DDS Router Documentation

Release ..

eProsima

Jul 06, 2022

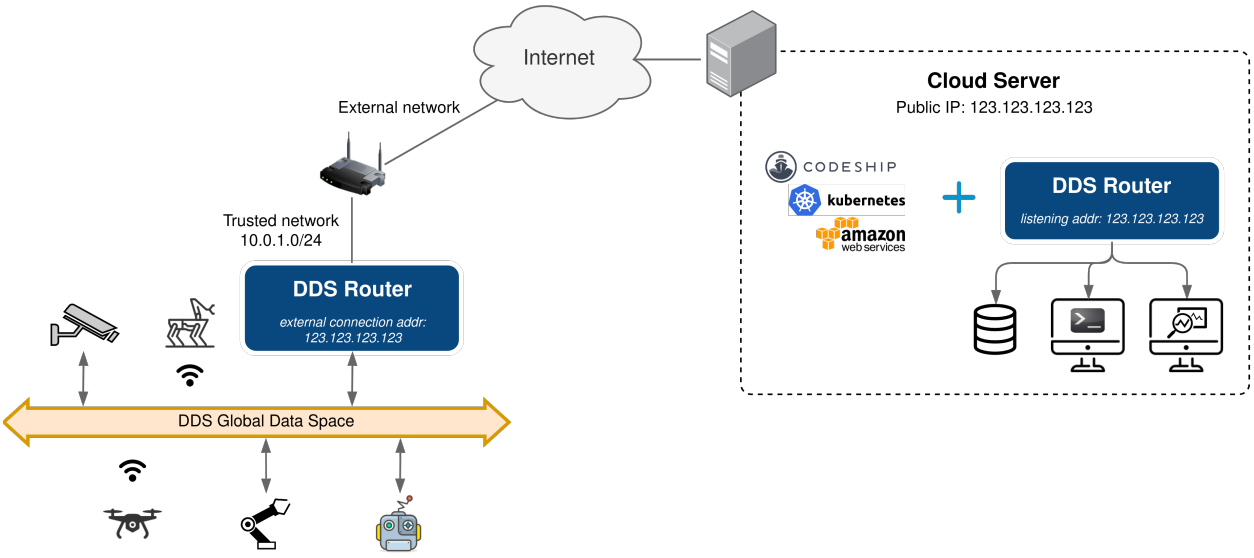
INTRODUCTION

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.



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.

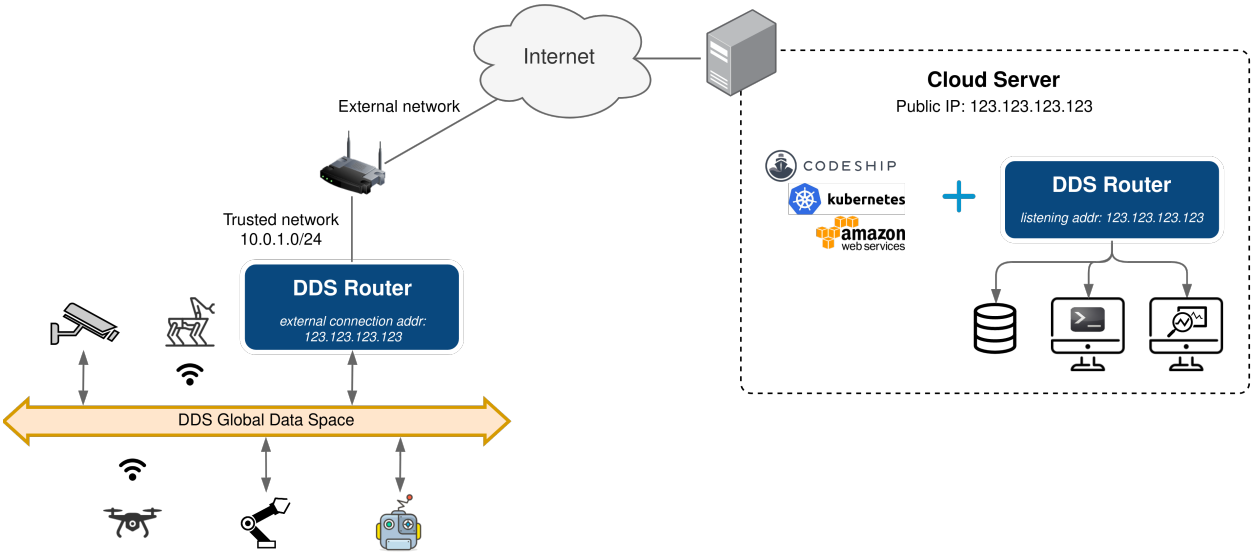
Furthermore, *DDS Router* is a software designed for various forms of distributed networks, such as mesh networks in which nodes are deployed in different private local networks that are auto-discovered without any centralized network node, or cloud-based networks where there is a data processing cloud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



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 20.04 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 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.

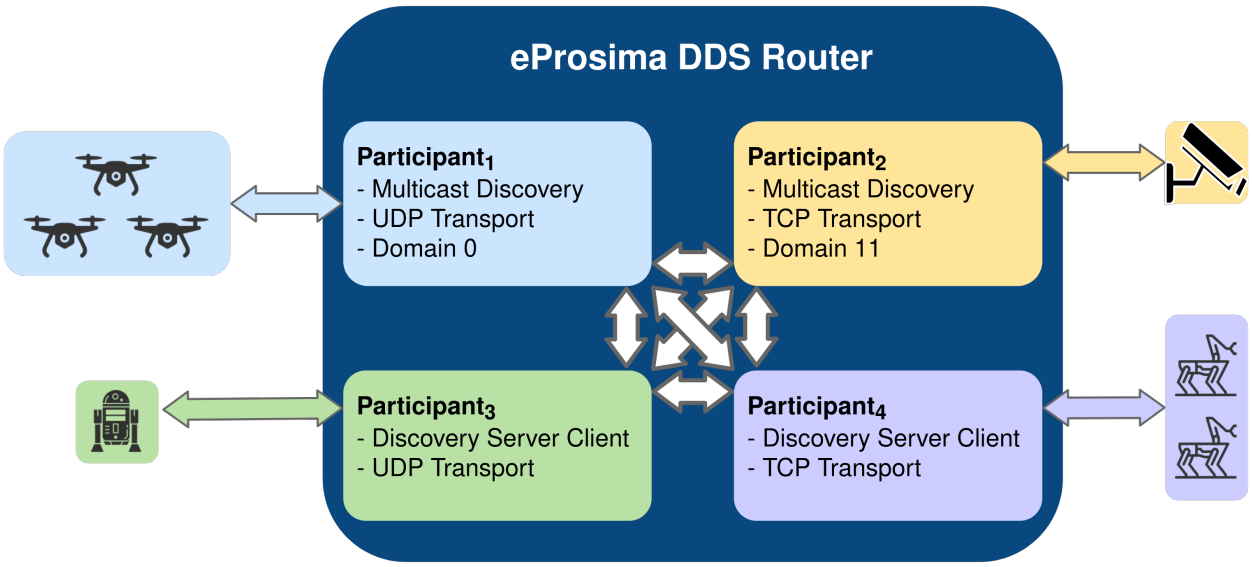
The *DDS Router* is an application that internally run *Participants*, which are an abstraction of DDS *DomainParticipants*. Each one of these Participants is an communication interface, a "door" to a specific DDS network configuration. These Participants allow the application to connect to different 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.

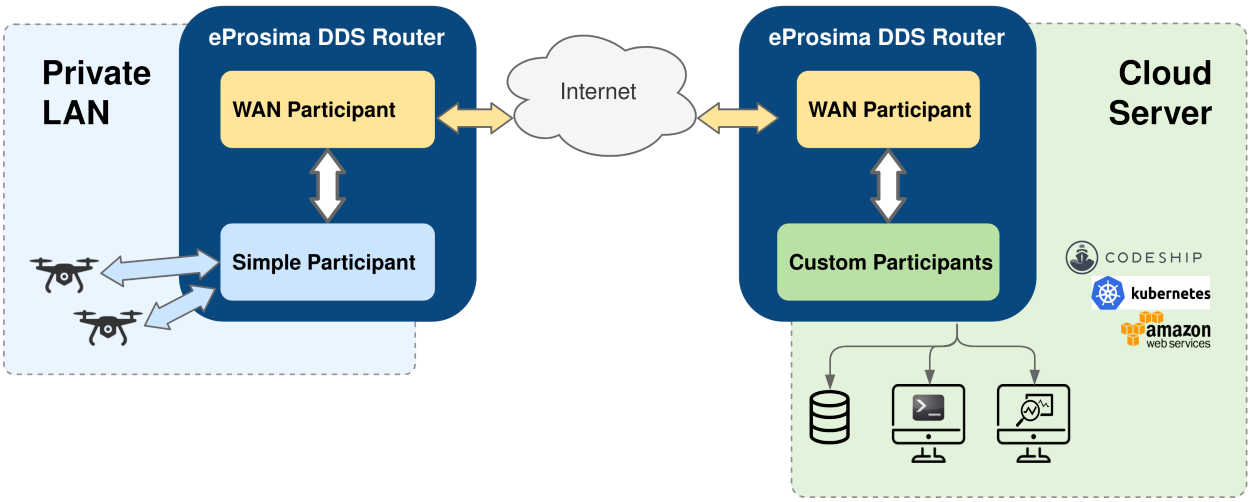
3.8.1 WAN Communication

To achieve 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 to each other using DDS over WAN, and will route 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 would belong to the same LAN.

Another important feature is that WAN communications is not limited to a single pair of *DDS Router*. The WAN communication is 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 works as a standard



DDS node, publishing and subscribing in the shared DDS topics. This allows to create a non limited 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* to check how to configure a DDS Router) (see section *section* to check the application supported arguments).
- **INTERACT**: Once the *DDS Router* application is running, the topics involved in this communication could 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

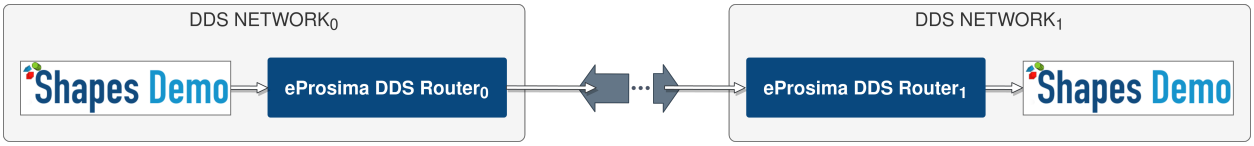
The different cases where the *DDS Router* could be applied are very varied, and would increase as new Participant Kinds will be added in future releases. These are most common use cases so far:

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 means of a pair of routers, allowing for the connection between 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.

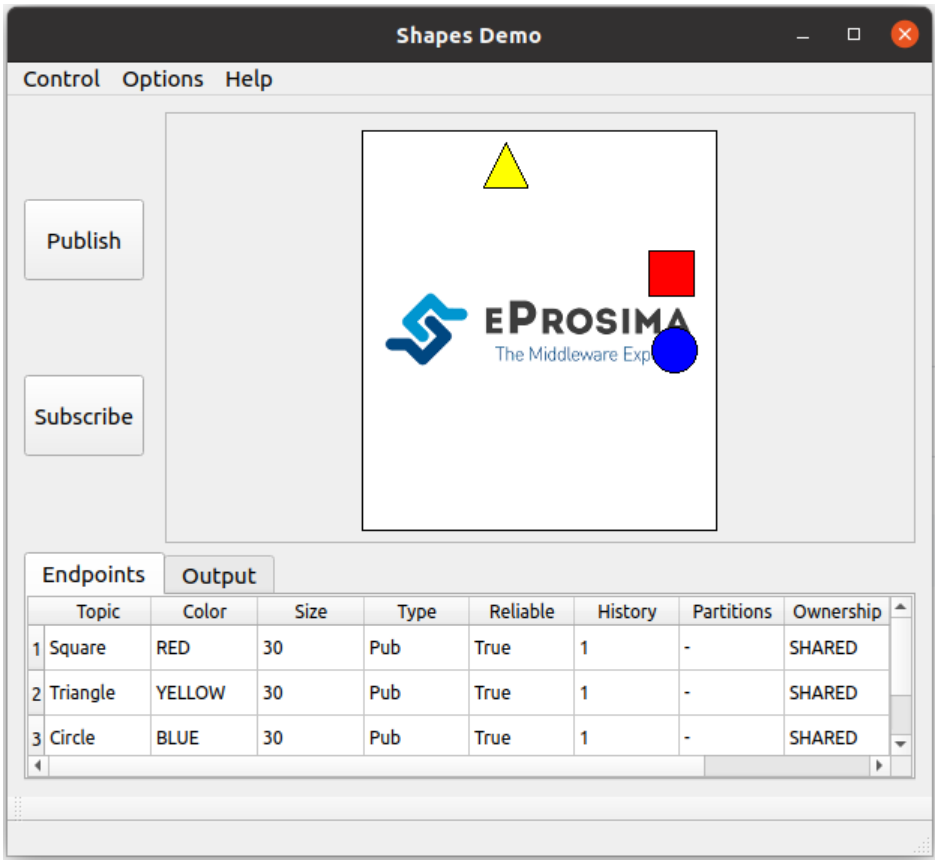
3.9.1 Launching ShapesDemo

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
```

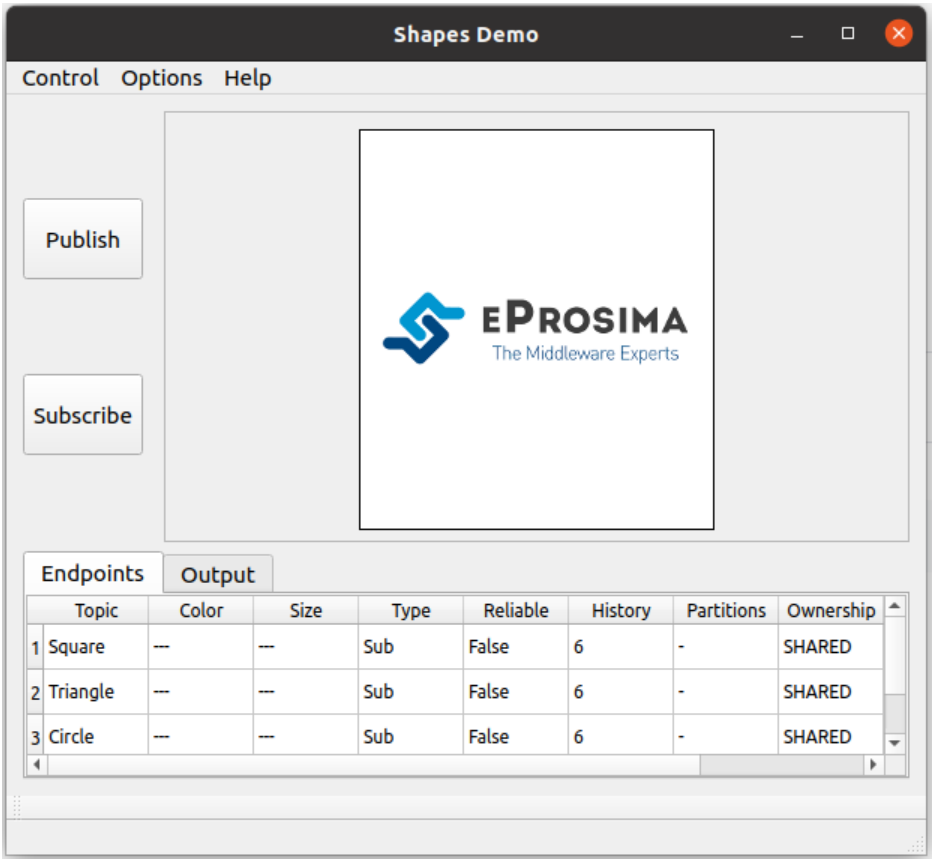
Apart from selecting on which topics we wish to send/receive data, we must configure as well the participants that will ultimately perform communication. Each router instance will contain a *simple* and a *WAN* participants. In brief, the simple participants will be in charge of locally communicating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

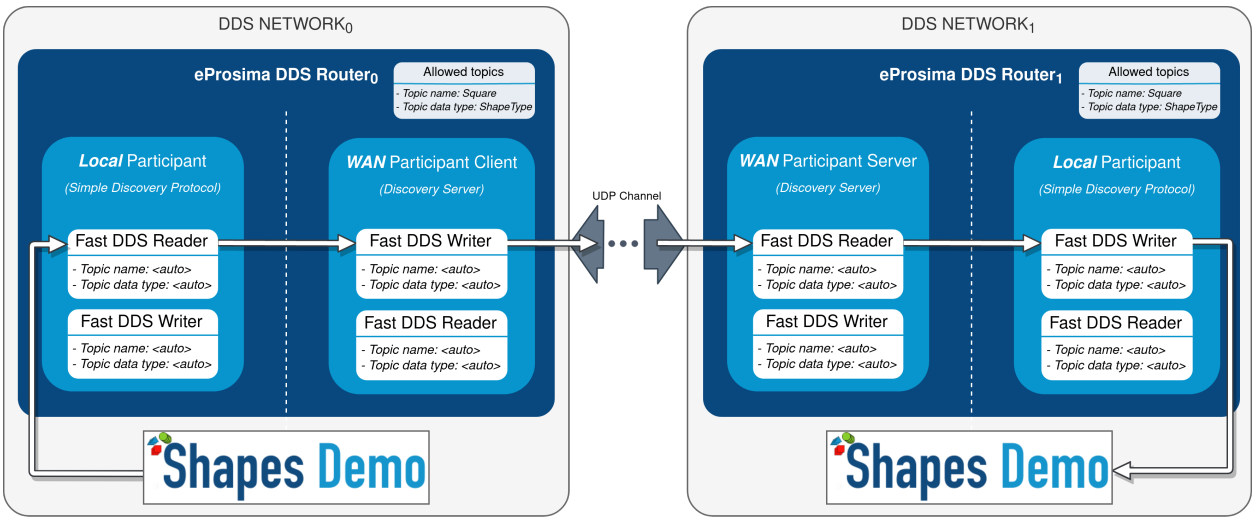
The only configuration required for simple participants is the DDS `domain` identifier.

```
  - 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
(Discovery Server)
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
(Discovery Server)
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`. Note that a unique `id` must be set for every WAN participant, and this entry should also be included in `connection 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: v2.0

allowlist:
  - name: Square

participants:

  - name: SimpleParticipant
    kind: local
    domain: 0

  - name: ClientWAN
    kind: wan
    discovery-server-guid:
      id: 1
    connection-addresses:
      - discovery-server-guid:
          id: 0
        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: v2.0

allowlist:
  - name: Square


participants:

  - name: SimpleParticipant
    kind: local
    domain: 1

  - name: ServerWAN
    kind: wan
    discovery-server-guid:
```

(continues on next page)

(continued from previous page)

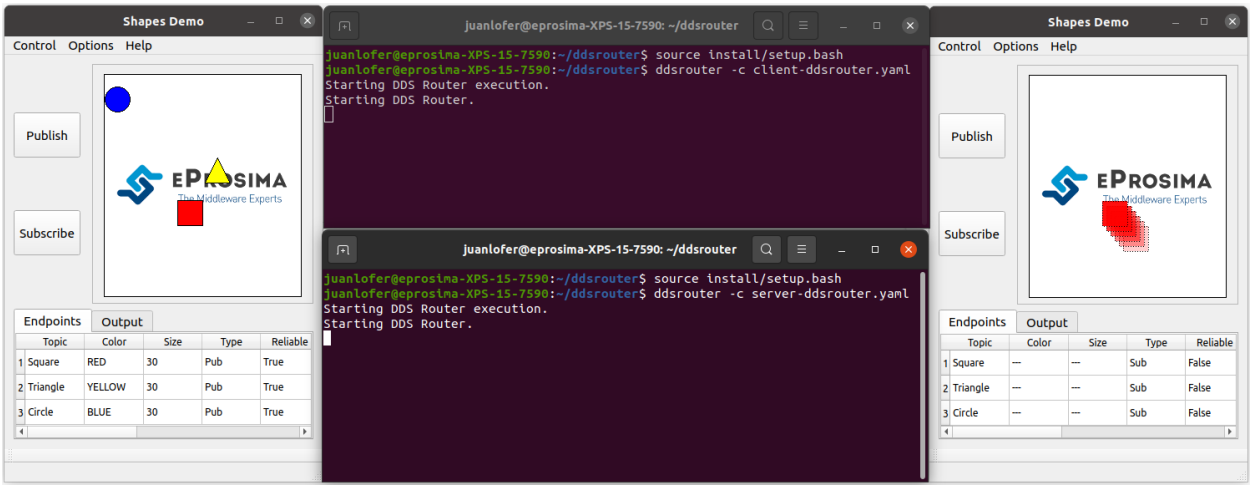
```
    id: 0
  listening-addresses:
    - ip: 192.168.1.8
      port: 11800
      transport: udp
```

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.

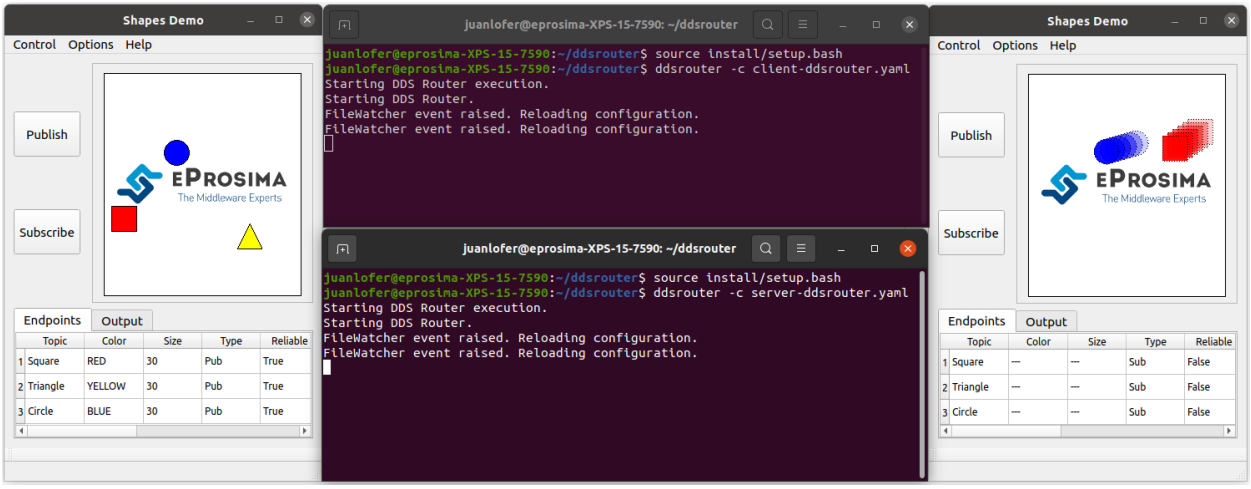


DDS Router supports the dynamic addition/deletion of topics at runtime (see *Reload Topics*). Let us test this feature by adding the circle topic to the allowlist of both routers. Also, by removing the square topic (removing this topic from one of the routers' allowlist will suffice) the square data 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 command line and configured through a YAML configuration file.

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`		
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`		

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:
  -h --help         Print this help message.
  -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 File Watcher
                    functionality is not available (e.g. config file is a symbolic link).
                    Value 0 does not reload file. [Default: 0].
  -d --debug        Activate debug Logs (be aware that some logs may require specific␣
↪CMAKE compilation options).
  -v --version      Print version, branch and commit hash.
```

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**.

Debug Argument

Activate `INFO` and `DEBUG` logs for the *DDS Router* execution. For this argument to work, the *DDS Router* must have been compiled with CMake option `CMAKE_BUILD_TYPE=Debug`, or compiled with CMake option `LOG_INFO=ON`.

Note: If this option is enabled and 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`.

Version Argument

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

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 *DDS Router 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 watches for 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 could 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 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. Write command `kill <pid>` in a different terminal, where `<pid>` is the id of the process running the *DDS Router*. Use `ps` or `top` programs to check the process ids.

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.

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, or set the Participant Name as the alias of the kind. 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:

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

```yaml
- name: simple         # New Participant with Name = 'simple' and 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`		Print in *stdout* every 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.
WAN Participant	`wan router`	`guid listening-addresses connection-addresses tls`	Discovery Server DDS DomainParticipant for WAN communication.

Echo Participant

This *Participant* prints in `stdout` every data that is received by the *DDS Router*. The resulted logs contain the echo *Participant Name*, the source *Endpoint Guid* that has originally generated the message, the *Topic* where this message has been received, and the *Payload* (in hexadecimal format) received. 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*.

```
Echo Participant: <participant_id> has received from Endpoint: <endpoint_guid> in topic:
↪<endpoint_topic> the following payload: <payload>
```

This is an example of a message received by a *DDS Router* in a Participant connected to a ROS2 `talker` and written by an **Echo Participant**:

```
Echo Participant: ParticipantId{echo} has received from Endpoint: 01.0f.b8.a8.2e.69.b1.
↪47.01.00.00.00|0.0.12.3 in topic: Topic{rt/chatter, std_msgs::msg::dds_::String_} the␣
↪following payload: <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}>
```

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. All the data received by any of the Participants of the router will be printed with timestamp, topic and source guid along with the payload.

Kind aliases

- `echo`

Configuration

Echo Participant does not allow any configuration.

Configuration Example

```yaml
- name: echo_participant     # Participant Name = echo_participant
  kind: echo
```

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`

Configuration

Local Discovery Server Participant allow configure the standard attributes of a Discovery Server.

- To configure the Discovery Server *GuidPrefix*, check the following section *Configuration section*.
- To configure the Discovery Server listening addresses, check the following section *Configuration section*.
- To configure the Discovery Server connection addresses to connect with other Discovery Servers, check the following section *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.

```yaml
- 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
      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
```

WAN Participant

This type of *Participant* refers to a *Discovery Server 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 relay from one to another depending on the topics filtered by each of them.

Kind aliases

- `wan`
- `router`

Configuration

WAN Discovery Server Participant allow configure the standard attributes of a Discovery Server.

- To configure the Discovery Server *GuidPrefix*, check the following section *Configuration section*.
- To configure the Discovery Server listening addresses, check the following section *Configuration section*.
- To configure the Discovery Server connection addresses to connect with other Discovery Servers, check the following section *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 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-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
```

(continues on next page)

(continued from previous page)

```
  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
```

3.12 DDS Router 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 files may be easily validated by using the *YAML Validator* tool.

3.12.1 Configuration version

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

Configuration Versions	String in `version` tag	*DDS Router* activation release
version 1.0	`v1.0`	*v0.1.0*
version 2.0	`v2.0`	*v0.2.0*

Current configuration version is v2.0. This is the configuration version that is described along this page.

Note: The current default version when tag `version` is not set is *v1.0*.

Warning: **Deprecation Warning**. In future releases tag `version` will be mandatory.

3.12.2 Number of Threads

The YAML Configuration support 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 characteristics. In case this value is not set, the default number of threads used is `12`.

Note: The number of threads set by this configuration does not affect the total number of thread used by the application. Internal library dependencies will create additional threads.

3.12.3 Topic Filtering

DDS Router allows filtering of DDS *Topics*, that is, it allows to define which DDS Topics are going to be relayed by the application. This way, it is possible to define a set of rules in *DDS Router* to filter those data samples the user does not wish to forward.

It is not mandatory to define such set of rules in the configuration file. In this case, a DDS Router will forward all the data published under the topics that it automatically discovers within the DDS network to which it connects.

To define these data filtering rules based on the Topics to which they belong, three lists are available:

- Allowed topics list (`allowlist`)
- Block topics list (`blocklist`)
- Builtin topics list (`builtin-topics`)

These three lists of topics listed above are defined by a tag in the *YAML* configuration file, which defines a *YAML* vector (`[]`). This vector contains the list of topics for each filtering rule. Each Topic is determined by its entries `name`, `type` and `keyed`, with only the first one being mandatory.

Topic entries	Data type	Default value
`name`	`string`	-
`type`	`string`	`"*"`
`keyed`	`bool`	Both `true` and `false`

The entry `keyed` determines whether the corresponding topic is keyed or not. See *Topic* section for further information about the topic.

Note: Tags `allowlist`, `blocklist` and `builtin-topics` must be at yaml base level (it must not be inside any other tag).

Note: Placing quotation marks around values in a YAML file is generally optional. However, values containing wildcard characters must be enclosed by single or double quotation marks.

Allow topic list (`allowlist`)

This is the list of topics that *DDS Router* will forward, i.e. the data published under the topics matching the expressions in the `allowlist` will be relayed by *DDS Router*.

Note: If no `allowlist` is provided, data will be forwarded for all topics (unless filtered out in `blocklist`).

Block topic list (`blocklist`)

This is the list of topics that the *DDS Router* will block, that is, all data published under the topics matching the filters specified in the `blocklist` will be discarded by the *DDS Router* and therefore will not be relayed.

This list takes precedence over the `allowlist`. If a topic matches an expression both in the `allowlist` and in the `blocklist`, the `blocklist` takes precedence, causing the data under this topic to be discarded.

Builtin topics list (`builtin-topics`)

DDS Router includes a mechanism to automatically detect which topics are being used in a DDS network. By automatically detecting these topics, a *DDS Router* creates internal DDS *Writers* and *Readers* for each topic and for each Participant in order to relay the data published on each discovered topic.

The discovery phase of the network topics can be accelerated by using the builtin topic list (`builtin-topics`). By defining topics in this list, the DDS router will create the DataWriters and DataReaders for these topics without waiting for them to be discovered. In this way, the initialization phase mentioned above is omitted and the application launching efficiency is improved.

Note that, for every topic contained in this list, both `name` and `type` must be specified and contain no wildcard characters. The entry `keyed` is optional, and defaults to `false`.

Examples of usage

The following is an example of how to use the `allowlist`, `blocklist` and `builtin-topics` configurations to setup the *DDS Router* filtering rules.

Dynamic topic discovery example

This example shows how the *DDS Router* is initially configured to forward the `rt/chatter` topic (default ROS 2 topic for `talker` and `listener`) with type name `std_msgs::msg::dds_::String_`, while the rest of the topics in the DDS network are expected to be dynamically discovered. Additionally, two rules are specified in the `blocklist` in order to filter out messages of no interest to the user.

```yaml
builtin-topics:
  - name: rt/chatter
    type: std_msgs::msg::dds_::String_

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

builtin-topics:
  - name: rt/chatter
    type: std_msgs::msg::dds_::String_
```

Allowlist and blocklist collision

In the following example, the `HelloWorldTopic` topic is both in the `allowlist` and (implicitly) in the `blocklist`, so according to the `blocklist` preference rule this topic is blocked. Moreover, only the topics present in the allowlist are relayed, regardless of whether more topics are dynamically discovered in the DDS network. In this case the forwarded topics are `AllowedTopic1` and `AllowedTopic2`.

```yaml
allowlist:
  - name: AllowedTopic1
    type: Allowed
  - name: AllowedTopic2
    type: "*"
  - name: HelloWorldTopic
    type: HelloWorld

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

3.12.4 Participant Configuration

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

3.12.5 Domain Id

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

```
domain: 101
```

3.12.6 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.
- *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 unique name to ask DNS server for the related IP. This field is ignored if `ip` is specified.

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

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

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

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

domain: localhost             # DNS call return value = 127.0.0.1
port: 33333
```

3.12.7 Discovery Server GuidPrefix

A *Discovery Server* requires a DDS *GuidPrefix* in order to other Participants 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, so it could be used using 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 allow an `id` value to specify which id is used in the GuidPrefix. 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
```

3.12.8 Discovery Server Listening Addresses

Tag `listening-addresses` configures the network addresses where the Discovery Server configured is going to listen for remote clients or servers. `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: 11668
    transport: tcp
```

3.12.9 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*.

```yaml
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
```

3.12.10 General Example

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

```yaml
# Version Latest
version: v2.0

# 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


participants:

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

# Simple DDS Participant in domain 3

  - name: Participant0              # Participant Name = Participant0

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

    domain: 3                       # DomainId = 3
```

(continues on next page)

(continued from previous page)

```
####################

# 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

    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
        transport: udp              # Transport = UDP

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

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

  - name: Wan                       # Participant Name = Wan ; if kind is not specified,␣
↪this value is used to determine the Participant Kind

    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: tcp          # Transport = TCP
```

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.

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.

3.13.2 TCP vs UDP

TCP and *UDP* are two well known network transport protocols. Both have their advantages and disadvantages regarding the scenario. These are a list of tips to help choosing whether to use one or the other.

Feature	UDP	TCP
Communication speed	Fast	Slower
Reliability	No Transport Layer reliability (could has DDS reliability)	Transport Layer reliability (duplicated if DDS reliability is used)
Port Forwarding	Require both sides of the communication to have ports forwarded from the router.	Require only server side of the communication to have port 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.

TLS

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

Tag	Requiredness	Description
`ca`	Mandatory for TLS servers and clients	Path to the CA (Certification- Authority) file.
`password`	Optional for TLS servers	Password of the `private_key` file.
`private_key`	Mandatory for TLS servers	Path to the private key certificate file.
`cert`	Mandatory for TLS servers	Path to the public certificate chain file.
`dh_params`	Mandatory for TLS servers	Path to the Diffie-Hellman parameters file.

Note: Although in principle only required for TLS clients, the CA (Certification- Authority) file must also be provided for TLS servers, as they might assume the client role when connecting to other participants configured as servers.

3.13.3 Examples

TCP Port Forwarding Example

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

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` (it is required to use the same public port as the internal one). User *A* should set its *listening-addresses* as follows:

```yaml
- name: WANServerParticipant_userA
  kind: wan

  discovery-server-guid:
    id: 2                               # Id to generate the GuidPrefix of the Discovery␣
↪Server of A
  listening-addresses:
    - ip: 1.1.1.1                       # Public IP of host Ha
      port: 11666                       # Port forwarded router Ra
      transport: tcp                    # Transport protocol
```

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

```yaml
- name: WANClientParticipant_userB
  kind: wan

  discovery-server-guid:
    id: 3                               # Must be different than A one
  connection-addresses:
    - discovery-server-guid:
        id: 2                           # Id of the Discovery Server of A
      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 be the scenario where user A host H_A has a private IP `192.168.1.2` given by network router R_A, with a public IP `1.1.1.1`. Let user B with host H_B has a private IP `192.168.2.2` given by network router R_B, with a 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` (it is required to use same public port as the internal one). User A should set its *listening-addresses* as follows:

```yaml
- name: WANServerParticipant_userA
  kind: wan

  discovery-server-guid:
    id: 2                               # Id to generate the GuidPrefix of the Discovery␣
↪Server of A
  listening-addresses:
    - ip: 1.1.1.1                       # Public IP of host Ha
      port: 11666                       # Port forwarded router Ra
```

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` (It is necessary to use same public port as the internal one). User B should set its *listening-addresses* and *connection-addresses* as follows:

```yaml
- name: WANClientParticipant_userB
  kind: wan

  discovery-server-guid:
    id: 3                               # Must be different than A one
  listening-addresses:
    - ip: 2.2.2.2                       # Public IP of host Hb
      port: 11777                       # Port forwarded router Rb
  connection-addresses:
    - discovery-server-guid:
        id: 2                           # Id of the Discovery Server of A
      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:

```yaml
- name: TLS_Server
  kind: wan

  discovery-server-guid:
    id: 0
  listening-addresses:
```

(continues on next page)

(continued from previous page)

```
    - 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

  discovery-server-guid:
    id: 1
  connection-addresses:
    - discovery-server-guid:
        id: 0
      addresses:
        - ip: 1.1.1.1
          port: 11666
          transport: tcp

  tls:
    ca: ca.crt
```

You may also have a look at <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 *DDS Router 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 `2.0` 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 over 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 belong 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 publish data in a specific Topic. It belong 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.

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`.

```yaml
###################################
# CONFIGURATION VERSION
version: v2.0                                                    # 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

  - name: EchoParticipant                                        # 6
    kind: echo                                                   # 7
```

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 does not require further configuration than name and kind.

```
  - name: EchoParticipant                                    # 6
    kind: echo                                               # 7
```

3.16.2 Execute example

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

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:

```
Echo Participant: ParticipantId{EchoParticipant} has received from Endpoint: 01.0f.44.59.
↪e6.de.2a.c8.01.00.00.00|0.0.1.3 in topic: Topic{HelloWorldTopic, HelloWorld} the␣
↪following payload: <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}>
Echo Participant: ParticipantId{EchoParticipant} has received from Endpoint: 01.0f.44.59.
↪e6.de.2a.c8.01.00.00.00|0.0.1.3 in topic: Topic{HelloWorldTopic, HelloWorld} the␣
↪following payload: <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}>
```

(continues on next page)

(continued from previous page)

```
...
Echo Participant: ParticipantId{EchoParticipant} has received from Endpoint: 01.0f.44.59.
↪e6.de.2a.c8.01.00.00.00|0.0.1.3 in topic: Topic{HelloWorldTopic, HelloWorld} the␣
↪following payload: <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}>
```

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:

```
Echo Participant: ParticipantId{EchoParticipant} has received from Endpoint: 01.0f.44.59.
↪33.e0.2b.cf.01.00.00.00|0.0.12.3 in topic: Topic{rt/chatter, std_msgs::msg::dds_
↪::String_} the following payload: <Payload{00 01 00 00 0f 00 00 00 48 65 6c 6c 6f 20␣
↪57 6f 72 6c 64 3a 20 34 00 00}>
Echo Participant: ParticipantId{EchoParticipant} has received from Endpoint: 01.0f.44.59.
↪33.e0.2b.cf.01.00.00.00|0.0.12.3 in topic: Topic{rt/chatter, std_msgs::msg::dds_
↪::String_} the following payload: <Payload{00 01 00 00 0f 00 00 00 48 65 6c 6c 6f 20␣
↪57 6f 72 6c 64 3a 20 35 00 00}>
...
```

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: v2.0                                                    # 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:
```

(continues on next page)

(continued from previous page)

```
###################################
# 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
    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 the 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 the 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 this *section* 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 *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.

```
###################################
# CONFIGURATION VERSION
version: v2.0                                        # 0

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

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

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

(continues on next page)

(continued from previous page)

```
###################################
# 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.

```yaml
  - 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: v2.0                                                   # 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
```

(continues on next page)

(continued from previous page)

```yaml
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 using Discovery Server protocol␣
↪as Server

  - name: WANServer                                            # 6
    kind: wan                                                  # 7
    discovery-server-guid:
      id: 1                                                    # 8
    listening-addresses:                                       # 9
      - ip: 1.1.1.1                                            # 10
        port: 11666                                            # 11
        transport: udp                                         # 12
```

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
  discovery-server-guid:
    id: 1                                            # 8
  listening-addresses:                               # 9
    - ip: 1.1.1.1                                    # 10
      port: 11666                                    # 11
      transport: udp                                 # 12
```

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
  discovery-server-guid:
    id: 2                                            # 8
  connection-addresses:                              # 9
    - discovery-server-guid:
        id: 1                                        # 10
      addresses:                                     # 11
        - ip: 1.1.1.1
          port: 11666
  listening-addresses:                               # 12
    - ip: 2.2.2.2                                    # 13
      port: 11670                                    # 14
      transport: udp                                 # 15
```

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
```

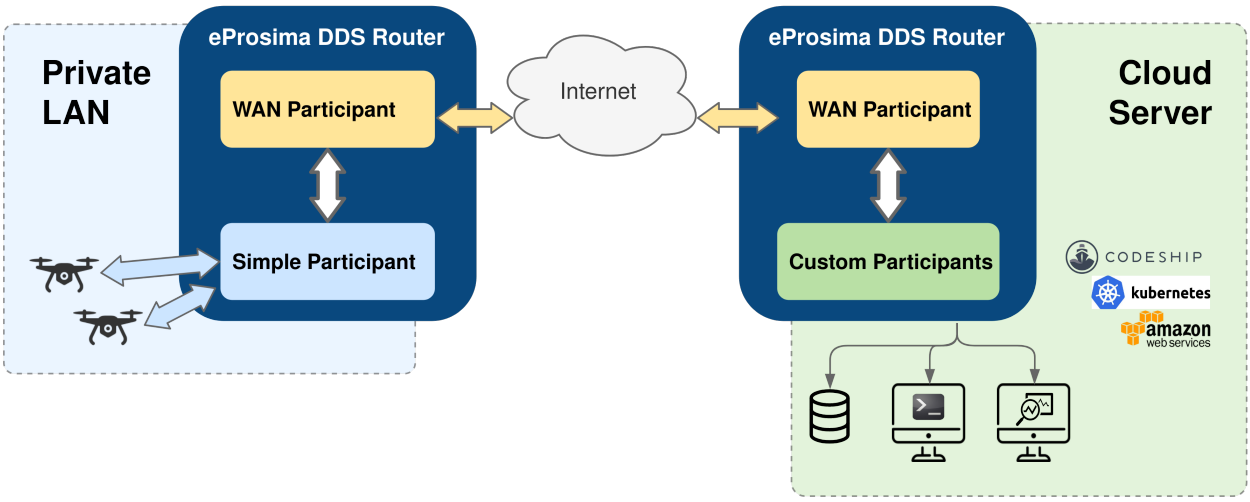
Result

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 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.

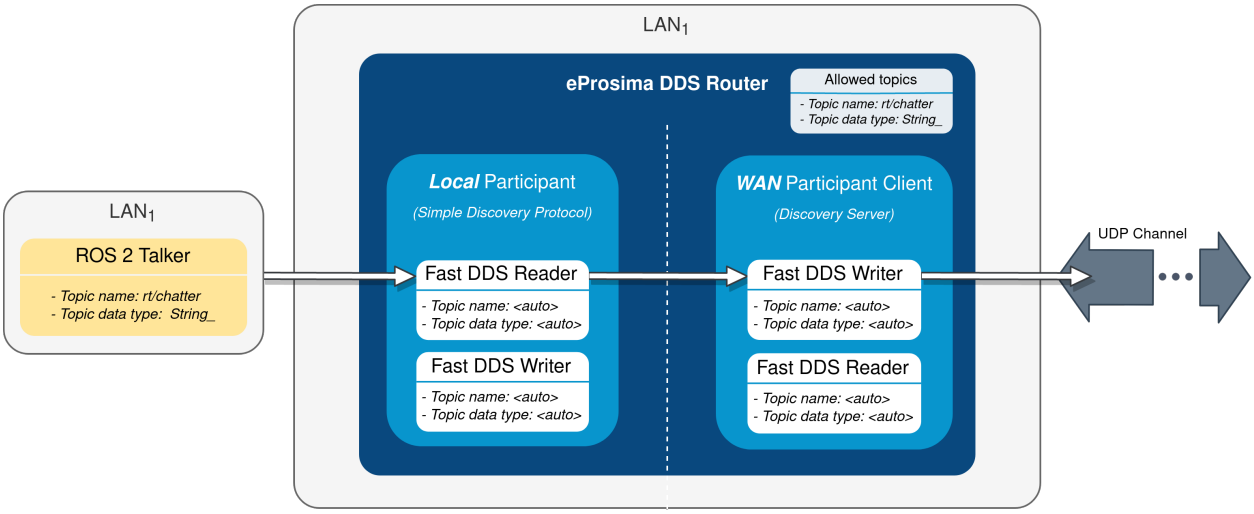


3.20.1 Local setup

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 *Discovery Server 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:

```yaml
# local-ddsrouter.yaml

version: v2.0

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

participants:

  - name: SimpleParticipant
    kind: local
    domain: 0

  - name: LocalWAN
    kind: wan
    discovery-server-guid:
      id: 3
    listening-addresses:      # Needed for UDP communication
      - ip: 3.3.3.3           # LAN public IP
        port: 30003
        transport: udp
    connection-addresses:
      - discovery-server-guid:
          id: 2
        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*.

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
```

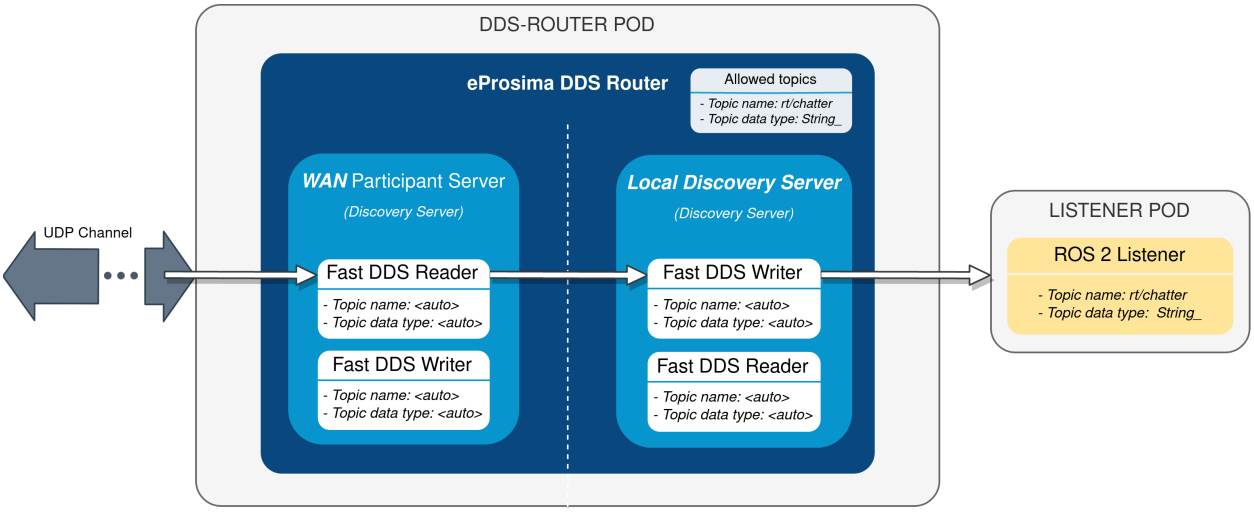
3.20.2 Kubernetes setup

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
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: ddsrouter
  labels:
    app: ddsrouter
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: v2.0

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

    participants:

      - name: LocalDiscoveryServer
```

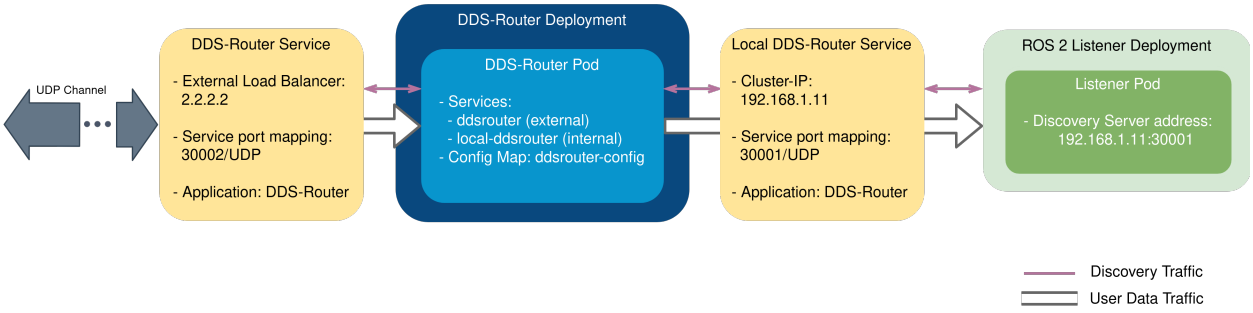
(continues on next page)

(continued from previous page)

```
        kind: local-discovery-server
        discovery-server-guid:
          ros-discovery-server: true
          id: 1
        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
        discovery-server-guid:
          id: 2
        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:
```

(continues on next page)

(continued from previous page)

```
      app: ddsrouter
  template:
    metadata:
      labels:
        app: ddsrouter
    spec:
      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
    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.

3.21 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:

3.21.1 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.

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*.

3.21.2 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.

3.21.3 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 ~/DDS-Router
```

2. Clone the following dependencies and compile them using CMake.

   - Foonathan memory

```
cd ~/DDS-Router
git clone https://github.com/eProsima/foonathan_memory_vendor.git
mkdir foonathan_memory_vendor/build
cd foonathan_memory_vendor/build
cmake .. -DCMAKE_INSTALL_PREFIX=~/DDS-Router/install -DBUILD_SHARED_
↪LIBS=ON
cmake --build . --target install
```

   - Fast CDR

```
cd ~/DDS-Router
git clone https://github.com/eProsima/Fast-CDR.git
mkdir Fast-CDR/build
cd Fast-CDR/build
```

(continues on next page)

(continued from previous page)

```
cmake .. -DCMAKE_INSTALL_PREFIX=~/DDS-Router/install
cmake --build . --target install
```

- Fast DDS

```
cd ~/DDS-Router
git clone https://github.com/eProsima/Fast-DDS.git
mkdir Fast-DDS/build
cd Fast-DDS/build
cmake .. -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*:

```
cd ~/DDS-Router
git clone https://github.com/eProsima/DDS-Router.git
mkdir DDS-Router/build
cd DDS-Router/build
cmake .. -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
```

3.21.4 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.

3.22 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:

3.22.1 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.

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*.

3.22.2 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.

3.22.3 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
```

2. Clone the following dependencies and compile them using CMake.

   - Foonathan memory

```
cd <path\to\user\workspace>\DDS-Router
git clone https://github.com/eProsima/foonathan_memory_vendor.git
cd foonathan_memory_vendor
mkdir build
cd build
cmake .. -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
git clone https://github.com/eProsima/Fast-CDR.git
cd Fast-CDR
mkdir build
cd build
cmake .. -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
git clone https://github.com/eProsima/Fast-DDS.git
cd Fast-DDS
```

(continues on next page)

(continued from previous page)

```
mkdir build
cd build
cmake .. -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*:

```
cd <path\to\user\workspace>\DDS-Router
git clone https://github.com/eProsima/DDS-Router.git
cd DDS-Router
mkdir build
cd build
cmake .. -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`

3.22.4 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

3.23 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

3.24 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 `ddsrouter_cmake` to implement general CMake functions and tools.
- The internal package `ddsrouter_event` is joint within the internal package `ddsrouter_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.

3.25 Previous Versions

3.25.1 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.
  - *ddsrouter_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.

3.25.2 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.

3.25.3 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.

3.26 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.

TCP **Transmission Control Protocol**

UDP **User Datagram Protocol**

WAN **Wide Area Network**

INDEX