

---

# **DDS Router Documentation**

***Release 0.1.0***

**eProxima**

**Dec 24, 2021**



# INTRODUCTION

<b>1</b>	<b>Contributing to the documentation</b>	<b>3</b>
<b>2</b>	<b>Structure of the documentation</b>	<b>5</b>
2.1	Contacts and Commercial support . . . . .	6
2.2	Contributing to the documentation . . . . .	6
2.3	Structure of the documentation . . . . .	6
2.4	DDS Router on Windows . . . . .	7
2.5	DDS Router on Linux . . . . .	7
2.6	Getting Started . . . . .	7
2.7	User Interface . . . . .	9
2.8	DDS Router Participant . . . . .	11
2.9	DDS Router Configuration . . . . .	18
2.10	WAN Configuration . . . . .	23
2.11	Nomenclature . . . . .	26
2.12	Echo Example . . . . .	27
2.13	Change Domain Example . . . . .	29
2.14	ROS2 Discovery Server Example . . . . .	31
2.15	WAN Example . . . . .	32
2.16	Vulcanexus Cloud . . . . .	36
2.17	Linux installation from sources . . . . .	43
2.18	Windows installation from sources . . . . .	47
2.19	CMake options . . . . .	52
2.20	Version v0.1.0 . . . . .	53
2.21	Glossary . . . . .	54
	<b>Index</b>	<b>57</b>



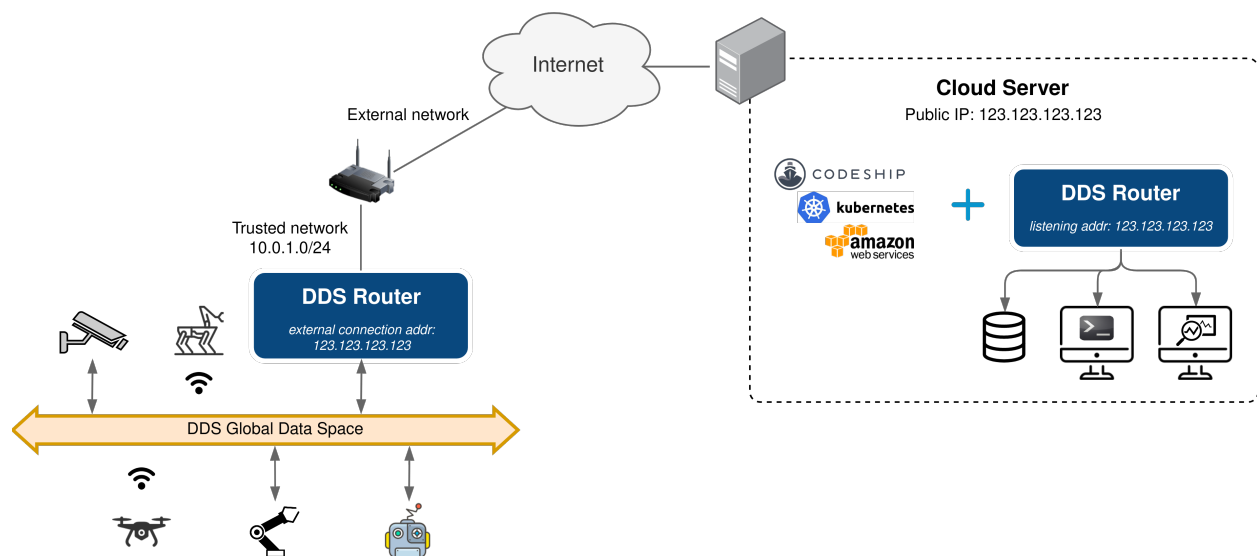


*eProsima DDS Router* is an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed 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.



Find more about us at [eProsima's webpage](https://www.eprosima.com).

Support available at:

- Email: [support@eprosima.com](mailto:support@eprosima.com)

- Phone: +34 91 804 34 48

## **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.





## STRUCTURE OF THE DOCUMENTATION

This documentation is organized into the sections below.

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

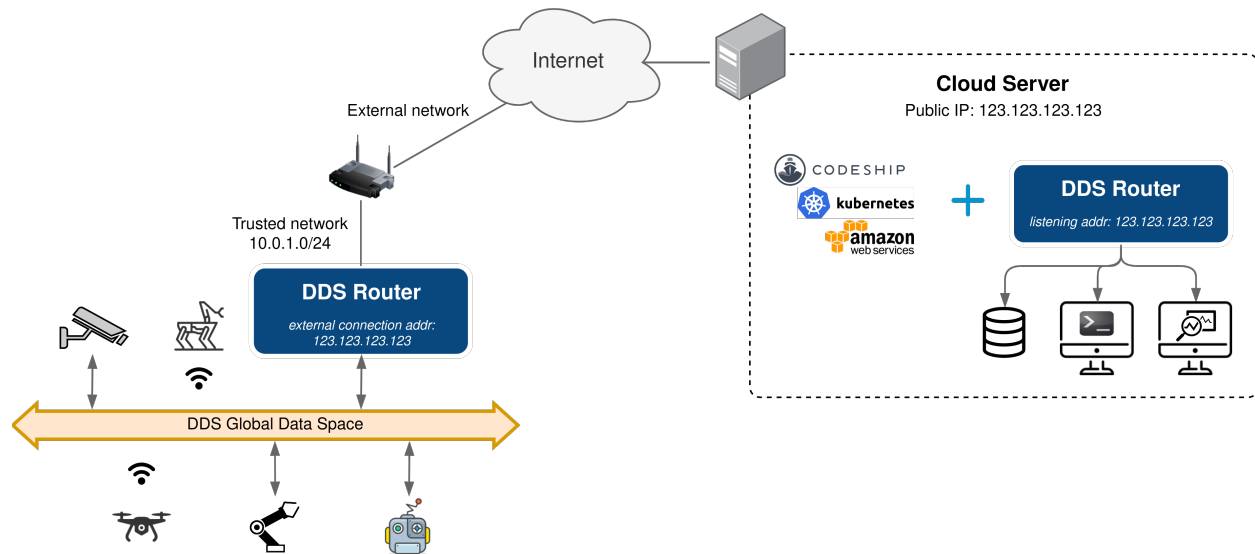


*eProsima DDS Router* is an end-user software application that enables the connection of distributed DDS networks. That is, DDS entities such as publishers and subscribers deployed in one geographic location and using a dedicated local network will be able to communicate with other DDS entities deployed 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.

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.



## 2.1 Contacts and Commercial support

Find more about us at [eProsimas webpage](#).

Support available at:

- Email: [support@eprosima.com](mailto:support@eprosima.com)
- Phone: +34 91 804 34 48

## 2.2 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.

## 2.3 Structure of the documentation

This documentation is organized into the sections below.

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

## 2.4 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.

## 2.5 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.

## 2.6 Getting Started

*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.

### 2.6.1 Project Overview

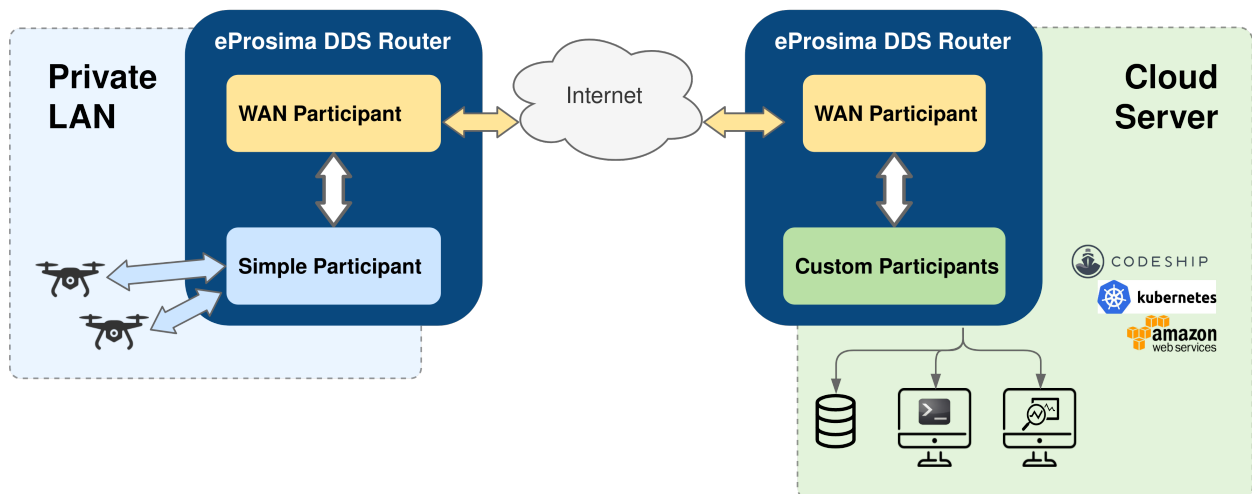
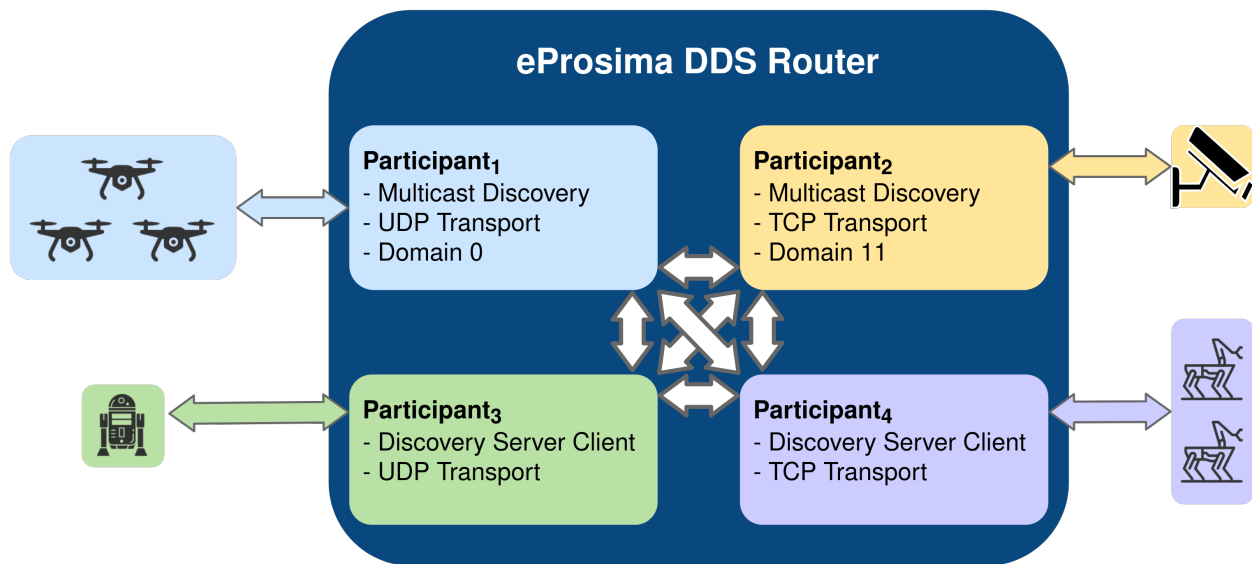
The *DDS Router* is an application that internally runs *Participants*, which are an abstraction of DDS *DomainParticipants*. Each one of these Participants is a communication interface, a “door” to a specific DDS network configuration. These Participants allow the application to connect to different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

### 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 work 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.



## 2.6.2 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](#) 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).

## 2.6.3 Common Use Cases

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

Use Case	Example
Communicating two different DDS Domain Ids	<a href="#">Change Domain Example</a>
Communicating ROS 2 Discovery Server executions	<a href="#">ROS2 Discovery Server Example</a>
WAN Communication	<a href="#">WAN Example</a>

## 2.7 User Interface

*eProsima DDS Router* is an application executed from command line.

- [Source Dependency Libraries](#)
- [Application Arguments](#)
- [Configuration File](#)
- [Reload Topics](#)
- [Close Application](#)

### 2.7.1 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.

## 2.7.2 Application Arguments

The *DDS Router* application supports several input arguments:

Command	Option	Long option	Value	Default Value
<i>Help Argument</i>	-h	--help		
<i>Configuration File Argument</i>	-c	--config-path	Readable File Path	./DDS_ROUTER_CONFIGURATION.yaml
<i>Reload Time Argument</i>	-r	--reload-time	Unsigned Integer	0
<i>Debug Argument</i>	-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.
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 FileWatcher 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).
```

### Configuration File Argument

Set the *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.

### 2.7.3 Configuration File

A *DDS Router* **requires** one and only one *YAML* configuration file. Check section *DDS Router Configuration* in order to know how to write 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.

### 2.7.4 Reload Topics

The topics that the *DDS Router* is routing could be changed at runtime. Including topics in configuration's `allowedlist` will create new *Writers* and *Readers* for each Participant in the Router. Removing a topic from `allowedlist` 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.

### 2.7.5 Close Application

In order to stop a *DDS Router* application, just send an interruption `SIGINT` | `^C` signal (*signal value 2*) to the process.

## 2.8 DDS Router Participant

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

- *Participant*
  - *Participant Id*
  - *Participant Type*

- *Participant creation*
- *Participant types*

## 2.8.1 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 Id* in a *DDS Router* execution and has a predefined *Participant Type* that specifies the internal general functionality of the Participant.

### Participant Id

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

### Participant Type

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

## 2.8.2 Participant creation

Each Participant has a unique Participant Id that can not be repeated in a *DDS Router* execution. This id is the name of the tag that contains the Participant configuration.

---

**Note:** If the id is repeated, the yaml will be bad formed and the *DDS Router* execution will fail when configured.

---

Each Participant Type is associated with one or several names or aliases that represent it. In order to use a Participant of a specific type, use `type` tag in the yaml configuration file, or set the Participant Id as the alias of the type. If the type 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 types could be repeated, but all ids must be unique.

---

Below are some examples on how to configure a Participant:

```
participant_1:      # New Participant with Id: 'participant_1'
  type: simple      # 'participant_1' will be created of type 'simple'
  extra_configuration: ...
```

```
simple:             # New Participant with Id: 'simple' and Type: 'simple'
  extra_configuration: ...
```



## 2.8.3 Participant types

Below is the list with all the available Participant Types.

Participant Type	Aliases	Specific configuration tags	Description
<i>Echo Participant</i>	echo		Print in <i>stdout</i> every data received.
<i>Simple Participant</i>	simple local	domain	Simple DDS DomainParticipant.
<i>Local Discovery Server Participant</i>	discovery-server local-ds ds	guid listening-addresses connection-addresses	Discovery Server DDS DomainParticipant for local communication.
<i>WAN Participant</i>	wan router	guid listening-addresses connection-addresses	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 Id*, 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.

### Type aliases

- echo

### Configuration

Echo Participant does not allow any configuration.

### Configuration Example

```
echo_participant:      # Participant Id = echo_participant
  type: echo
```

### Simple Participant

This type 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.

### Type 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

```
simple_participant:      # Participant Id = simple_participant
  type: simple
  domain: 2              # Domain Id = 2
```

### Local Discovery Server Participant

This type 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.

## Type 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.ca.fe Discovery Server GuidPrefix.

```
local_discovery_server_participant:           # Participant Id = local_discovery_
↪server_participant

    type: "discovery-server"

    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
    [
        {                                     # Use UDP by default
            ip: "127.0.0.1",
            port: 11600,
        },
    ],
```

(continues on next page)

(continued from previous page)

```

    {
        ip: "127.0.0.1",
        port: 11601,
        transport: "tcp"                # Use TCP transport
    }
]

connection-addresses:                  # External Discovery Server Listening
↪Addresses
[
    {
        id: 4                          # External Discovery Server id =>
↪GuidPrefix = 01.0f.04.00.00.00.00.00.00.ca.fe
        addresses:
        [
            {                            # Use UDP by default
                ip: "2001:4860:4860::8888",
                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.

## Type 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.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.ca.fe using UDP transport.

```
wan_participant:                # Participant Id = wan_participant

  type: "wan"

  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",
      port: 11600,
    }
  ]
                                # Use TCP by default

  connection-addresses:        # Another WAN Participant Listening_
↪Addresses
  [
    {
      id: 4                      # External Discovery Server id =>
↪GuidPrefix = 01.0f.04.00.00.00.00.00.00.ca.fe
      addresses:
      [
        {
          ip: "2001:4860:4860::8888",
          port: 11666,
```

(continues on next page)

(continued from previous page)

```

        transport: udp           # Use UDP transport
    }
  ]
}
]

```

## 2.9 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.

### 2.9.1 Topic Filtering

**Note:** The *DDS Router* topic discovery module is a work in progress. Thus, the functionality regarding Topic filtering is still in its early stages.

The *DDS Router* requires a list of allowed *Topics*. The data transmitted under these topics will be the data relayed by the *DDS Router*. The YAML configuration file must contain an `allowlist` tag which is a vector (`[]`). This vector elements contains the Topics that will be forwarded (subsequently the *DDS Router* creates a *Writers* and *Readers* for each topic for each Participant). See Topic section for further information about the topic.

In the following configuration example, the *DDS Router* will transmit the topic `rt/chatter` (default ROS 2 topic for talker and listener) with type name `std_msgs::msg::dds_::String_`. It also will transmit the topic `HelloWorldTopic` (default FastDDS topic for `HelloWorldExample`) with type name `HelloWorld`.

```

allowlist:
[
  {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"},
  {name: "HelloWorldTopic", type: "HelloWorld"},
]

```

**Note:** Tag `allowlist` must be at yaml base level (it must not be inside any other tag).

### 2.9.2 Participant Configuration

At the yaml base level, along with `allowlist` tag, there will be the Participants that will be created, together with their specific configuration. Each Participant is identified by a unique *Participant Id* that will be the yaml *key*. The yaml *value* for this *key* is the configuration for this specific Participant. There could be any number of Participants, and Participant types could be repeated.

Each Participant has its specific configuration. Please, refer to *Participant types* in order to see each of the *Participant Types* 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.

```
Participant0:      # Participant Id = Participant0
  type: local      # Participant Type = simple
  domain: 3        # DomainId = 3

#####

simple:            # Participant Id = simple ; Participant Type = simple
  domain: 6        # DomainId = 6
```

The first Participant *Participant0* has Participant Id *Participant0* and is configured to be of the *simple* Participant Type. The second Participant has Participant Id *simple* and it is configured to be of the *simple* type. That is because the type is not required to be specified as it is get from the Participant Id.

---

**Note:** Participant Type is not case sensitive. A Participant called *Simple* would be of type *simple*.

---



---

**Note:** The Participant Id is get as Participant Type when type is not specified. If type is explicitly specified, the Participant Id is not used to get the type.

---

### 2.9.3 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
```

### 2.9.4 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 Type.
- *IP version*: v4 or v6. If it is not set, it would be chosen depending on the *IP* string format.

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

#####

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

**Warning:** `ip` field does not currently support DNS names, only well-formed IP addresses.

## 2.9.5 Discovery Server GuidPrefix

A *Discovery Server* requires a DDS *GuidPrefix* in order to other Participants connect to it. 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 `..`

```
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>.00.00.00.00.00.00.00.00.ca.fe`. Default value for `id` is 0.

```
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](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.

```
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
```



## 2.9.6 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:
[
  {
    # UDP by default
    ip: "127.0.0.1",
    port: 11667,
  },
  {
    ip: "2001:4860:4860::8844",    # Recognized as IPv6
    port: 11668,
    transport: "tcp"
  }
]
```

## 2.9.7 Discovery Server Connection Addresses

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

```
connection-addresses:
[
  {
    guid: "44.53.0d.5f.45.50.52.4f.53.49.4d.41"
    addresses:
    [
      {
        ip: "127.0.0.1",
        port: 11666,
      }
    ]
  },
  {
    id: 4,
    addresses:
    [
      {
        ip: "2001:4860:4860::8888",
        port: 11667,
        transport: "tcp"
      },
      {
        ip: "2001:4860:4860::8844",
        port: 11668,
        transport: "tcp"
      }
    ]
  }
]
```

## 2.9.8 General Example

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

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

allowlist:
[
  {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"},
  {name: "HelloWorldTopic", type: "HelloWorld"},
]

#####

# Simple DDS Participant in domain 3

Participant0:                                # Participant Id = Participant0

    type: local                               # Participant Type = simple

    domain: 3                                # DomainId = 3

#####

# 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

simple:                                        # Participant Id = simple

    type: "local-discovery-server"           # Participant Type = local-discovery-server

    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.

Wan:                                         # Participant Id = simple ; Participant Type = wan
```

(continues on next page)

(continued from previous page)

```

    id: 2                                # Internal WAN Discovery Server id => GuidPrefix_
    ↪= 01.0f.02.00.00.00.00.00.00.ca.fe

    connection-addresses:                # WAN Discovery Server Connection Addresses
    [
        {
            id: 4,                        # External WAN Discovery Server id => GuidPrefix_
            ↪= 01.0f.04.00.00.00.00.00.00.ca.fe
            addresses:
            [
                {
                    ip: "8.8.8.8",          # IP = 8.8.8.8 ; transport = TCP (by_
                    ↪default)
                    port: 11666,           # Port = 11666
                }
            ]
        }
    ]

```

## 2.10 WAN Configuration

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

### 2.10.1 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.

### 2.10.2 TCP vs UDP

*TCP* and *UDP* are two well known network transport protocols. Both has its 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
<b>Commu- nication speed</b>	Fast	Slower
<b>Reliability</b>	No Transport Layer reliability (could has DDS reliability)	Transport Layer reliability (duplicated if DDS reliability is used)
<b>Port For- warding</b>	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.

## 2.10.3 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  $11666 \rightarrow 192.168.1.2:11666$ . 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:

```
WANServerParticipant_userA:
  type: "wan"

  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 *connection-addresses* to connect to  $H_A$  as follows:

```
WANClientParticipant_userB:
  type: "wan"

  id: 3                                # Must be different than A one
  connection-addresses:
  [
    {
      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  $A$ .  $A$  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  $11666 \rightarrow 192.168.1.2:11666$ . 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:

```
WANServerParticipant_userA:
  type: "wan"

  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  $11777 \rightarrow 192.168.2.2:11777$ . 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:

```
WANClientParticipant_userB:
  type: "wan"

  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:
  [
    {
      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  $1.1.1.1:11666$ .

## 2.11 Nomenclature

### 2.11.1 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 Id** Unique identifier of a Participant.

This term is explained [here](#).

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

This term is explained [here](#).

### 2.11.2 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.

## 2.12 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.

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

allowlist:
  [
    {name: "HelloWorldTopic", type: "HelloWorld"},           # 1
    {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}, # 2
  ]

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

SimpleParticipant:                                           # 3
  type: local                                              # 4
  domain: 0                                                # 5

#####
# ECHO PARTICIPANT
# This Participant will print in stdout every message received by the other
↳ Participants

EchoParticipant:                                           # 6
  type: echo                                              # 7
```

### 2.12.1 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", type: "HelloWorld"},           # 1
    {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}, # 2
  ]
```

## Simple Participant

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

<b>SimpleParticipant:</b>	# 3
<b>type:</b> local	# 4
<b>domain:</b> 0	# 5

## Echo Participant

This Participant does not require further configuration than name and type.

<b>EchoParticipant:</b>	# 6
<b>type:</b> echo	# 7

## 2.12.2 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:

<code>./DDSHelloWorldExample publisher</code>
---

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

<pre>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: &lt;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}&gt; 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: &lt;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}&gt; ... 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: &lt;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}&gt;</pre>
--



## 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>/src/ddsrouter/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}>
...
```

## 2.13 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.

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

allowlist:
[
  {name: "HelloWorldTopic", type: "HelloWorld"},           # 1
  {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}, # 2
]

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

SimpleParticipant_domain0:                                   # 3
  type: local                                                # 4
  domain: 0                                                  # 5

#####
# SIMPLE PARTICIPANT DOMAIN 0
# This Participant will print in stdout every message received by the other
↪Participants

SimpleParticipant_domain1:                                   # 6
  type: local                                                # 7
  domain: 1                                                  # 8
```

## 2.13.1 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", type: "HelloWorld"},           # 1
  {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}, # 2
]
```

### Simple Participant Domain 0

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

```
SimpleParticipant_domain0:                                # 3
  type: local                                             # 4
  domain: 0                                              # 5
```

### Simple Participant Domain 1

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

```
SimpleParticipant_domain1:                                # 6
  type: local                                             # 7
  domain: 1                                              # 8
```

## 2.13.2 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>/resources/configurations/examples/change\_domain.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.

## 2.14 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.

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

allowlist:
[
  {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}, # 1
]

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

SimpleROS2:                                # 2
  type: local                               # 3
  domain: 0                                # 4

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

ServerROS2:                                # 5
  type: local-discovery-server              # 6
  ros-discovery-server: true                # 7
  id: 1                                     # 8
  listening-addresses:                      # 9
  [
    {
      ip: "127.0.0.1",                      # 10
      port: 11888                           # 11
    },                                       # 12
  ]
]
```

### 2.14.1 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", type: "std_msgs::msg::dds_::String_"}, # 1
]
```

## Simple Participant

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

```
SimpleROS2: # 2
  type: local # 3
  domain: 0 # 4
```

## Discovery Server Participant

This Participant is configured by a name, a type and a listening addresses where Discovery Server will expect meta-traffic data from clients.

```
ServerROS2: # 5
  type: local-discovery-server # 6
  ros-discovery-server: true # 7
  id: 1 # 8
  listening-addresses: # 9
  [
    {
      ip: "127.0.0.1", # 10
      port: 11888 # 11
    }, # 12
  ]
```

### 2.14.2 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>/src/ddsrouter/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`.

## 2.15 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*.

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

allowlist:
  [
```

(continues on next page)

(continued from previous page)

```

    {name: "HelloWorldTopic", type: "HelloWorld"},           # 1
    {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}, # 2
  ]

#####
# SIMPLE PARTICIPANT
# This participant will subscribe to topics in allowlist in domain 0 and listen every_
↳message published there

SimpleParticipant:                                           # 3
  type: local                                                # 4
  domain: 0                                                  # 5

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

WANServer:                                                  # 6
  type: wan                                                  # 7
  id: 1                                                      # 8
  listening-addresses:                                       # 9
  [
    {
      ip: "1.1.1.1",                                         # 10
      port: 11666,                                           # 11
      transport: "udp"                                       # 12
    },                                                       # 13
  ]
]

```

## 2.15.1 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: "HelloWorldTopic", type: "HelloWorld"},           # 1
  {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}, # 2
]

```

## Simple Participant

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

```
SimpleParticipant:                                # 3
  type: local                                     # 4
  domain: 0                                       # 5
```

## WAN Participant Server

This Participant is configured with a name, a type 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).

```
WANServer:                                        # 6
  type: wan                                       # 7
  id: 1                                           # 8
  listening-addresses:                           # 9
  [
    {
      ip: "1.1.1.1",                             # 10
      port: 11666,                                # 11
      transport: "udp"                             # 12
    },                                           # 13
  ]
```

## WAN Participant Client

In order to create a WAN Participant Client, check the configuration file <path/to/ddsrouter>/src/ddsrouter/resources/configurations/examples/wan\_client.yaml

```
WANClient:                                        # 6
  type: wan                                       # 7
  id: 2                                           # 8
  connection-addresses:                          # 9
  [
    {
      id: 1,                                       # 10
      addresses:
      [
        {ip: "1.1.1.1", port: 11666, transport: "udp"}, # 11
      ]
    }
  ]
```

## 2.15.2 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>/src/ddsrouter/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 Server, which will wait for other WAN Clients to connect to it. Execute *DDS Router* using file <path/to/ddsrouter>/src/ddsrouter/resources/configurations/examples/wan\_client.yaml. Remember to change the IP and port on the configuration file to the actual public IP of  $R_A$ , and be sure that port forwarding is 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*.

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.**

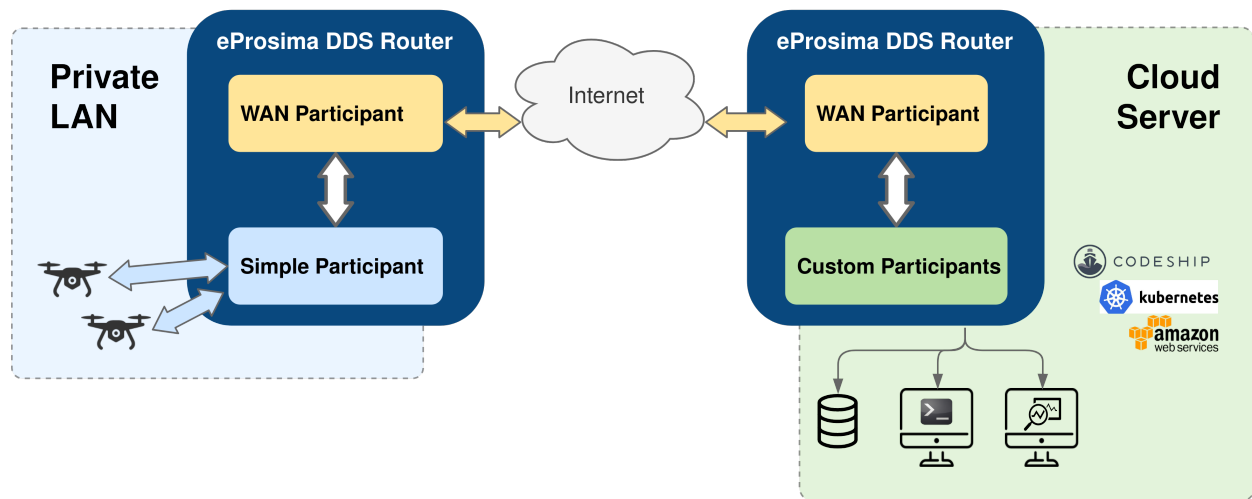
Remember that the Participants in every *DDS Router* could be configured as any [Participant Type](#), allowing to use local Discovery Server, connect to several domains in the same LAN, connect to several WANs, etc. Endless Possibilities. Just remember uncle Ben's words: *with great power comes great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

**Warning:** *Vulcanexus* distribution is not currently available, but it will very soon. To follow this tutorial, a ROS 2 galactic Docker image may be used instead, [installing DDS Router](#) whenever required. Please also take into account that environment variable RMW\_IMPLEMENTATION must be exported so as to utilize *Fast DDS* as middleware in ROS (see [Working with eProsima Fast DDS](#)).

## 2.16 Vulcanexus Cloud

*Vulcanexus* is an extended *ROS 2* distribution provided by *eProsima* which includes additional tools for an improved user and development experience, such as *Fast-DDS-Monitor* for monitoring the health of DDS networks, or *micro-ROS*, a framework aimed at optimizing and extending *ROS 2* toolkit to microcontroller-based devices.

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 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.



### 2.16.1 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:

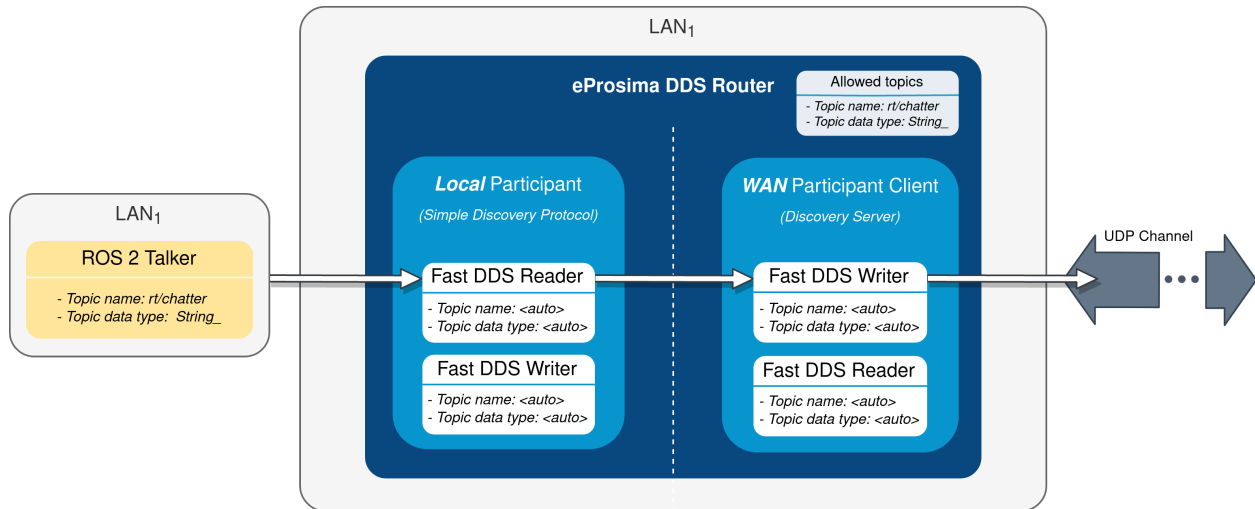
```
# local-ddsrouter.yaml

allowlist:
[
  {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}
]

SimpleParticipant:
```

(continues on next page)





(continued from previous page)

```

type: local
domain: 0

LocalWAN:
  type: wan
  id: 3
  listening-addresses: # Needed for UDP communication
  [
    {
      ip: "3.3.3.3", # LAN public IP
      port: 30003,
      transport: "udp"
    }
  ]
  connection-addresses:
  [
    {
      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](#).

To launch the local router from within a Vulcanexus Docker image, execute the following:

```
docker run -it --net=host -v local-ddsrouter.yaml:/tmp/local-ddsrouter.yaml ubuntu-
↳vulcanexus:galactic -r "ddsrouter -c /tmp/local-ddsrouter.yaml"
```

## Talker

ROS 2 demo nodes is not installed in *Vulcanexus* distribution by default, but one can easily create a new Docker image including this feature by using the following Dockerfile:

```
FROM ubuntu-vulcanexus:galactic

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

# Setup entrypoint
ENTRYPOINT ["/bin/bash", "/vulcanexus_entrypoint.sh"]
CMD ["bash"]
```

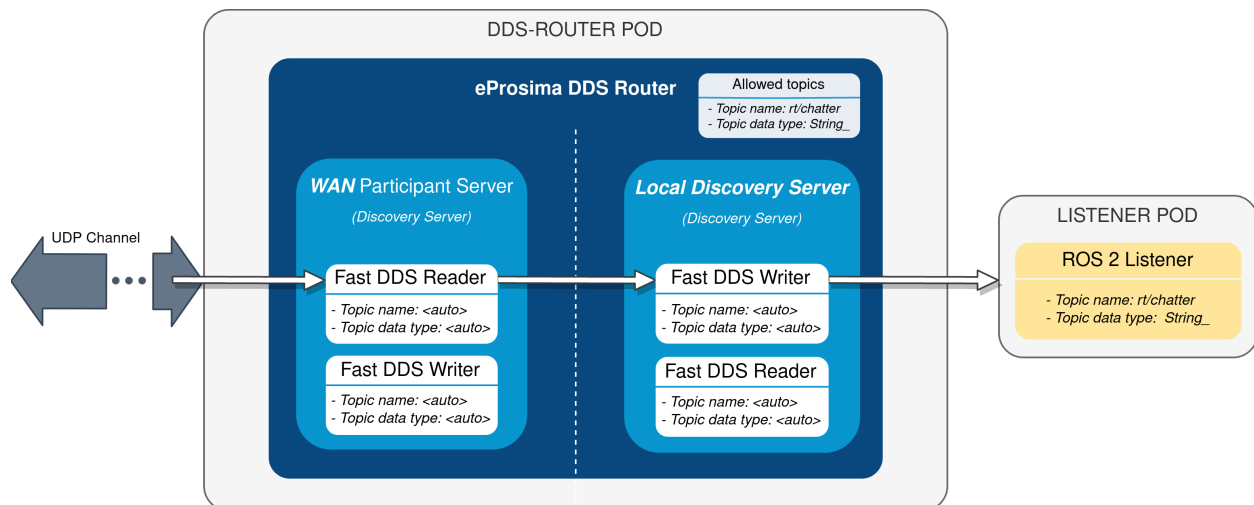
Create the new image and start publishing messages by executing:

```
docker build -t vulcanexus-demo-nodes:galactic -f Dockerfile .
docker run -it --net=host vulcanexus-demo-nodes:galactic -r "ros2 run demo_nodes_cpp_
↳talker"
```

## 2.16.2 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, and a *Local Discovery Server* (local DS) that propagates them to a ROS 2 listener node hosted in a different *K8s* pod. The choice of a Local Discovery Server instead of a Simple Participant to communicate with the listener has to do with the difficulty of enabling multicast routing in cloud environments.

The described scheme is represented in the following figure:



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

```
kind: Service
apiVersion: v1
metadata:
  name: ddsrouter
  labels:
    app: ddsrouter
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: |-
    allowlist:
      [
        {name: "rt/chatter", type: "std_msgs::msg::dds_::String_"}
      ]

    LocalDiscoveryServer:
      type: local-discovery-server
```

(continues on next page)

(continued from previous page)

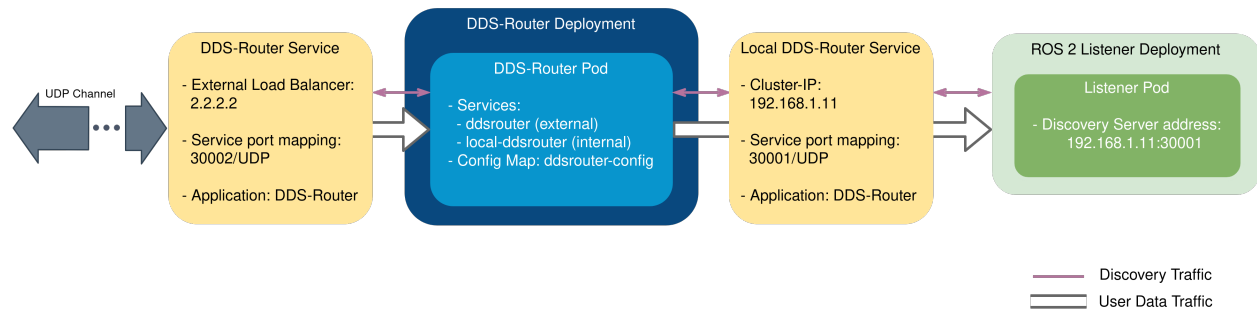
```

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"
  }
]

CloudWAN:
type: wan
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 represented the overall configuration of our *K8s* cluster:



## DDS-Router deployment

The cloud router is launched from within a Vulcanexus Docker image, which uses as configuration file the one hosted in the previously set up ConfigMap. The cloud router will be deployed with the following settings:

```

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

```

(continues on next page)

(continued from previous page)

```

template:
  metadata:
    labels:
      app: ddsrouter
  spec:
    volumes:
      - name: config
        configMap:
          name: ddsrouter-config
          items:
            - key: ddsrouter.config.file
              path: DDSROUTER_CONFIGURATION.yaml
    containers:
      - name: ubuntu-vulcanexus
        image: ubuntu-vulcanexus:galactic
        ports:
          - containerPort: 30001
            protocol: UDP
          - containerPort: 30002
            protocol: UDP
        volumeMounts:
          - name: config
            mountPath: /vulcanexus_ws/install/ddsrouter/share/resources
            args: ["-r", "ddsrouter -r 10 -c /vulcanexus_ws/install/ddsrouter/share/
↪resources/DDSROUTER_CONFIGURATION.yaml"]
        restartPolicy: Always

```

## Listener deployment

Again, since demo nodes is not installed by default in *Vulcanexus* we have to create a new Docker image adding in this functionality. The Dockerfile used for the listener will slightly differ from the one utilized to launch a talker in our LAN, as here we need to specify the port and IP address of the local DS. This can be achieved by using the following Dockerfile and entrypoint:

```

FROM ubuntu-vulcanexus:galactic

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

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

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

```

```

#!/bin/bash

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

```

(continues on next page)

(continued from previous page)

```
    NODE="talker"
fi

SERVER_IP=$2
SERVER_PORT=$3

# Setup environment
source "/opt/ros/${ROS_DISTRO}/setup.bash"
source "/vulcanexus_ws/install/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}
```

As before, to build the extended Docker image it suffices to run:

```
docker build -t vulcanexus-demo-nodes:galactic -f Dockerfile .
```

Now, 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: vulcanexus-demo-nodes
          image: vulcanexus-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. Feel free to interchange the locations of the ROS nodes by slightly modifying the provided configuration files, so that the talker is the one hosted in the *K8s* cluster while the listener runs in our LAN.

## 2.17 Linux installation from sources

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

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

### 2.17.1 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]

### 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](#).

### Dependencies

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

- *Asio and TinyXML2 libraries*
- *OpenSSL*
- *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
```

## 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](#).

### 2.17.2 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.

---

## 2.17.3 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
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
```

### 2.17.4 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/bin/ddsrouter`:

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

Be sure that this executable has execute permissions.

## 2.18 Windows installation from sources

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

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

### 2.18.1 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]

### 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.

## Dependencies

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

- *Asio and TinyXML2 libraries*
- *OpenSSL*
- *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
```

### 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*.

### 2.18.2 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.

---

### 2.18.3 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
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`

### 2.18.4 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`

## 2.19 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_TESTS	Build the <i>DDS Router</i> application and documentation tests. Setting BUILD_TESTS to ON sets BUILD_APP_TESTS and BUILD_DOCUMENTATION_TESTS to ON.	OFF ON	OFF
BUILD_APP_TESTS	Build the <i>DDS Router</i> application tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
BUILD_DOCUMENTATION_TESTS	Build the <i>DDS Router</i> documentation tests. It is set to ON if BUILD_TESTS is set to ON.	OFF ON	OFF
BUILD_DOCUMENTATION	Build the <i>DDS Router</i> documentation. It is set to ON if BUILD_TESTS_DOCUMENTATION is set to ON.	OFF ON	OFF
LOG_INFO	Active <i>DDS Router</i> execution logs. It is set to ON if BUILD_TESTS_DOCUMENTATION is set to ON.	OFF ON	ON if Debug OFF otherwise

## 2.20 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.

## 2.21 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

### D

DataReader, [26](#)  
DataWriter, [26](#)  
Discovery Server, [26](#)  
Domain Id, [26](#)  
DomainParticipant, [26](#)

### E

Endpoint, [26](#)

### G

Guid, [26](#)  
GuidPrefix, [26](#)

### L

LAN, [54](#)

### N

NAT, [54](#)

### P

Participant, [26](#)  
Participant Id, [26](#)  
Participant Type, [26](#)  
Payload, [26](#)

### T

TCP, [54](#)  
Topic, [27](#)

### U

UDP, [54](#)

### W

WAN, [55](#)