Discovery Server Documentation

Release 2.0.0

eProsima

Nov 11, 2022

INSTALLATION MANUAL

The RTPS standard specifies in section 8.5 a non-centralized, distributed simple discovery mechanism. This mechanism was devised to allow interoperability among independent vendor-specific implementations but is not expected to be optimal in every environment. There are several scenarios were the simple discovery mechanism is unsuitable or plainly cannot be applied: a) a high number of endpoint entities are continuously entering and leaving the communication, b) wide communication systems deployment, and c) networks without multicasting capabilities.

In order to cope with the above issues, the *eProsima Fast DDS* discovery mechanism was extended with a new Discovery Server discovery mechanism. This mechanism is based on a client-server discovery paradigm, i.e. the metatraffic (message exchange among DDS DomainParticipants to identify each other) is managed by one or several server DomainParticipants (left figure), as opposed to simple discovery (right figure), where metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol. Please, refer to Fast DDS documentation for further information about the Discovery Server discovery mechanism.

Warning: This documentation refers to the GitHub Discovery Server repository, which implements an application to test the Discovery Server discovery mechanism. Therefore it should be emphasized that **Discovery Server is a discovery mechanism already available in Fast DDS** and this documentation refers to an application mainly used to test this functionality.

This documentation is organized into the following sections:

- *Installation Manual*
- *User Manual*
- *XML examples*
- *C++ Examples*
- *Release Notes*

CHAPTER

ONE

LINUX INSTALLATION

The instructions for installing the Discovery Server tool in a Linux environment are provided in this page. In order to use the Discovery Server tool, its necessary to have a compatible version of eProsima Fast DDS installed (over release 2.0.2).

eProsima Fast DDS dependencies as tinyxml must be accessible, either because *Fast DDS* was build-installed defining THIRDPARTY=ON or because those libraries have been specifically installed. The cross-platform tool colcon was chosen to simplify the installation of the several mutually dependent CMake projects. In order to use colcon, Python3 and CMake must be first installed.

1.1 Requirements

The installation of the Discovery Server tool in a Linux environment from sources requires the following tools to be installed in the system:

- *CMake, g++, pip3, wget and git*
- *Python3 modules* [optional]

1.1.1 CMake, g++, pip3, wget and git

These packages provide the tools required to install the Discovery Server tool and its dependencies from command line. Install CMake, g++, pip3, wget and git using the package manager of the appropriate Linux distribution. For example, on Ubuntu use the command:

```
sudo apt install cmake g++ python3-pip wget git
```

1.1.2 Python3 modules

To execute the tests that verify the proper operation of the Discovery Server discovery mechanism, it is necessary to install some Python3 modules. These can be installed using *pip*.

```
pip3 install jsondiff==1.2.0 xmltodict==0.12.0
```

1.2 Dependencies

The Discovery Server tool and *eProsima Fast DDS* has the following dependencies, when installed from binaries in a Linux environment:

- *Asio and TinyXML2 libraries*
- *OpenSSL*

1.2.1 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
```

1.2.2 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
```

1.3 Installation steps

colcon is a command line tool based on CMake aimed at building sets of software packages. This section explains how to use it to compile the Discovery Server tool and its dependencies.

1. 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.

2. Create a Discovery Server workspace and download the repos file that will be used to install the Discovery Server tool and its dependencies:

```
$ mkdir -p discovery-server-ws/src && cd discovery-server-ws
$ wget https://raw.githubusercontent.com/eProsima/Discovery-Server/master/discovery-
↪server.repos
$ vcs import src < discovery-server.repos
```

The discovery-server.repos file is provided in order to profit from vcstool capabilities to download the needed repositories.

Note: In order to avoid using vcstool the following repositories should be downloaded from Github into the `discovery-server-ws/src` directory:

PACKAGE	URL	BRANCH
eProsima/Fast-CDR	https://github.com/eProsima/Fast-CDR.git	master
eProsima/Fast-RTPS	https://github.com/eProsima/Fast-RTPS.git	master
eProsima/Discovery-Server	https://github.com/eProsima/Discovery-Server.git	master
eProsima/foonathan_memory_vendor	https://github.com/eProsima/foonathan_memory_vendor.git	master

3. Finally, use colcon to compile all software. Choose the build configuration by declaring `CMAKE_BUILD_TYPE` as Debug or Release. For this example, the Debug option has been chosen, which would be the choice of advanced users for debugging purposes.

```
$ colcon build --base-paths src \
        --packages-up-to discovery-server \
        --cmake-args -DLOG_LEVEL_INFO=ON -DCOMPILE_EXAMPLES=ON -DINTERNALDEBUG=ON -
↪DCMAKE_BUILD_TYPE=Debug
```

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.

1.4 Run an application

1. If you installed the Discovery Server tool following the steps outlined above, you can try the `HelloWorldExampleDS`. To run the example navigate to the following directory

   `<path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/examples/HelloWorldExampleDS`

   and run

   ```
   $ ./HelloWorldExampleDS --help
   ```

   to display the example usage instructions.

   In order to test the `HelloWorldExampleDS` open three terminals and run the above command. Then run the following command in each terminal:

   - Terminal 1:

     ```
     $ cd <path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/
     ↪examples/HelloWorldExampleDS
     $ ./HelloWorldExampleDS publisher
     ```

   - Terminal 2:

     ```
     $ cd <path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/
     ↪examples/HelloWorldExampleDS
     $ ./HelloWorldExampleDS subscriber
     ```

   - Terminal 3:

     ```
     $ cd <path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/
     ↪examples/HelloWorldExampleDS
     $ ./HelloWorldExampleDS server
     ```

CHAPTER

TWO

WINDOWS INSTALLATION

The instructions for installing the Discovery Server tool in a Windows environment are provided in this page. In order to use the Discovery Server tool, its necessary to have a compatible version of eProsima Fast DDS installed (over release 2.0.2).

eProsima Fast DDS dependencies as tinyxml must installed and accessible in the system. The cross-platform tool colcon was chosen to simplify the installation of the several mutually dependent CMake projects. In order to use colcon, Python3 and CMake must be first installed.

2.1 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*
- *Python3 modules* [optional]

2.1.1 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.

2.1.2 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.

2.1.3 CMake, pip3, wget and git

These packages provide the tools required to install the Discovery Server tool, *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.

2.1.4 Python3 modules

To execute the tests that verify the proper operation of the Discovery Server discovery mechanism, it is necessary to install some Python3 modules. These can be installed using *pip*.

```
> pip3 install jsondiff==1.2.0 xmltodict==0.12.0
```

2.2 Dependencies

eProsima Fast RTPS has the following dependencies, when installed from sources in a Windows environment:

- *Asio and TinyXML2 libraries*
- *OpenSSL*

2.2.1 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.

2.2.2 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
```

2.3 Installation steps

colcon is a command line tool based on CMake aimed at building sets of software packages. This section explains how to use it to compile the Discovery Server tool and its dependencies.

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

---

1. Install the ROS 2 development tools (colcon and vcstool) by executing the following command:

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

   and add the path to the `vcs` executable to the `PATH` from the *Edit the system environment variables* control panel.

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

   ---

2. Create a Discovery Server workspace and download the repos file that will be used to install the Discovery Server tool and its dependencies:

   ```
   > mkdir discovery-server-ws
   > cd discovery-server-ws
   > mkdir src
   > wget https://raw.githubusercontent.com/eProsima/Discovery-Server/master/discovery-
   ↪server.repos
   > vcs import src < discovery-server.repos
   ```

   A discovery-server.repos file is available in order to profit from vcstool capabilities to download the needed repositories.

   Note: In order to avoid using vcstool the following repositories should be downloaded from Github into the `discovery-server-ws/src` directory:

PACKAGE	URL	BRANCH
eProsima/Fast-CDR	https://github.com/eProsima/Fast-CDR.git	master
eProsima/Fast-RTPS	https://github.com/eProsima/Fast-RTPS.git	master
eProsima/Discovery-Server	https://github.com/eProsima/Discovery-Server.git	master
eProsima/foonathan_memory_vendor	https://github.com/eProsima/foonathan_memory_vendor.git	master

   ---

3. If the generator (compiler) of choice is Visual Studio, launch colcon from a visual studio console. Any console can be setup into a visual studio one by executing a batch file. For example, in VS2017 is usually `C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\Common7\Tools\VsDevCmd.bat`.

4. Finally, use colcon to compile all software. Choose the build configuration by declaring `CMAKE_BUILD_TYPE` as Debug or Release. For this example, the Debug option has been chosen, which would be the choice of advanced users for debugging purposes. If using a multi-configuration generator like Visual Studio we recommend to build both in debug and release modes

```
> colcon build --base-paths src \
        --packages-up-to discovery-server \
        --cmake-args -DLOG_LEVEL_INFO=ON -DCOMPILE_EXAMPLES=ON \
                -DINTERNALDEBUG=ON -DCMAKE_BUILD_TYPE=Debug
> colcon build --base-paths src \
        --packages-up-to discovery-server \
        --cmake-args -DCOMPILE_EXAMPLES=ON -DCMAKE_BUILD_TYPE=Release
```

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.

2.4 Run an application

1. If you installed the Discovery Server tool following the steps outlined above, you can try the `HelloWorldExampleDS`. To run the example navigate to the following directory

   `<path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/examples/HelloWorldExampleDS`

   and run

```
> HelloWorldExampleDS --help
```

   to display the example usage instructions.

   In order to test the `HelloWorldExampleDS` open three consoles and run the above command. Then run the following command in each console:

   - Console 1:

```
> cd <path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/
↪examples/HelloWorldExampleDS
> HelloWorldExampleDS publisher
```

   - Console 2:

```
> cd <path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/
↪examples/HelloWorldExampleDS
> HelloWorldExampleDS subscriber
```

   - Console 3:

```
> cd <path/to/discovery-server-ws>/discovery-server-ws/install/discovery-server/
↪examples/HelloWorldExampleDS
> HelloWorldExampleDS server
```

CHAPTER

THREE

CMAKE OPTIONS

eProsima Discovery Server provides some CMake options for changing the behavior and configuration of *Discovery Server* application. These options allow the user to enable/disable certain settings by defining these options to ON/OFF at the CMake execution.

Option	Description	Possible values	Default
`COMPILE_EXAMPLES`	Build Discovery Server example.	`Release` `Debug`	`Release`
`LOG_LEVEL_INFO`	Set logging level to Info.	`ON` `OFF`	`OFF`
`LOG_LEVEL_WARN`	Set logging level to Warning.	`ON` `OFF`	`OFF`
`LOG_LEVEL_ERROR`	Set logging level to Error.	`ON` `OFF`	`OFF`
`SANITIZER`	Adds run-time instrumentation to the code. Supported options are: • `Thread` enables Thread Sanitizer. • `Address` enables Address Sanitizer.	`OFF` `Address` `Thread`	`OFF`

CHAPTER

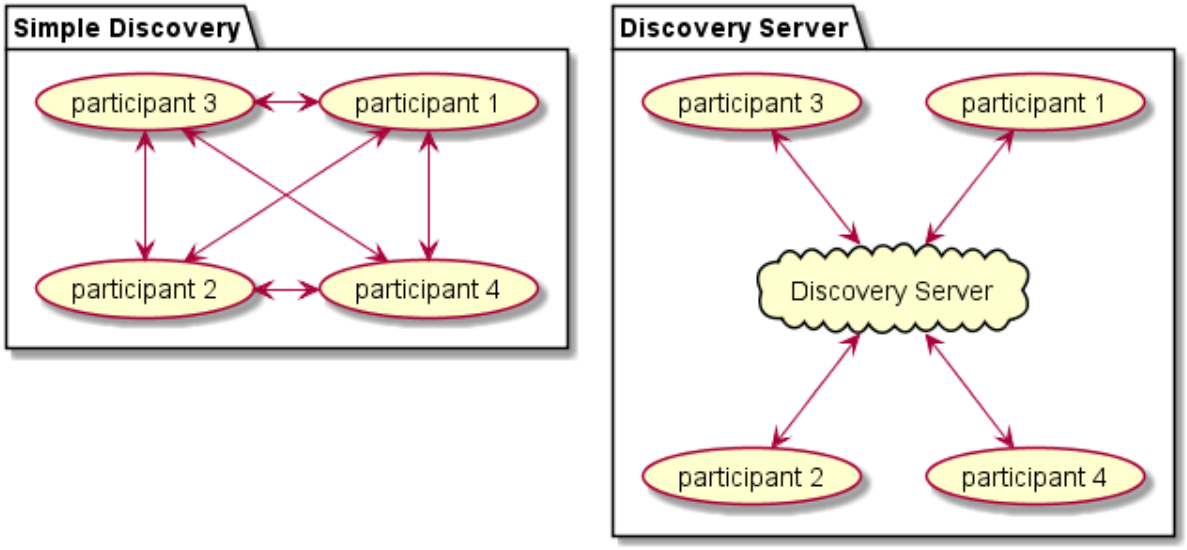
FOUR

GETTING STARTED

This section explains the basic concepts of the Discovery Server discovery mechanism. For more information on the Discovery Server mechanism, please refer to the Fast DDS documentation.

4.1 Basic concepts

Under the new client-server discovery paradigm, the metatraffic (message exchange among participants to identify each other) is centralized in one or several server participants (right figure), as opposed to simple discovery (left figure), where metatraffic is exchanged using a message broadcast mechanism like an IP multicast protocol.



Clients must be aware of how to reach the server, usually by specifying an IP address and a transport protocol like UDP or TCP. Servers do not need any beforehand knowledge of their clients but, we must specify where they may be reached by them, usually by specifying a listening IP address and transport protocol.

One of the design goals of the current implementation was to keep both the discovery messages structure and standard RTPS writer and reader behavior unchanged. In order to do so, clients must be aware of their server's GuidPrefix. GuidPrefix is the RTPS standard participant unique identifier (basically 12 bytes) which allows clients to assess whether they are receiving messages from the right server, as each standard RTPS message contains this piece of information. Note that the server's IP address may not be a reliable server's identifier because several can be specified and multicast addresses are acceptable. In future implementations, any other more convenient and non-standard identifier may substitute the GuidPrefix at the expense of adding non-standard members to the RTPS discovery messages structure.

Finally, the discovery between clients is only performed in case of a match at the topic level between publishers and subscribers. That is, customers with publishers and subscribers on different topics will never discover each other. This implies a notorious reduction in the number of messages exchanged.

4.1.1 RTPS attributes dealing with discovery services

Several *Fast DDS* configuration structures have been updated in order to deal with the new client-server discovery strategy. Note that the following elements belong exclusively to fast RTPS builtin discovery architecture and that the discovery server application just profits from the capabilities provided by *Fast DDS* library.

RTPSParticipantAttributes

- `GuidPrefix_t guidPrefix` member specifies the server's identity. This member has only significance if *discovery_config.discoveryProtocol* is **SERVER** or **BACKUP**. There is a *ReadguidPrefix* method to easily fill in this member from a string formatted like `"4D.49.47.55.45.4c.5f.42.41.52.52.4f"` (note that each byte must be a valid hexadecimal figure).

BuiltinAttributes

- All discovery related info is gathered in a `DiscoverySettings discovery_config` member.
- In order to receive client metatraffic, *metatrafficUnicastLocatorList* or *metatrafficMulticastLocatorList* must be populated with the addresses that were given to the clients.

DiscoverySettings

- `DiscoveryProtocol_t discoveryProtocol` member specifies the participant's discovery kind:
  - **SIMPLE** generates a standard participant with complete backward compatibility with any other RTPS implementation.
  - **CLIENT** generates a *client* participant, which relies on a server to be notified of other *clients* presence. This participant can create publishers and subscribers of any topic (static or dynamic) as ordinary participants do.
  - **SERVER** generates a *server* participant, which receives, manages and spreads its linked *clients* metatraffic assuring any single one is aware of the others. This participant can create publishers and subscribers of any topic (static or dynamic) as ordinary participants do. Servers can link to other servers in order to share its clients information.
  - **BACKUP** generates a *server* participant with additional functionality over **SERVER**. Specifically, it uses a database to backup its client information, so that if for whatever reason it disappears, it can be automatically restored and continue spreading metatraffic to late joiners. A **SERVER** in the same scenario ought to collect client information again, introducing a recovery delay.
- `RemoteServerList_t m_DiscoveryServers` lists the servers linked to the participant. This member has only significance if `discoveryProtocol` is **CLIENT**, **SERVER** or **BACKUP**. These member elements are `RemoteServerAttributes` objects that identify each server and report where the servers can be reached:
  - `GuidPrefix_t guidPrefix` is the RTPS unique identifier of the server participant we want to link to. There is a *ReadguidPrefix* method to easily fill in this member from a string formatted like *"4D.49.47.55.45.4c.5f.42.41.52.52.4f"* (note that each octet must be a valid hexadecimal figure).

- `metatrafficUnicastLocatorList` and *metatrafficMulticastLocatorList* are ordinary *LocatorList_t* (see *Fast DDS* documentation) where the server's locators must be specified. At least one of them should be populated.
- `Duration_t discoveryServer_client_syncperiod` specifies the time span of PDP metatraffic exchange, and has only significance if `discoveryProtocol` is **CLIENT**, **SERVER** or **BACKUP**. The default value is half a second.

4.1.2 RTPS schema elements dealing with discovery services

Each of the attributes in *Fast DDS* has its equivalent in the XML profiles. XML profiles make it possible to avoid tiresome hard-coded settings within application sources using XML configuration files. The fast XML schema was duly updated to accommodate the new client-server attributes:

- The participant profile `rtps` tag contains a new optional `prefix` tag where the server `GuidPrefix_t` must be specified. Any other discovery selection as simple or clients may disregard this member.
- The participant profile `builtin` tag contains a `discovery_config` tag where all discovery-related info is gathered. This new tag contains the following new XML child elements: - <`discoveryProtocol`>: specifies the discovery type through the `DiscoveryProtocol_t` enumeration. - <`discoveryServersList`>: specifies the server or servers linked with a Client/Server. - <`clientAnnouncementPeriod`>: specifies the time span between PDP metatraffic exchange.

An XML profiles examples using this new tags can be found *here*.

CHAPTER

FIVE

USAGE

Each setting in *Fast DDS* can be configured through XML profiles. XML profiles allows to avoid tiresome hard-coded settings within applications sources using XML configuration files. The *Fast DDS* XML schema was duly updated to accommodate the new Discovery Server tool settings. Please refer to *Configuration files* for more information on the new Discovery Server xml configuration files. Moreover, an XML configuration file example can be found *here*.

The discovery server binary (named after the pattern `discovery-server-X.X.X(d)` where `X.X.X` is the version number and the optional *d* denotes a debug builds) is set up from one XML profiles files passed as command-line arguments. To have the tool accessible in the terminal session it is necessary to source the setup file.

- **Linux**

```
$ source <path/to/discovery-server-ws>/discovery-server-ws/install/setup.bash
$ discovery-server-X.X.X(d) config_file.xml
```

- **Windows**

```
> . <path\to\discovery-server-ws>\discovery-server-ws\install\setup.bat
> discovery-server-X.X.X(d).exe config_file.xml
```

CHAPTER

SIX

CONFIGURATION FILES

Discovery-server operation is managed from an XML configuration file that follows the Discovery Server XSD schema, which is an extension of the *Fast DDS* XML schema. The discovery-server main goals are:

- Simplify the configuration of *Fast DDS* servers. Using a *Fast DDS* participant profile for each server is tiresome, given the large number of boilerplate code to move around. New XML syntax extensions are introduced to ease this task.
- Provide a flexible testing tool for the Discovery Server discovery mechanism. Testing the discovery involves creating a large number of participants, publishers, subscribers that use specific topics and types (static or dynamic ones) over different transports. Besides, all these entities may be instantiated or removed at different times, giving the possibility to check the discovery status (collective participant knowledge) at any of these times.

The outermost XML tag is `DS`. It admits an optional boolean attribute called `user_shutdown` that defaults to *true*. By default, the discover-server binary runs indefinitely until the user decides to shutdown. This default behavior is suitable for practical applications but not for testing. Test XML files use `user_shutdown="false"`, which grants that the discovery server is closed as soon as the test is fulfilled. The `DS` tag can contain the following tags:

- `profiles`: is plainly the *Fast DDS* profiles. It can be used to fine-tune the server operation. Please refer to the Fast DDS documentation for further information on the `profiles` element.
- `servers`: is a list of servers that the discovery-server must create and setup. It must contain at least a `server` tag. Each server admits the following attributes:
  - `name`: non-mandatory but advisable for debugging purposes.
  - `prefix`: server unique identifier. It is optional because it may be specified in the profile. By using this attribute the generation of server profiles that only differ in prefix can be avoided.
  - `profile_name`: identifies the profile associated with this server. It is a mandatory.
  - `persist`: specifies if the participant is a *SERVER*) or a *BACKUP*.
  - `creation_time`: specifies in seconds when a server must be created. It is introduced for testing purposes.
  - `removal_time`: specifies in seconds when a server must be destroyed. It is introduced for testing purposes.

  Each server element admits the following tags:
  - `ListeningPorts`: contains lists of locators where this server will listen for incoming client metatraffic.
  - `ServersList`: contains at least one `RServer` tag that references the servers this one wants to link to. `RServer`: only has a prefix attribute. Based on this prefix the discover-server parser would search for the corresponding server locators within the config file.
  - `publisher`: introduced for testing purposes. Creates a dummy publisher characterized by `profile_name`, `topic`, `creation_time`, and `removal_time`.
  - `subscriber`: introduced for testing purposes. Creates a dummy publisher characterized by `profile_name`, `topic`, `creation_time`, and `removal_time`.

- `clients` introduced for testing purposes. It is a list of dummy clients that the Discovery Server tool must create and set up. It must contain at least a `client` tag. Each client admits the following attributes:
  - `name`: non-mandatory but advisable for debugging purposes.
  - `profile_name`: identifies the profile associated with this server is a mandatory one.
  - `server` specifies the prefix of the server we want to link to. This optional attribute saves us the nuisance of creating a `ServerList` (only if this client references a single server). Based on this prefix the Discovery Server parser would search for the corresponding server locators within the config file.
  - `listening_port`: specifies a physical port where to listen for incoming traffic. This attribute is mandatory in TCP transport (client wouldn't receive other clients traffic without it). When using the `TCPv4` the format is: `[XXX.XXX.XXX.XXX:]XXXX` where the IP address is the client's WAN address that must be specified if the client has to be reachable from outside a local NAT.
  - `creation_time`: specifies in seconds when a server must be created. It is introduced for testing purposes.
  - `removal_time`: specifies in seconds when a server must be destroyed. It is introduced for testing purposes.

  Each client element admits the following tags:
  - `ServersList` contains at least one `RServer` tag that references the servers this one wants to link to. `RServer` only has a prefix attribute. Based on this prefix the discover-server parser would search for the corresponding server locators within the config file.
  - `publisher` introduced for testing purposes. Creates a publisher characterized by `profile_name`, `topic`, `creation_time`, and `removal_time`.
  - `subscriber` introduced for testing purposes. Creates a publisher characterized by `profile_name`, `topic`, `creation_time`, and `removal_time`.
- `topic`: is plainly the Fast DDS topics. It is introduced here for testing purposes to check how topic and type discovery info is handled by EDP. It is worth mentioning that only *HelloWorld* type is supported for now.
- `types`: configuration not supported for now. Please use `HelloWorld` topic type.
- `snapshots`: contains `snapshot` tags. Whenever a Discovery Server creates a participant (client or a server) it becomes its *listener* in the sense that all discovery info received by the participant is relayed to it. The reported discovery info is stored in a database. A `snapshot` is a *commit* of this database in a given time point. The `snapshots` element has a `file` attribute that must be filled with the filename of the XML results file. The `snapshot` tag has a single mandatory attribute `time` which specifies when the snapshot must be taken.

CHAPTER

SEVEN

BASIC XML CONFIGURATION

This example creates a server and two clients of this server. Each of the clients has a publisher and a subscriber on two different topics. In addition, a snapshot of the discovery table status is launched in the second five of execution.

```xml
<?xml version="1.0" encoding="utf-8"?>
<DS xmlns="http://www.eprosima.com/XMLSchemas/discovery-server" user_shutdown="false">

    <servers>
        <server name="server" profile_name="UDP server" />
    </servers>

    <clients>
        <client name="client1" profile_name="UDP_client1_server1">
            <publisher topic="topic1"/>
            <subscriber topic="topic2"/>
        </client>
        <client name="client2" profile_name="UDP_client2_server1">
            <subscriber topic="topic1"/>
            <publisher topic="topic2"/>
        </client>
    </clients>

    <snapshots file="test_01_trivial.snapshot">
        <snapshot time="5">test_01_trivial_snapshot_1</snapshot>
    </snapshots>

    <profiles>
        <participant profile_name="UDP_client1_server1" >
            <rtps>
                <prefix>63.6c.69.65.6e.74.31.5f.73.31.5f.5f</prefix>
                <builtin>
                    <discovery_config>
                        <discoveryProtocol>CLIENT</discoveryProtocol>
                        <discoveryServersList>
                            <RemoteServer prefix="44.49.53.43.53.45.52.56.45.52.5F.31">
                                <metatrafficUnicastLocatorList>
                                    <locator>
                                        <udpv4>
                                            <address>127.0.0.1</address>
                                            <port>01811</port>
                                        </udpv4>
```

(continues on next page)

(continued from previous page)

```
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>

    <participant profile_name="UDP_client2_server1" >
        <rtps>
            <prefix>63.6c.69.65.6e.74.32.5f.73.31.5f.5f</prefix>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>CLIENT</discoveryProtocol>
                    <discoveryServersList>
                        <RemoteServer prefix="44.49.53.43.53.45.52.56.45.52.5F.31">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>127.0.0.1</address>
                                        <port>01811</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>

    <participant profile_name="UDP server">
    <rtps>
        <prefix>44.49.53.43.53.45.52.56.45.52.5F.31</prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>127.0.0.1</address>
                        <port>01811</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
    </rtps>
    </participant>

    <topic profile_name="topic1">
```

(continues on next page)

(continued from previous page)

```
            <name>topic_1</name>
            <dataType>HelloWorld</dataType>
        </topic>

        <topic profile_name="topic2">
            <name>topic_2</name>
            <dataType>HelloWorld</dataType>
        </topic>

    </profiles>
</DS>
```

CHAPTER

EIGHT

ADVANCED XML CONFIGURATION

This XML configures an advanced Discovery Server topology in which multiple servers, with publishers and subscribers, have multiple clients with publishers and subscribers in turn. This example also shows how to launch participants and endpoints during the execution of the Discovery Server tool. Under the snapshots tag are specified the times at which a snapshot of the discovery state will be taken.

```
<?xml version="1.0" encoding="utf-8"?>
<DS xmlns="http://www.eprosima.com/XMLSchemas/discovery-server" user_shutdown="false">

    <servers>
        <server name="server1" profile_name="UDP_server1" />
        <server name="server2" prefix="44.49.53.43.53.45.52.56.45.52.5f.32" profile_name=
↪"UDP_server2">
            <subscriber topic="topic1" removal_time="20"/>
        </server>
        <server name="server3" prefix="44.49.53.43.53.45.52.56.45.52.5f.33" profile_name=
↪"UDP_server3" removal_time="60">
            <subscriber topic="topic1" removal_time="10"/>
            <subscriber topic="topic2" creation_time="17"/>
        </server>
    </servers>

    <clients>
        <client removal_time="40" name="client1_server1" profile_name="UDP_client1_
↪server1">
            <publisher removal_time="30" topic="topic1"/>
        </client>
        <client creation_time="50" name="client2_server1" profile_name="UDP_client2_
↪server1">
            <publisher creation_time="50" topic="topic2"/>
        </client>
    </clients>

    <snapshots file="./test_14_disposals_remote_servers.snapshot~">

        <!-- Starting point -->
        <snapshot time="8">test_14_disposals_remote_servers_snapshot_1</snapshot>
        <!-- Remove subscriber1 from server3 -->
        <snapshot time="15">test_14_disposals_remote_servers_snapshot_2</snapshot>
        <!--
            Remove subscriber1 from server2
```

(continues on next page)

(continued from previous page)

```
            Create subscriber2 in server3
         -->
        <snapshot time="25">test_14_disposals_remote_servers_snapshot_3</snapshot>
        <!-- Remove publisher 1 from client1_server1 -->
        <snapshot time="35">test_14_disposals_remote_servers_snapshot_4</snapshot>
        <!-- Remove client1_server1 -->
        <snapshot time="45">test_14_disposals_remote_servers_snapshot_5</snapshot>
        <!--
            Create client2_server1
            Create a publisher in client2_server1
         -->
        <snapshot time="55">test_14_disposals_remote_servers_snapshot_6</snapshot>
        <!-- Remove server3 -->
        <snapshot time="65">test_14_disposals_remote_servers_snapshot_7</snapshot>

    </snapshots>

    <profiles>
        <participant profile_name="UDP_client1_server1" >
            <rtps>
                <prefix>63.6c.69.65.6e.74.31.5f.73.31.5f.5f</prefix>
                <builtin>
                    <discovery_config>
                        <discoveryProtocol>CLIENT</discoveryProtocol>
                        <discoveryServersList>
                            <RemoteServer prefix="44.49.53.43.53.45.52.56.45.52.5F.31">
                                <metatrafficUnicastLocatorList>
                                    <locator>
                                        <udpv4>
                                            <address>127.0.0.1</address>
                                            <port>14811</port>
                                        </udpv4>
                                    </locator>
                                </metatrafficUnicastLocatorList>
                            </RemoteServer>
                        </discoveryServersList>
                    </discovery_config>
                </builtin>
            </rtps>
        </participant>

        <participant profile_name="UDP_client2_server1" >
            <rtps>
                <prefix>63.6c.69.65.6e.74.32.5f.73.31.5f.5f</prefix>
                <builtin>
                    <discovery_config>
                        <discoveryProtocol>CLIENT</discoveryProtocol>
                        <discoveryServersList>
                            <RemoteServer prefix="44.49.53.43.53.45.52.56.45.52.5F.31">
                                <metatrafficUnicastLocatorList>
                                    <locator>
                                        <udpv4>
```

(continues on next page)

(continued from previous page)

```
                                        <address>127.0.0.1</address>
                                        <port>14811</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                </discovery_config>
            </builtin>
        </rtps>
    </participant>

    <participant profile_name="UDP_server1">
        <rtps>
            <prefix>44.49.53.43.53.45.52.56.45.52.5F.31</prefix>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>127.0.0.1</address>
                            <port>14811</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>

    <participant profile_name="UDP_server2">
        <rtps>
            <prefix>44.49.53.43.53.45.52.56.45.52.5F.32</prefix>
            <builtin>
                <discovery_config>
                    <discoveryServersList>
                        <RemoteServer prefix="44.49.53.43.53.45.52.56.45.52.5F.31">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>127.0.0.1</address>
                                        <port>14811</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
```

(continues on next page)

(continued from previous page)

```
                        <udpv4>
                            <address>127.0.0.1</address>
                            <port>14812</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>

    <participant profile_name="UDP_server3">
        <rtps>
            <prefix>44.49.53.43.53.45.52.56.45.52.5F.33</prefix>
            <builtin>
                <discovery_config>
                    <discoveryServersList>
                        <RemoteServer prefix="44.49.53.43.53.45.52.56.45.52.5F.32">
                            <metatrafficUnicastLocatorList>
                                <locator>
                                    <udpv4>
                                        <address>127.0.0.1</address>
                                        <port>14812</port>
                                    </udpv4>
                                </locator>
                            </metatrafficUnicastLocatorList>
                        </RemoteServer>
                    </discoveryServersList>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                    <leaseAnnouncement>DURATION_INFINITY</leaseAnnouncement>
                    <leaseDuration>DURATION_INFINITY</leaseDuration>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>127.0.0.1</address>
                            <port>14813</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>

    <topic profile_name="topic1">
        <name>topic_1</name>
        <dataType>sample_type_1</dataType>
    </topic>

    <topic profile_name="topic2">
        <name>topic_2</name>
        <dataType>HelloWorld</dataType>
    </topic>
```

(continues on next page)

(continued from previous page)

```
188
189     </profiles>
190 </DS>
```

CHAPTER

NINE

TRANSPORT PROTOCOL CONFIGURATION

9.1 UDP settings

The XML basically mimics the *UDP attribute C++ source code*:

```
<?xml version="1.0" encoding="utf-8"?>
<DS xmlns="http://www.eprosima.com/XMLSchemas/discovery-server" >

  <servers>
    <server name="server" profile_name="UDP server" />
  </servers>

  <profiles>

    <participant profile_name="UDP server">
      <rtps>
        <prefix>
          4D.49.47.55.45.4c.5f.42.41.52.52.4f
        </prefix>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
                <leaseDuration>
                    <sec>DURATION_INFINITY</sec>
                </leaseDuration>
            </discovery_config>
            <metatrafficUnicastLocatorList>
            <locator>
                <udpv4>
                <!-- UDP address placeholder -->
                <address>192.168.1.113</address>
                <port>64863</port>
                </udpv4>
            </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
      </rtps>
    </participant>

  </profiles>
```

(continues on next page)

(continued from previous page)

```
</DS>
```

- Server prefix is specified.
- Discovery kind set to SERVER.
- Metatraffic locators set to the UDP listening port.

Note: `leaseDuration` is set to `INFINITY` in order to mimic the HelloWorldExample participants but can be whatever value without affecting the discovery operation.

9.2 TCP settings

The XML basically mimics the *TCP attribute C++ source code*:

```
<?xml version="1.0" encoding="utf-8"?>
<DS xmlns="http://www.eprosima.com/XMLSchemas/discovery-server" >
  <servers>
    <server name="server" profile_name="TCP server" />
  </servers>

  <profiles>
      <transport_descriptors>
          <transport_descriptor>
              <transport_id>TCPv4_SERVER</transport_id>
              <type>TCPv4</type>
              <listening_ports>
                  <port>64863</port>
              </listening_ports>
          </transport_descriptor>
      </transport_descriptors>

    <participant profile_name="TCP server">
      <rtps>
        <prefix>4D.49.47.55.45.4c.5f.42.41.52.52.4f</prefix>
        <userTransports>
              <transport_id>TCPv4_SERVER</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
        <builtin>
            <discovery_config>
                <discoveryProtocol>SERVER</discoveryProtocol>
                <leaseDuration>
                    <sec>DURATION_INFINITY</sec>
                </leaseDuration>
            </discovery_config>
            <metatrafficUnicastLocatorList>
            <locator>
                <tcpv4>
```

(continues on next page)

(continued from previous page)

```
                    <!-- if no address is provided the server would export all its␣
↪public interfaces as address -->
                    <!-- this is a logical port, the physical one is specify as␣
↪listening port above -->
                    <port>65215</port>
                </tcpv4>
            </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
      </rtps>
    </participant>
  </profiles>
</DS>
```

- A TCP transport descriptor is created specifying the physical listening port as 9843.
- The above transport descriptor is added to the participant user transports.
- Builtin transport is disabled to avoid UDP operation. This wouldn't disturb TCP communication in any way and is specified merely to prove that the actual discovery traffic is not going through UDP.
- Server prefix is specified
- Discovery kind set to SERVER.
- Metatraffic locators set to the logical listening port. The real TCP locator is provided in the transport this one is merely a port number that is linked with this particular server.

Note: `leaseDuration` is set to `INFINITY` in order to mimic the HelloWorldExample participants but can be whatever value without affecting the discovery operation.

9.3 UDP and TCP simultaneously

The XML config generates a server able to listen simultaneously on TCP or UDP ports. It mixes concepts from previous UDP and TCP config files:

```
<?xml version="1.0" encoding="utf-8"?>
<DS xmlns="http://www.eprosima.com/XMLSchemas/discovery-server" >

  <servers>
    <server name="tcp-udp server" profile_name="TCP-UDP server" />
  </servers>

  <profiles>
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>TCPv4_SERVER</transport_id>
                <type>TCPv4</type>
                <listening_ports>
                    <port>64863</port>
                </listening_ports>
```

(continues on next page)

(continued from previous page)

```
            </transport_descriptor>
        </transport_descriptors>

        <participant profile_name="TCP-UDP server">
            <rtps>
            <prefix>
                4D.49.47.55.45.4c.5f.42.41.52.52.4f
            </prefix>
            <userTransports>
                    <transport_id>TCPv4_SERVER</transport_id>
            </userTransports>
            <useBuiltinTransports>true</useBuiltinTransports>
            <builtin>
                <discovery_config>
                    <discoveryProtocol>SERVER</discoveryProtocol>
                    <leaseDuration>
                        <sec>DURATION_INFINITY</sec>
                    </leaseDuration>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <tcpv4>
                            <!-- Placeholder for server address -->
                            <address>192.168.1.113</address>
                            <!-- This is a logical port, the physical one was specified as listening port -->
                            <port>65215</port>
                        </tcpv4>
                    </locator>
                    <locator>
                        <udpv4>
                            <!-- Placeholder for server address -->
                            <address>192.168.1.113</address>
                            <port>64863</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
            </rtps>
        </participant>

  </profiles>

</DS>
```

- A TCP transport descriptor is created specifying the physical listening port as 9843.
- The above transport descriptor is added to the participant user transports.
- Builtin transport is not disabled in order to allow UDP traffic.
- Server prefix is specified
- Discovery kind set to SERVER.

- Metatraffic locators set to the logical TCP listening port and UDP actual IP address and listening port.

Using this last config XML file to generate a server allows, not only that participants with the same transport (either UDP or TCP) discover each other, but that all participants (disregarding selected transport) discover each other. A publisher in a TCP participant can match a subscriber in a TCP one (cannot exchange data due to the configuration of the HelloWorldExample Clients; only one transport is selected).

CHAPTER

TEN

C++ EXAMPLE APPLICATION

The *eProsima Fast DDS HelloWorldExample* has been updated to illustrate the Discovery Server functionality. Its installation details are explained in the installation section. Basically, the DDS DomainParticipants are now *Clients* and can only discover each other when a *Server* participant is created.

As usual, publishers and subscribers are launched by running the *HelloWorldExampleDS* executable with the corresponding *publisher* or *subscriber* argument. Each publisher and subscriber is launched within its own participant, but now the `HelloWorldPublisher::init()` and `HelloWorldSubscriber::init()` functions are modified to create clients and add the server address specified by command line (see *LAN testing using HelloWorldExampleDS*).

10.1 HelloWorldExample command line syntax

The environmental variables must be appropriately set up as explained in the *Linux installation* and *Windows installation* by employing a colcon generated script file. For colcon builds the relative path to the script from the example directory would be:

- **Linux**

```
$ . ../../../../../local_setup.bash
```

- **Windows**

```
> ..\..\..\..\..\local_setup.bat
```

Otherwise, modify the console `PATH` or the `LIB_PATH_DIR` environmental variables to allow the example binary to locate *Fast DDS* shared libraries.

The command-line syntax is the usual one for the *HelloWorldExample*, although a new flag `-t` or `--tcp` is introduced to enforce the use of TCP transport:

- **Linux**

```
$ ./HelloWorldExampleDS publisher|subscriber|server [ -h | -t | -c [<num>] | -i [
↪<num>]  | -l ip[:port] ]
```

- **Windows**

```
> HelloWorldExampleDS publisher|subscriber|server [ -h | -t | -c [<num>] | -i [<num>
↪] | -l ip[:port] ]
```

SHORTCUT	FLAG	MEANING
-h	--help	Produce help message
-t	--tcp	Use TCP transport instead of the default UDP one
-c <num>	--count=<num>	Number of datagrams to send (0=infinite) defaults to 10
-i <num>	--Interval=<num>	Time between samples in milliseconds defaults to 100
-l <ip[:port]>	--ip=<ip[:port]>	Server address and physical port

Additionally to the Publisher and Subscriber instances, a Server participant must be launched in order to allow publishers and subscribers to discover each other. A simple test would be as follows:

- **Linux**
  - Terminal 1:

```
$ ./HelloWorldExampleDS publisher
```

  - Terminal 2:

```
$ ./HelloWorldExampleDS subscriber
```

  - Terminal 3:

```
$ ./HelloWorldExampleDS server
```

- **Windows**
  - Console 1:

```
> HelloWorldExampleDS publisher
```

  - Console 2:

```
> HelloWorldExampleDS subscriber
```

  - Console 3:

```
> HelloWorldExampleDS server
```

The HelloWorldExampleDS Server instance can be replaced by a Discovery Server instance that creates a suitable Server. Thus instead of running `HelloWorldExampleDS server`, it can be done running the following commands:

- **Linux**

```
$ ./discovery-server-X.X.X(d) config-file.xml
```

- **Windows**

```
> discovery-server-X.X.X(d).exe config-file.xml
```

being the `config-file.xml`,

```
<participant profile_name="UDP server">
    <rtps>
        <prefix>44.49.53.43.53.45.52.56.45.52.5F.31</prefix>
        <builtin>
            <discovery_config>
```

(continues on next page)

(continued from previous page)

```
                    <discoveryProtocol>SERVER</discoveryProtocol>
                </discovery_config>
                <metatrafficUnicastLocatorList>
                    <locator>
                        <udpv4>
                            <address>127.0.0.1</address>
                            <port>01811</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
        </rtps>
    </participant>
```

10.1.1 LAN testing using HelloWorldExampleDS

First, the Server network address and its physical port must be known. In this example it would be *192.168.1.113:64863*. The UDP protocol is used as the default transport protocol, but it is possible to change it to the TCP protocol by adding the *–tcp* flag to the following commands:

- **Linux**
  - Terminal 1:

```
$ . ../../../../../local_setup.bash
$ ./HelloWorldExampleDS publisher --count=0 --ip=192.168.1.113:64863
```

    by specifying `--count=0` the publisher keeps publishing samples forever.

  - Terminal 2:

```
$ . ../../../../../local_setup.bash
$ ./HelloWorldExampleDS subscriber --ip=192.168.1.113:64863
```

  - Terminal 3:

```
$ . ../../../../../local_setup.bash
$ ./HelloWorldExampleDS server --ip=0.0.0.0:64863
```

- **Windows**
  - Console 1:

```
> ..\..\..\..\..\local_setup.bat
> HelloWorldExampleDS publisher --count=0 --ip=192.168.1.113:64863
```

    by specifying `--count=0` the publisher keeps publishing samples forever.

  - Console 2:

```
> ..\..\..\..\..\local_setup.bat
> HelloWorldExampleDS subscriber --ip=192.168.1.113:64863
```

  - Console 3:

```
> ..\..\..\..\..\local_setup.bat
> HelloWorldExampleDS server --ip=0.0.0.0:64863
```

Note that by using `0.0.0.0` as IP address, the server is forced to publish its metatraffic information through all the local interfaces (`192.168.1.133` would be one of then in this example). The clients, once received the server metadata, would choose the fastest interface among the server's interfaces. Of course, the server can be configured to use single interface by doing `--ip=192.168.1.133:64863`. Finally, specifying the localhost network address as the interface (`--ip=127.0.0.1:64863`) only local clients will be able to reach the server.

Note: If no port number is provided a default one will used (`11811`).

CHAPTER

ELEVEN

UDP TRANSPORT ATTRIBUTE SETTINGS

To use UDP, the application relies on the default transport where the locators are actual ports and IP addresses.

11.1 UDP transport code setup for a Client

According to the *former RTPS attributes explanation*, the `DiscoverySettings discovery_config` must be populated specifying `DiscoveryProtocol_t::CLIENT` and adding a new `RemoteServerAttributes` object to the `m_DiscoveryServers` list. In this case the UDP port 64863 is set as is the server prefix.

```
RemoteServerAttributes ratt;
ratt.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");

ParticipantAttributes PParam;
PParam.rtps.builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_t::CLIENT;
PParam.rtps.builtin.domainId = 0;
PParam.rtps.builtin.discovery_config.leaseDuration = c_TimeInfinite;
PParam.rtps.setName("Participant_pub");

// Placeholder values for the server address
Locator_t server_address(LOCATOR_KIND_UDPv4, 64863);
IPLocator::setIPv4(server_address, 192, 168, 1, 113);

ratt.metatrafficUnicastLocatorList.push_back(server_address);
PParam.rtps.builtin.discovery_config.m_DiscoveryServers.push_back(ratt);

mp_participant = Domain::createParticipant(PParam);
```

11.2 UDP transport code setup for a server

According to the *former RTPS attributes explanation*, the `DiscoverySettings discovery_config` specifying we want to create a `DiscoveryProtocol_t::SERVER` and adding a new listening locator to any `BuiltinAttributes` metatraffic lists (this locator or locators must be known by the Clients). In this case, the UDP port 64863 is set as is the Server prefix.

```
ParticipantAttributes PParam;
PParam.rtps.builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_t::SERVER;
PParam.rtps.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");
PParam.rtps.builtin.domainId = 0;
```

(continues on next page)

(continued from previous page)

```
PParam.rtps.builtin.discovery_config.leaseDuration = c_TimeInfinite;
PParam.rtps.setName("Participant_server");

// Placeholder values for the server address
Locator_t server_address(LOCATOR_KIND_UDPv4, 64863);
IPLocator::setIPv4(server_address, 192, 168, 1, 113);

PParam.rtps.builtin.metatrafficUnicastLocatorList.push_back(server_address);

mp_participant = Domain::createParticipant(PParam);
```

CHAPTER

TWELVE

TCP TRANSPORT ATTRIBUTE SETTINGS

For TCP transport is mandatory to disable the default transport setting the `RTPSParticipantAttributes::useBuiltinTransports` as `false` and creating a new *transport descriptor* thus *Fast DDS* framework might create a suitable transport object.

12.1 TCP transport code setup for a client

The `DiscoverySettings discovery_config` is almost the same as in *UDP client case*. Note that here the `server_address` locator specifies 65215 as the logical port and 9843 as the physical one. The reason behind this is that TCP transport was devised in order to allow a single TCP connection tunnel several participants traffic through it. In order to differentiate each participant sharing the connection, a *logical port concept* was introduced. The transport will understand that must connect to the physical port (using TCP protocol) and relay metatraffic to the logical port 65215, which is the metatraffic mailbox of the Server.

A new `TCPv4TransportDescriptor` must be created and a physical listening port selected. In this case, each *HelloWorldExample* instance creates a single participant thus the linked process ID is a suitable seed to make up a listening port number (this way each time a new Client is created a different port is selected).

```
RemoteServerAttributes ratt;
ratt.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");

ParticipantAttributes PParam;
PParam.rtps.builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_t::CLIENT;
PParam.rtps.builtin.domainId = 0;
PParam.rtps.builtin.discovery_config.leaseDuration = c_TimeInfinite;
PParam.rtps.setName("Participant_pub");

// Placeholder values for the server address
Locator_t server_address;
server_address.kind = LOCATOR_KIND_TCPv4;
IPLocator::setLogicalPort(server_address, 64863);
IPLocator::setPhysicalPort(server_address, 9843);
IPLocator::setIPv4(server_address, 192, 168, 1, 113);

ratt.metatrafficUnicastLocatorList.push_back(server_address);
PParam.rtps.builtin.discovery_config.m_DiscoveryServers.push_back(ratt);

PParam.rtps.useBuiltinTransports = false;
std::shared_ptr<TCPv4TransportDescriptor> descriptor = std::make_shared
↪<TCPv4TransportDescriptor>();
```

(continues on next page)

(continued from previous page)

```
// Generate a listening port for the client
std::default_random_engine gen(System::GetPID());
std::uniform_int_distribution<int> rdn(49152, 65535);
descriptor->add_listener_port(rdn(gen)); // IANA ephemeral port number

descriptor->wait_for_tcp_negotiation = false;
PParam.rtps.userTransports.push_back(descriptor);

mp_participant = Domain::createParticipant(PParam);
```

12.2 TCP transport code setup for a server

The `DiscoverySettings discovery_config` is almost the same as in *UDP server case*. Here the *server_address* locator specifies 64863 as the logical port instead of the physical one.

A new `TCPv4TransportDescriptor` must be created and a physical listening port selected. Unlike the client code, this listening port (9843 in the example) must be known beforehand for all Clients in order to successfully deliver metatraffic to the server.

```
ParticipantAttributes PParam;
PParam.rtps.builtin.discovery_config.discoveryProtocol = DiscoveryProtocol_t::SERVER;
PParam.rtps.ReadguidPrefix("4D.49.47.55.45.4c.5f.42.41.52.52.4f");
PParam.rtps.builtin.domainId = 0;
PParam.rtps.builtin.discovery_config.leaseDuration = c_TimeInfinite;
PParam.rtps.setName("Participant_server");

// Placeholder values for the server address
Locator_t server_address;
server_address.kind = LOCATOR_KIND_TCPv4;
IPLocator::setLogicalPort(server_address, 64863);
IPLocator::setIPv4(server_address, 192, 168, 1, 113);

PParam.rtps.builtin.metatrafficUnicastLocatorList.push_back(server_address);

std::shared_ptr<TCPv4TransportDescriptor> descriptor = std::make_shared
↪<TCPv4TransportDescriptor>();
descriptor->wait_for_tcp_negotiation = false;
descriptor->add_listener_port(9843);

PParam.rtps.useBuiltinTransports = false;
PParam.rtps.userTransports.push_back(descriptor);

mp_participant = Domain::createParticipant(PParam);
```

CHAPTER

THIRTEEN

VERSION 2.0.0

This version mainly updates the validation system to adapt it to the new version of the Discovery Server discovery mechanism released in eProsima Fast DDS 2.0.2.

- Updated the entire test suite.
- Validation system based on expected and fixed test results.
- Removed the snapshots validation mode since the discovery databases of the participants are unique.

13.1 Previous versions

13.1.1 Version 1.0.0

First release.

- Server creation and setup capabilities.
- Discovery testing capabilities.
- Single process suite of tests added.
- Multi-process test capability added (generation and validation of snapshots serialized as XML).
- HelloWorld example over UDP and TCP.
- Extended schema that simplifies setup for server creation and testing purposes.