_utils`, an eProsima utils library for CMake.
- `cpp_utils`, an eProsima utils library for C++.

First of all, the *Requirements* and *Dependencies* detailed below need to be met. Afterwards, the user can choose whether to follow either the *colcon* or the *CMake* installation instructions.

Requirements

The installation of *eProsima Fast DDS* in a Windows environment from sources requires the following tools to be installed in the system:

- *Visual Studio*
- *Chocolatey*
- *CMake, pip3, wget and git*
- *Colcon* [optional]
- *Gtest* [for test only]
- *PyYAML* [for YAML Validator only]
- *jsonschema* [for YAML Validator only]

Visual Studio

Visual Studio is required to have a C++ compiler in the system. For this purpose, make sure to check the `Desktop development with C++` option during the Visual Studio installation process.

If Visual Studio is already installed but the Visual C++ Redistributable packages are not, open Visual Studio and go to `Tools -> Get Tools and Features` and in the `Workloads` tab enable `Desktop development with C++`. Finally, click `Modify` at the bottom right.

Chocolatey

Chocolatey is a Windows package manager. It is needed to install some of *eProsima Fast DDS*'s dependencies. Download and install it directly from the website.

CMake, pip3, wget and git

These packages provide the tools required to install *eProsima Fast DDS* and its dependencies from command line. Download and install CMake, pip3, wget and git by following the instructions detailed in the respective websites. Once installed, add the path to the executables to the `PATH` from the *Edit the system environment variables* control panel.

Colcon

colcon is a command line tool based on CMake aimed at building sets of software packages. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

```
pip3 install -U colcon-common-extensions vcstool
```

Note: If this fails due to an Environment Error, add the `--user` flag to the `pip3` installation command.

Gtest

Gtest is a unit testing library for C++. By default, *DDS Router* does not compile tests. It is possible to activate them with the opportune CMake options when calling colcon or CMake. For more details, please refer to the *CMake options* section.

Run the following commands on your workspace to install Gtest.

```
git clone https://github.com/google/googletest.git
cmake -DCMAKE_INSTALL_PREFIX='C:\Program Files\gtest' -Dgtest_force_shared_crt=ON -
↪DBUILD_GMOCK=ON ^
    -B build\gtest -A x64 -T host=x64 googletest
cmake --build build\gtest --config Release --target install
```

or refer to the Gtest Installation Guide for a detailed description of the Gtest installation process.

PyYAML

PyYAML is a YAML parser and emitter for Python.

It is used by the DDS-Router *YAML Validator* for loading the content of configuration files.

Install `pyyaml` by executing the following command:

```
pip3 install -U pyyaml
```

jsonschema

jsonschema is an implementation of the JSON Schema specification for Python.

It is used by the DDS-Router *YAML Validator* for performing validation of configuration files against a given JSON schema.

Install `jsonschema` by executing the following command:

```
pip3 install -U jsonschema
```

Dependencies

DDS Router has the following dependencies, when installed from sources in a Windows environment:

- *Asio and TinyXML2 libraries*
- *OpenSSL*
- *yaml-cpp*
- *eProsima dependencies*

Asio and TinyXML2 libraries

Asio is a cross-platform C++ library for network and low-level I/O programming, which provides a consistent asynchronous model. TinyXML2 is a simple, small and efficient C++ XML parser. They can be downloaded directly from the links below:

- Asio
- TinyXML2

After downloading these packages, open an administrative shell with *PowerShell* and execute the following command:

```
choco install -y -s <PATH_TO_DOWNLOADS> asio tinyxml2
```

where <PATH_TO_DOWNLOADS> is the folder into which the packages have been downloaded.

OpenSSL

OpenSSL is a robust toolkit for the TLS and SSL protocols and a general-purpose cryptography library. Download and install the latest OpenSSL version for Windows at this link. After installing, add the environment variable `OPENSSL_ROOT_DIR` pointing to the installation root directory.

For example:

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

yaml-cpp

yaml-cpp is a YAML parser and emitter in C++ matching the YAML 1.2 spec, and is used by *DDS Router* application to parse the provided configuration files. From an administrative shell with *PowerShell*, execute the following commands in order to download and install yaml-cpp for Windows:

```
git clone --branch yaml-cpp-0.7.0 https://github.com/jbeder/yaml-cpp
cmake -DCMAKE_INSTALL_PREFIX='C:\Program Files\yamlcpp' -B build\yamlcpp yaml-cpp
cmake --build build\yamlcpp --target install    # If building in Debug mode, add --
↪config Debug
```

eProsima dependencies

If it already exists in the system an installation of *Fast DDS* library with version greater than *2.4.0*, just source this library when building the *DDS Router* application by using the command:

```
source <fastdds-installation-path>/install/setup.bash
```

In other case, just download *Fast DDS* project from sources and build it together with *DDS Router* using colcon as it is explained in section *Colcon installation*.

3.27.2 Colcon installation

Important: Run colcon within a Visual Studio prompt. To do so, launch a *Developer Command Prompt* from the search engine.

1. Create a `DDS-Router` directory and download the `.repos` file that will be used to install *DDS Router* and its dependencies:

```
mkdir <path\to\user\workspace>\DDS-Router
cd <path\to\user\workspace>\DDS-Router
mkdir src
wget https://raw.githubusercontent.com/eProsima/DDS-Router/main/ddsrouter.repos
vcs import src < ddsrouter.repos
```

Note: In case there is already a *Fast DDS* installation in the system it is not required to download and build every dependency in the `.repos` file. It is just needed to download and build the *DDS Router* project having sourced its dependencies. Refer to section *eProsima dependencies* in order to check how to source *Fast DDS* library.

2. Build the packages:

```
colcon build
```

Note: Being based on CMake, it is possible to pass the CMake configuration options to the `colcon build` command. For more information on the specific syntax, please refer to the CMake specific arguments page of the colcon manual.

3.27.3 CMake installation

This section explains how to compile *DDS Router* with CMake, either *locally* or *globally*.

Local installation

1. Open a command prompt, and create a `DDS-Router` directory where to download and build *DDS Router* and its dependencies:

```
mkdir <path\to\user\workspace>\DDS-Router
mkdir <path\to\user\workspace>\DDS-Router\src
mkdir <path\to\user\workspace>\DDS-Router\build
cd <path\to\user\workspace>\DDS-Router
wget https://raw.githubusercontent.com/eProsima/DDS-Router/main/ddsrouter.repos
vcs import src < ddsrouter.repos
```

2. Compile all dependencies using CMake.

   - Foonathan memory

```
cd <path\to\user\workspace>\DDS-Router
mkdir build\foonathan_memory_vendor
cd build\foonathan_memory_vendor
cmake <path\to\user\workspace>\DDS-Router\src\foonathan_memory_vendor -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DBUILD_SHARED_LIBS=ON
cmake --build . --config Release --target install
```

- Fast CDR

```
cd <path\to\user\workspace>\DDS-Router
mkdir build\fastcdr
cd build\fastcdr
cmake <path\to\user\workspace>\DDS-Router\src\fastcdr -DCMAKE_INSTALL_
↪PREFIX=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install
```

- Fast DDS

```
cd <path\to\user\workspace>\DDS-Router
mkdir build\fastdds
cd build\fastdds
cmake <path\to\user\workspace>\DDS-Router\src\fastdds -DCMAKE_INSTALL_
↪PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install
```

- Dev Utils

```
# CMake Utils
cd <path\to\user\workspace>\DDS-Router
mkdir build\cmake_utils
cd build\cmake_utils
cmake <path\to\user\workspace>\DDS-Router\src\dev-utils\cmake_utils -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install

# C++ Utils
cd <path\to\user\workspace>\DDS-Router
mkdir build\cpp_utils
cd build\cpp_utils
cmake <path\to\user\workspace>\DDS-Router\src\dev-utils\cpp_utils -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install
```

- DDS Pipe

```
# ddspipe_core
cd <path\to\user\workspace>\DDS-Router
mkdir build\ddspipe_core
cd build\ddspipe_core
```

(continues on next page)

(continued from previous page)

```
cmake <path\to\user\workspace>\DDS-Router\src\ddspipe\ddspipe_core -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install

# ddspipe_participants
cd <path\to\user\workspace>\DDS-Router
mkdir build\ddspipe_participants
cd build\ddspipe_participants
cmake <path\to\user\workspace>\DDS-Router\src\ddspipe\ddspipe_
↪participants -DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-
↪Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install

# ddspipe_yaml
cd <path\to\user\workspace>\DDS-Router
mkdir build\ddspipe_yaml
cd build\ddspipe_yaml
cmake <path\to\user\workspace>\DDS-Router\src\ddspipe\ddspipe_yaml -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install
```

3. Once all dependencies are installed, install *DDS Router*:

```
# ddsrouter_core
cd <path\to\user\workspace>\DDS-Router
mkdir build\ddsrouter_core
cd build\ddsrouter_core
cmake <path\to\user\workspace>\DDS-Router\src\ddsrouter\ddsrouter_core ^
    -DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install -DCMAKE_
↪PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install

# ddsrouter_yaml
cd <path\to\user\workspace>\DDS-Router
mkdir build\ddsrouter_yaml
cd build\ddsrouter_yaml
cmake <path\to\user\workspace>\DDS-Router\src\ddsrouter\ddsrouter_yaml -DCMAKE_
↪INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install

# ddsrouter_tool
cd <path\to\user\workspace>\DDS-Router
mkdir build\ddsrouter_tool
cd build\ddsrouter_tool
cmake <path\to\user\workspace>\DDS-Router\src\ddsrouter\tools\ddsrouter_tool -
↪DCMAKE_INSTALL_PREFIX=<path\to\user\workspace>\DDS-Router\install ^
    -DCMAKE_PREFIX_PATH=<path\to\user\workspace>\DDS-Router\install
cmake --build . --config Release --target install
```

Note: By default, *DDS Router* does not compile tests. However, they can be activated by downloading and installing Gtest and building with CMake option `-DBUILD_TESTS=ON`.

Global installation

To install *DDS Router* system-wide instead of locally, remove all the flags that appear in the configuration steps of `foonathan_memory_vendor`, `Fast-CDR`, `Fast-DDS`, and `DDS-Router`

3.27.4 Run an application

If the *DDS Router* was compiled using colcon, when running an instance of a *DDS Router*, the colcon overlay built in the dedicated `DDS-Router` directory must be sourced. There are two possibilities:

- Every time a new shell is opened, prepare the environment locally by typing the command:

```
setup.bat
```

- Add the sourcing of the colcon overlay permanently, by opening the *Edit the system environment variables* control panel, and adding `~/Fast-DDS/install/setup.bat` to the `PATH`.

However, when running an instance of a *DDS Router* compiled using CMake, it must be linked with its dependencies where the packages have been installed. This can be done by opening the *Edit system environment variables* control panel and adding to the `PATH` the *DDS Router*, *Fast DDS* and *Fast CDR* installation directories:

- *Fast DDS*: C:\Program Files\fastrtps
- *Fast CDR*: C:\Program Files\fastcdr
- *DDS Router*: C:\Program Files\ddsrouter

3.28 CMake options

eProsima DDS Router provides numerous CMake options for changing the behavior and configuration of *DDS Router*. These options allow the developer to enable/disable certain *DDS Router* settings by defining these options to `ON/OFF` at the CMake execution, or set the required path to certain dependencies.

Warning: These options are only for developers who installed *eProsima DDS Router* following the compilation steps described in *Linux installation from sources*.

Option	Description	Possible values	Default
CMAKE_BUILD_TYPE	CMake optimization build type.	Release Debug	Release
BUILD_ALL	Build the *DDS Router* application tool *ddsrouter*. Setting to ON sets to ON BUILD_TOOL, BUILD_LIBRARY, and BUILD_DOCS.	OFF ON	OFF
BUILD_LIBRARY	Build the *DDS Router* documentation. It is set to ON if BUILD_ALL is set to ON.	OFF ON	ON
BUILD_TOOL	Build the *DDS Router* documentation. It is set to ON if BUILD_ALL is set to ON.	OFF ON	ON
BUILD_DOCS	Build the *DDS Router* documentation. It is set to ON if BUILD_ALL is set to ON.	OFF ON	OFF
BUILD_TESTS	Build the *DDS Router* application and documentation tests. Setting BUILD_TESTS to ON sets BUILD_ALL, BUILD_LIBRARY_TESTS, BUILD_TOOL_TESTS, and BUILD_DOCS_TESTS to ON.	OFF ON	OFF
BUILD_LIBRARY_TESTS	Build the *DDS Router* library tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
BUILD_TOOL_TESTS	Build the *DDS Router* application tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
BUILD_DOCS_TESTS	Build the *DDS Router* documentation tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
LOG_INFO	Activate *DDS Router* execution logs. It is set to ON if CMAKE_BUILD_TYPE is set to Debug.	OFF ON	ON if Debug OFF otherwise
ASAN_BUILD	Activate address sanitizer build.	OFF ON	OFF
TSAN_BUILD	Activate thread sanitizer build.	OFF ON	OFF

3.29 Version v2.2.0

This release includes the following **Features**:

- Publish the *Logs* in a DDS topic.
- New *Monitor* module.

This release includes the following **Configuration features**:

- New `logging` configuration option to configure the DDS Router *internal logs*.

This release includes the following **Dependencies Update**:

	Repository	Old Version	New Version
Foonathan Memory Vendor	eProsima/foonathan_memory_vendor	v1.3.1	v1.3.1
Fast CDR	eProsima/Fast-CDR	v2.1.2	v2.2.0
Fast DDS	eProsima/Fast-DDS	v2.13.1	v2.14.0
Dev Utils	eProsima/dev-utils	v0.5.0	v0.6.0
DDS Pipe	eProsima/DDS-Pipe	v0.3.0	v0.4.0

3.30 Previous Versions

3.30.1 Version v2.1.0

This release includes the following **Features**:

- *Forwarding Routes*.
- *Remove Unused Entities*.
- *Manual Topics*.
- *Max Transmission Rate*.
- *Max Reception Rate*.
- *Downsampling*.
- *Discovery Trigger*.
- Rename the *max-depth* under the *specs* tag to *history-depth*.
- Set *app_id* and *app_metadata* attributes in *eProsima DDS Router* participants.
- Support both Fast CDR v1 and v2.

This release includes the following **Bugfixes**:

- Save the instance handle data for keyed topics.

This release includes the following **Dependencies Update**:

	Repository	Old Version	New Version
Foonathan Memory Vendor	eProsima/foonathan_memory_vendor	v1.3.1	v1.3.1
Fast CDR	eProsima/Fast-CDR	v1.1.0	v2.1.2
Fast DDS	eProsima/Fast-DDS	v2.11.0	v2.13.1
Dev Utils	eProsima/dev-utils	v0.4.0	v0.5.0
DDS Pipe	eProsima/DDS-Pipe	v0.2.0	v0.3.0

3.30.2 Version v2.0.0

This release adds new **Requirements**:

- DDS Pipe project.

This release has the following **Features**:

- Main functionality has been moved to new repository DDS Pipe to reuse it in other projects.
- *XML Participant*
- Support for DDS Security.

This release includes the following **Internal Implementation Features**:

- Support *Interface Whitelisting*.
- Support *Custom Transport Descriptors* (UDP or Shared Memory only) for Simple Participants.
- Support *Ignore Participant Flags* for Simple Participants.
- Add tests for Dynamic Types to ROS 2.

- New `CommonParticipant` class for a Fast DDS Participant wrapper.
- New `XmlParticipant` class for a Fast DDS Participant in which the participant is created following a profile name loaded by XML configuration.
- New `CommonReader` class for a Fast DDS Data Reader wrapper.
- New `SimpleReader` class for a Fast DDS Data Reader that implements abstract `CommonReader`.
- New `SpecificQoSReader` class for a Fast DDS Data Reader with specific QoS policies.
- New `CommonWriter` class for a Fast DDS Data Writer wrapper.
- New `SimpleWriter` class for a Fast DDS Data Writer that implements abstract `CommonWriter`.
- New `QoSSpecificWriter` class for a Fast DDS Data Writer with specific QoS policies.
- New `MultiWriter` class for a Fast DDS Data Writer collection that contains multiple `QoSSpecificWriter`.

This release includes the following **Bugfixes**:

- Add TSAN fixes.
- Include missing DLLs.
- Resolve Windows warnings.
- Restore default DomainParticipantQoS after creating and enabling `DynTypesParticipant`.

This release includes the following **Dependencies Update**:

	Repository	Old Version	New Version
Foonathan Memory Vendor	eProsima/foonathan_memory_vendor	v1.3.0	v1.3.1
Fast CDR	eProsima/Fast-CDR	v1.0.27	v1.1.0
Fast DDS	eProsima/Fast-DDS	v2.10.1	v2.11.0
Dev Utils	eProsima/dev-utils	v0.3.0	v0.4.0
DDS Pipe	eProsima/DDS-Pipe	v0.1.0	v0.2.0

3.30.3 Version v1.2.0

This release includes the following **CI improvements**:

- Add blackbox tests for repeater in WAN case.
- Upgrade deprecated actions.

3.30.4 Version v1.1.0

This release includes the following **features**:

- New *Log Verbosity Argument* and *Log Filter Argument* arguments to configure *DDS Router* and *Fast DDS* logs shown in application.

This release includes the following **CI improvements**:

- Add *Address Sanitizer* check to all tests.
- Add *Thread Sanitizer* check to all tests.
- Compile with *-Wall* flag all CI jobs.

3.30.5 Version v1.0.0

This is the first **stable release** of eProsima *DDS Router*.

This release includes the following **major features**:

- New **QoS Transparency Module**.

This release includes the following **features**:

- New *Repeater behaviour*, a new configuration to create *TURN* servers.
- Add RPC support (compatibility with *ROS 2* services).
- Timeout argument `--timeout` to close the application after the time set has passed. Check section *Timeout Argument* for more information.
- Add TLS-SNI support.
- Add **external port** support so internal and external port in network router configuration could be different.

This release includes the following **DDS features**:

- New *Initial Peers* Participant that allows to create a fast WAN communication avoiding the Discovery Server configuration.
- Participants names are set from ParticipantIds.

This release includes the following **configuration changes**:

- New Configuration Version `v3.0` that allow Repeater and initial peers configuration.
- Yaml Validator only validates `v3.0` configurations.
- Address supports an `external-port` value to set this in a listening address.

This release includes the following **major changes**:

- `wan` Participant Kind uses now Initial Peers Discovery Protocol, while Discovery Server requires a new Participant Kind `wan-discovery-server`.
- Internal packages for utils as `ddsrouter_cmake` and `ddsrouter_utils` have been moved to an external repository.

This release includes the following **performance improvements**:

- No locking in the `Track` hot-path.

3.30.6 Version v0.4.0

This release includes the following **features**:

- New *YAML Validator*, a simple tool to assert the correctness of DDS Router configuration files.
- New *Version Argument* to show the current version of DDS Router.

This release includes the following **improvementes**:

- New `FastPayloalPool` class that will allow unblocking access to shared payloads stored by FastDDS and DDS Router. This slightly improves the **performance** of the router.
- New `ThreadPool` class for handling parallel processing. Limit the number of threads spawned by the DDS Router, improving the performance of the application. The user can set-up this parameter on the YAML configuration *Number of Threads*.

This release includes the following **major changes**:

- New auxiliary package `cmake_utils` to implement general CMake functions and tools.
- The internal package `ddsrouter_event` is joint within the internal package `cpp_utils`.

This release includes the following **minor changes**:

- Refactor all `enum` to `enum class`.

This release includes the following **Continuous-Integration features**:

- New CI workflow to build dependencies nightly. This allows to run the CI faster, as the build is only done for the ddsrouter packages and not the whole fastdds.
- Add `ubuntu 22.04` to platforms in CI.
- Add `windows 19` to platforms in CI.

This release fixes the following **minor bugs**:

- Fix race condition occurred when handling signals (due to concurrent access to *SignalManager* singleton).
- Bug in TLS Configuration initialization when setting server configuration files.
- Set FastDDS DomainParticipants and Readers listeners after creation to avoid race conditions in entities creation.

3.30.7 Version v0.3.0

This release includes the following **major changes**:

- New DDS Router library that provides the DDS Router features through a C++ API.
- Division of DDS Router application into several packages.
  - *ddsrouter_event*: C++ library which implements System Operating (SO)-dependent signal handlers.
  - *cpp_utils*: C++ library which implements various utility functions.
  - *ddsrouter_core*: C++ library which implements the DDS Router operation and exports the DDS Router C++ API.
  - *ddsrouter_yaml*: C++ library to parse the DDS Router *yaml* configuration files.
  - *ddsrouter_tool*: DDS Router end-user application.
- New dynamic topic discovery feature to automatically discover DDS Topics in a DDS network.
- Allow using wildcard characters to define topics in `allowlist` and `blocklist`.
- Build internal topic tracks via the `builtin-topics` configuration.

This release includes the following **Routing features**:

- Apply DDS Topic filtering rules using the `allowlist` and `blocklist` lists.

This release includes the following **User Interface features**:

- Upgrade the *yaml* configuration file to version 2.0 which breaks compatibility with version 1.0.
- Support for both version 1.0 and version 2.0 of the *yaml* configuration file, maintaining version 1.0 by default.
- Improve reporting of errors resulting from parsing a malformed *yaml* configuration file.

This release includes the following **Continuous-Integration features**:

- Add tests for the *yaml* parsing library (*ddsrouter_yaml*).
- Specific testing GitHub actions for each DDS Router package for both Windows and Linux platforms.

This release fixes the following **major bugs**:

- Fix deadlock between Track and Fast DDS Reader mutex.
- Support any size for in and out messages.

This release fixes the following **minor bugs**:

- Fix parsing of *reload-time* executable argument.
- Handle signals in dedicated threads to prevent hangs when terminating execution.
- Fix rare deadlock in EventHandler when destroying and callback called.

This release includes the following **Documentation features**:

- DDS Router execution from a provided Docker image.
- Update all examples of *yaml* configuration files to be consistent with the new *yaml* configuration version.
- High-level repository structure description and developer contribution guidelines.
- Improved and extended *Topic Filtering* section according to the new dynamic topic discovery functionality.

3.30.8 Version v0.2.0

This release includes the following **Configuration features**:

- Support TLS over TCP configuration and communication.
- Support IPv6 communication via UDP, TCP and TLS over TCP.
- Support DNS by given Domain Name in configuration instead of an IP address.
- Support keyed topics.

This release includes the following **Routing features**:

- Zero-Copy data transmission between internal Participants.

This release includes the following **User Interface features**:

- Shutdown the DDS Router application gracefully sending `SIGTERM` (`kill`) or `SIGINT` (`^C`) signals.

This release includes the following **Continuous-Integration features**:

- Add communication tests for UDP, TCP and TLS over TCP WAN cases.
- Extend tool test with more complex configurations.
- Remove Flaky tests from CI required passing tests.
- Implement a new class to check that no warning or error logs are produced during test executions.
- Add gMock to test libraries.

This release fixes the following **major bugs**:

- Fix GUID creation when explicit guid is provided.
- Show error when participant ids are duplicated.

This release fixes the following **minor bugs**:

- Change `YAML` example configurations to `YAML` format (instead of `JSON`) fixing an issue when blank lines were missing at the end of the file.
- Normalize the error and process exit when failure.
- Fix documentation typos.

3.30.9 Version v0.1.0

This is the first release of eProsima *DDS Router*.

This release includes several **features** regarding the routing of DDS data, *the DDS Router* configuration, the user interaction with the *DDS Router*, and the different DDS configurations that the application is able to reproduce.

This release includes the following **User Interface features**:

- Application executable.
- Application executable arguments.
- Signal handler to close the application.
- FileWatcher thread to watch and reload the configuration file.
- Periodic timer to force reload configuration.
- Application run-time user logs.
- Application run-time debug logs.
- Error handling:
  - Error log and exit program when reading configuration fails.
  - Error log and exit program when initializing Participants fails.
  - Error log and continuing execution when execution error occurs.

This release includes the following **Configuration features**:

- Allow to execute the application with a *YAML* configuration file.
- Support for initial topics in allowlist.
- Support for block topic filters.
- Different Participant configurations:
  - Domain Id.
  - Discovery Server GuidPrefix.
  - Listening addresses.
  - Connection addresses.

This release includes the following **Routing features**:

- Support for routing Topics specified in allowlist regarding Topic name and Topic Type name.
- Support for connecting to new Topics in run-time (by reloading configuration).
- Support for disabling a Topic in run-time.
- Support for enabling a Topic that has been disabled in run-time.
- Route messages of each Participant to all the other Participants.
- Agnostic to topic data types.

This release includes the following **DDS features**:

- Allow UDP, TCP and SHM transport communication.
- Allow dynamic discovery of new entities.
- Using *eProsima Fast DDS* RTPS layer for discovery, publication and subscription.

This release includes the following **Participant features**:

- **Echo Participant**.
- **Simple Participant**, able to connect to a Simple Discovery UDP DDS network.
- **Local Discovery Server Participant**, able to connect to a local Discovery Server as Client or Server.
- **WAN Participant**, able to connect to a WAN Discovery Server network as Client or Server.

This release includes the following **Examples**:

- **Echo Example**, to monitor a local simple network.
- **Domain Change Example**, to connect two different domains.
- **ROS 2 Discovery Server Example**, to connect a regular DDS network with a Discovery Server network using ROS 2 configuration.
- **WAN Example**, to connect two DDS networks in different LANs.

This release includes the following **Documentation features**:

- This same documentation.

3.31 Glossary

LAN **Local Area Network**

NAT **Network Address Translation**: Typically an internet router multiplexes all the traffic through a public IP to several private IPs. Usually, the machines under the router network cannot be accessed from the outside unless a Port is forwarded in the router configuration, or if such host has previously started a TCP communication with the message source.

QoS **Quality of Service**: Collection of attributes and settings that configure the behavior of DDS communications.

QoS Profile Specific QoS that is related with a name, called *profile*. This is used to create several entities with the same QoS, or to specify which already established QoS an entity should use.

RPC **Remote Procedural Call**: Client-Server protocol that can be run over DDS. ROS2 services and actions use this kind of protocol.

TCP **Transmission Control Protocol**

TURN **Traversal Using Relays around NAT** A TURN server is a network element that supports a common point of redirection of messages. It redirect messages between N nodes connected to it. This is a very common technique in order to traverse a NAT or an unreachable network.

UDP **User Datagram Protocol**

WAN **Wide Area Network**

INDEX

D

DataReader, 55
DataWriter, 55
Discovery Server, 55
Domain Id, 55
DomainParticipant, 55

E

Endpoint, 55

G

Guid, 55
GuidPrefix, 55

I

Initial Peers, 56

L

LAN, 111

N

NAT, 111

P

Participant, 55
Participant Kind, 55
Participant Name, 55
Payload, 55

Q

QoS, 111
QoS Profile, 111

R

RPC, 111

T

TCP, 111
Topic, 56
TURN, 111

U

UDP, 111

W

WAN, 111