# Discovery Server Documentation

***Release 1.0.0***

**eProsima**

**Jul 15, 2019**

# Installation

The Current DDS-RTPS standard in its section 8.5 specifies a non-centralized, distributed simple discovery mechanism
for RTPS. 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 high number of endpoint entities are continuously entering and exiting a large network.
- a network without multicasting capabilities.

In order to cope with the above issues, the Fast-RTPS discovery mechanism was extended with a client-server functionality. Besides, to simplify the management and testing of this new functionality this discovery-server application was devised.

This documentation is organized into the following sections:

# CHAPTER 1

# Installation

**Table of Contents**

## 1.1 Dependencies

In order to use discovery server, its necessary have a compatible version of Fast RTPS installed (over release 1.9.0). Fast RTPS dependencies as tinyxml must be accessible, either because Fast RTPS was build-installed defining THIRD-PARTY=ON or because those libraries have been specifically installed.

The well known cross-platform tool colcon was chosen to simplify the installation of the several mutually dependent CMake projects. In order to use colcon, python and CMake must be already installed as detailed in the corresponding hyperlinks.

A discovery-server.repos file is available in order to profit from vcstool capabilities to download the needed repositories. Once vcstool python package is installed download the sources is as easy as download the discovery-server.repos and call:

```
[SOURCES]$ vcs import --input discovery-server.repos
```

on the **[SOURCES]** directory where the user wants to keep the repositories.

In order to avoid using vcstool the following repositories should be downloaded from github into **SOURCES**:

| | |
|---|---|
| eProsima/Fast-CDR: | https://github.com/eProsima/Fast-CDR.git |
| eProsima/Fast-RTPS: | https://github.com/eProsima/Fast-RTPS.git |
| eProsima/Discovery-Server: | https://github.com/eProsima/Discovery-Server.git |
| leethomason/tinyxml2: | https://github.com/leethomason/tinyxml2.git |

We also assume that the user wants to keep the build, log and installation files in a separate directory called **[BUILD]**. If this is not the case, flag **–base-paths [SOURCES]** can be ignored in what follows.

## 1.2 Installation steps

Discovery Server supports the same platforms supported by Fast-RTPS: Windows, Linux and Mac. We proceed to detail each platform specifics.

### 1.2.1 Linux setup steps

Valid placeholders for the Linux example may be:

| PLACEHOLDER | EXAMPLE PATHS |
|---|---|
| [SOURCES] | /home/username/Documents/colcon_sources |
| [BUILD] | /home/username/Documents/colcon_build |

1. Create directory **[BUILD]** where we want to keep the build, install and log compilation results.
2. Compile using the colcon tool. Choose the build configuration by declaring CMAKE_BUILD_TYPE as Debug or Release. In this example we have chosen Debug which would be the choice of advance users for debugging purposes:

```
[BUILD]$ colcon build --base-paths [SOURCES] --packages-up-to discovery-server --
↪cmake-args -DTHIRDPARTY=ON -DLOG_LEVEL_INFO=ON -DCOMPILE_EXAMPLES=ON -DCMAKE_BUILD_
↪TYPE=Debug
```

3. In order to run the tests use the following command:

```
[BUILD]$ colcon test --base-paths [SOURCES] --packages-select discovery-server
```

note that only the test matching the build (step 2) configuration would run.

4. To run the example navigate to directory **[BUILD]**/install/discovery-server/examples/HelloWorldExampleDS/bin. The configuration bash file located in the install folder must be run first in order to set the required environment variables:

```
[BUILD]/install/discovery-server/examples/C++/HelloWorldExampleDS/bin$ . ../../../../.
↪./local_setup.bash
```

In order to test the example open three terminals and run the above command. Then launch the application with different arguments:

```
[BUILD]/install/discovery-server/examples/HelloWorldExampleDS/bin$ ./
↪HelloWorldExampleDS publisher
[BUILD]/install/discovery-server/examples/HelloWorldExampleDS/bin$ ./
↪HelloWorldExampleDS subscriber
[BUILD]/install/discovery-server/examples/HelloWorldExampleDS/bin$ ./
↪HelloWorldExampleDS server
```

### 1.2.2 Windows setup steps

Valid placeholders for the windows example may be:

| PLACEHOLDER | EXAMPLE PATHS |
|---|---|
| [SOURCES] | C:\Users\username\Documents\colcon_sources |
| [BUILD] | C:\Users\username\Documents\colcon_build |

1. Create directory **[BUILD]** where you want to keep the build, install and log compilation results.
2. If the generator (compiler) of choice is Visual Studio, launch colcon from a visual studio console. Any console can be set up 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
```

3. Compile using the colcon tool. If using a multi-configuration generator like Visual Studio we recommend to build both in debug and release modes:

```
[BUILD]> colcon build --base-paths [SOURCES] --packages-up-to discovery-server --
↪cmake-args -DTHIRDPARTY=ON -DLOG_LEVEL_INFO=ON -DCOMPILE_EXAMPLES=ON -DCMAKE_BUILD_
↪TYPE=Debug
[BUILD]> colcon build --base-paths [SOURCES] --packages-up-to discovery-server --
↪cmake-args -DTHIRDPARTY=ON -DCOMPILE_EXAMPLES=ON -DCMAKE_BUILD_TYPE=Release
```

If using a single configuration tool just make the above call with your configuration of choice.

4. In order to run the tests in a multi-configuration generator like Visual Studio use the following command:

```
[BUILD]> colcon test --base-paths [SOURCES] --packages-select discovery-server --
↪ctest-args -C Debug
```

Here, –ctest-args allows specifying the configuration (Debug or Release) of interest (names are case sensitive). If using a single configuration tool this flag has no effect, as only the test matching the build (step 3) configuration would run.

5. In order to run the example, navigate to the directory **[BUILD]**\install\discovery-server\examples\HelloWorldExampleDS\bin and run the executable, running first the configuration bat file located within the install folder in order to set required environment variables:

```
[BUILD]\install\discovery-server\examples\C++\HelloWorldExampleDS\bin>..\..\..\..\..
↪\local_setup.bat
```

To test the helloworld example open three consoles, run the above bat file and launch the application with different arguments:

```
[BUILD]\install\discovery-server\examples\C++\HelloWorldExampleDS\bin>␣
↪HelloWorldExampleDS publisher
[BUILD]\install\discovery-server\examples\C++\HelloWorldExampleDS\bin>␣
↪HelloWorldExampleDS subscriber
[BUILD]\install\discovery-server\examples\C++\HelloWorldExampleDS\bin>␣
↪HelloWorldExampleDS server
```

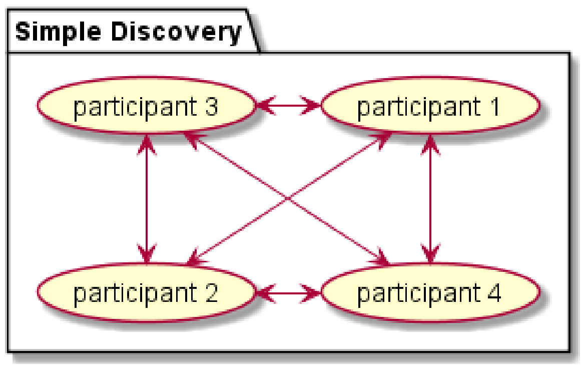
CHAPTER 2

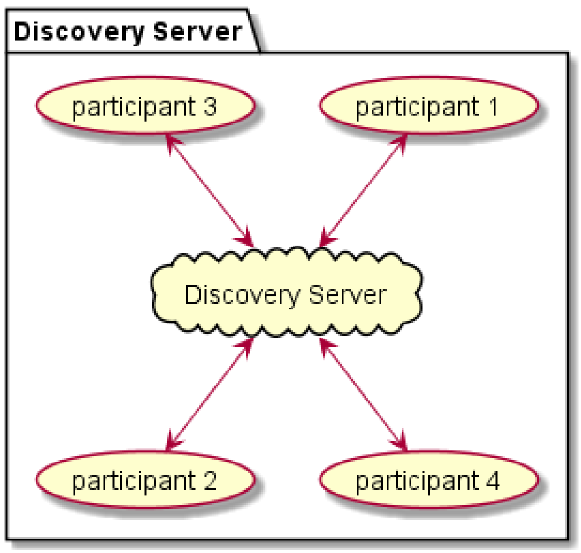
# Executable use

**Table of Contents**

## 2.1 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 don't 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.

## 2.1.1 RTPS Attributes dealing with discovery services

Several Fast-RTPS 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-RTPS library.

### RTPSParticipantAttributes

- A *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

- A **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.

- A **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 octec must be a valid hexadecimal figure).
  - **metatrafficUnicastLocatorList** and *metatrafficMulticastLocatorList* are ordinary *LocatorList_t* (see Fast-RTPS 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.

## 2.1.2 RTPS schema elements dealing with discovery services

Each of the attributes in Fast-RTPS 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 elements:
  - a **discoveryProtocol** tag, where the discovery type can be specified through the *DiscoveryProtocol_t* enumeration quoted *above*.
  - a **discoveryServersList** tag, where the server or servers linked with a participant can be specified.
  - a **clientAnnouncementPeriod** tag, where the time span between PDP metatraffic exchange can be specified.

Below we provide an example XML participant profile using this new *tags*:

```xml
<participant profile_name="UDP client" >
  <rtps>
    <builtin>
        <discovery_config>
          <discoveryProtocol>CLIENT</discoveryProtocol>
          <discoveryServersList>
            <RemoteServer prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f">
              <metatrafficUnicastLocatorList>
                <locator>
                  <udpv4>
                    <address>127.0.0.1</address>
                    <port>64863</port>
                  </udpv4>
                </locator>
              </metatrafficUnicastLocatorList>
            </RemoteServer>
          </discoveryServersList>
        </discovery_config>
    </builtin>
  </rtps>
```

(continues on next page)

(continued from previous page)

```
    </participant>

    <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>
            </discovery_config>
            <metatrafficUnicastLocatorList>
                <locator>
                    <udpv4>
                        <address>127.0.0.1</address>
                        <port>64863</port>
                    </udpv4>
                </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
      </rtps>
    </participant>
```

## 2.2 Directions for use

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 or several XML config files passed as command-line arguments. There are two modes of execution:

- *Processing* mode. In this mode a single config file answering to the *discovery-server.xsd <../../schemas/discovery-server.xsd>* _schema. This schema is basically an extension of the fast RTPS one that simplifies the creation of custom servers and provides tools for easily testing specific discovery scenarios.

  When using the discovery server with testing purposes one may:

  - immediately validate when test execution is finished. This is the usual case when testing a single process scenario.
  - not validate the test results and generate an XML file with the test results. This results file follows a specific ds-snapshot.xsd schema. This mode is activated by passing a filename in the input config XML. It's the usual case when testing a multiprocess or multimachine scenario. Each process or machine will generate a results XML file (with each one's discovery information record) and all of them would be validated later by running the discovery server binary in *validation* mode.

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

- *Validation* mode. Only for testing purposes. In this mode several XML results files generated on *Processing* mode would be compared to each other in other to assess if all discovery information was properly propagated by the server or servers involved in the testing.

  ```
  > discovery-server-X.X.X(d).exe results_file_1.xml results_file_2.xml␣
  ↪results_file_3.xml ...
  ```

Note that if the colcon deployment strategy described in section *Installation steps <installation.html#installation-steps>* was followed, before using the binary we must setup the appropriate environmental variables using colcon generated scripts:

Linux:

```
[BUILD]/install/discovery-server/bin$ . ../../local_setup.bash
```

Windows:

```
[BUILD]\install\discovery-server\bin>..\..\local_setup.bat
```

where:

- the local_setup batch sets up the environment variables for the binary execution.
- the discovery-server binary name depends on build configuration (debug introduces *d* postfix) and version number.
- the config_file.xml is a placeholder for any XML config file that follows the discovery-server.xsd.

CHAPTER 3

# Configuration files

Discovery-server operation is managed from an XML config file that follows the **discovery-server.xsd** schema located in **[SOURCES]**/discovery-server/resources/xsd (this is an extension of the Fast-RTPS XML schema). The discovery-server main goals are:

- Simplify the configuration of Fast-RTPS servers. Using a Fast-RTPS 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 client-server discovery implementation. 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 appear or disappear at different times, and we need to be able 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-RTPS profiles. We can use them to fine-tune the server operation. The associated documentation can be found fast RTPS docs, note the updates introduced in the attributes section.
- **servers** is a list of servers that the discovery-server must create and setup. 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's optional because it may be specified in the profile but using this attribute we can avoid generating server profiles that only differ in prefix.
  - **profile_name** identifies the profile associated with this server is a mandatory one.
  - **persist** specifies if the participant is a SERVER) or a BACKUP.
  - **creation_time** introduced for testing purposes specifies in seconds when a server must be created.
  - **removal_time** introduced for testing purposes specifies in seconds when a server must be destroyed.

  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's a list of dummy clients that the discovery-server must create and set up. 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 discover-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 we want the client to be reachable from outside a local NAT.
  - **creation_time** introduced for testing purposes specifies in seconds when a client must be created.
  - **removal_time** introduced for testing purposes specifies in seconds when a client must be destroyed.

  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*.

- **types** is plainly the Fast-RTPS types. It's introduced here for testing purposes to check how topic and type discovery info is handled by EDP.
- **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 him. This 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 if immediate validation is not desired (see instructions).

  *snapshots* are useful because within is recorded how much information each participant is aware of. If a participant reported no info or incomplete one there is a discovery failure. When the discovery server has finished processing the config file (creating and destroying entities or taking snapshots) then all **snapshots** taken are checked. If any of them shows discovery info leakages discovery server returns -1 and the unsuccessful *snapshot* is logged to the standard error. Note:

  - this validation can be avoided using the **file** attribute of the **snapshots** collection.

- there is a particular case that requires special treatment by using **snapshot**'s attribute **someone**. By default **someone** is true and no discovery info reported by any participant is regarded as a failure, yet in some tests, we want to validate the absence of discovery if settings are wrong so **someone** may be set false then.

**snapshot** tag has a single mandatory attribute **time** which specifies when the snapshot must be taken and an optional **someone** whose functionality is explained above. The text content of the tag is regarded as a description where the user can specify which event may require validation (participant creation or removal, etc. . . ).

The tests are probably the best examples of the above XML definitions put into practice.

CHAPTER 4

# Example application

**Table of Contents**

The Fast-RTPS **HelloWorldExample** has been updated to illustrate the client-server functionality. Its installation details are explained in the installation section. Basically, the publisher and subscriber participants are now *clients* and can only discover each other when a *server* participant is created.

As usual, we launch publishers and subscribers by running HelloWorldExampleDS.exe 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()` methods are modified to create clients and hard code the server's reference.

## 4.1 UDP transport attribute settings

In order to use UDP, we can rely on the default transport where the locators are actual ports and IP addresses.

### 4.1.1 UDP transport code setup for a client

```
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");
```

(continues on next page)

(continued from previous page)

```
Locator_t server_address(LOCATOR_KIND_UDPv4, 65215);
IPLocator::setIPv4(server_address, 127, 0, 0, 1);

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

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

Note that according to former attributes explanation we must populate the **DiscoverySettings discovery_config** specifying we want to create a **DiscoveryProtocol_t::CLIENT** and adding a new *RemoteServerAttributes* object to the *m_DiscoveryServers* list. In this case the UDP port 65215 is hardcoded as is the server prefix.

### 4.1.2 UDP transport code setup for a 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");

Locator_t server_address(LOCATOR_KIND_UDPv4, 65215);
IPLocator::setIPv4(server_address, 127, 0, 0, 1);

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

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

Note that according to former attributes explanation we must populate 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 65215 is hardcoded as is the server prefix.

## 4.2 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 RTPS framework might create a suitable transport object.

### 4.2.1 TCP transport code setup for a client

```
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");
```

(continues on next page)

(continued from previous page)

```
Locator_t server_address;
server_address.kind = LOCATOR_KIND_TCPv4;
IPLocator::setLogicalPort(server_address, 65215);
IPLocator::setPhysicalPort(server_address, 9843);
IPLocator::setIPv4(server_address, 127, 0, 0, 1);

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>();

// 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);
```

The **DiscoverySettings discovery_config** is almost the same as in *UDP client case*. Note that here the *server_address* locator specifies 65215 as a logical port and 9843 as a 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 meta traffic to the logical port 65215 which is the meta traffic mailbox of the server we are interested in.

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).

### 4.2.2 TCP transport code setup for a 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");

Locator_t server_address;
server_address.kind = LOCATOR_KIND_TCPv4;
IPLocator::setLogicalPort(server_address, 65215);
IPLocator::setIPv4(server_address, 127, 0, 0, 1);

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);
```

(continues on next page)

(continued from previous page)

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

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

The **DiscoverySettings discovery_config** is almost the same as in *UDP server case*. Note that here the *server_address* locator specifies 65215 as a logical port instead of a 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 meta traffic to the server.

### HelloWorldExample command line syntax

The environmental variables must be appropriately set up as explained in the installation steps 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 terminal LIB_PATH_DIR environmental variables to allow the example binary to locate fast 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>
↪] ]
```

Windows:

```
> HelloWorldExampleDS publisher|subscriber|server [ -h | -t | -c [<num>] | -i [<num>]␣
↪]
```

| 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 |

The main difference with the plain HelloWorldExample is that additionally to the Publisher and Subscriber instance we must launch a server instance 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 calling `HelloWorldExample --server`, we can call:

Windows:

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

Linux:

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

As an example we show here the configuration files associated with each specific kind of transport:

## 4.3 HelloWorld_UDP_config.xml

```

    <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>
```

(continues on next page)

(continued from previous page)

```
                <address>127.0.0.1</address>
                <port>65215</port>
                </udpv4>
            </locator>
            </metatrafficUnicastLocatorList>
        </builtin>
      </rtps>
    </participant>

```

The XML basically mimics the *UDP attribute source code* showed above:

- server prefix is specified.
- discovery kind set to SERVER.
- metatrafic locators set to the UDP listening port.

Note that leaseDuration was set to INFINITY in order to mimic the HelloWorldExample usual participants but can be whatever value without affecting the discovery operation.

## 4.4 HelloWorld_TCP_config.xml

```
      <transport_descriptors>
          <transport_descriptor>
              <transport_id>TCPv4_SERVER</transport_id>
              <type>TCPv4</type>
              <listening_ports>
                  <port>9843</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>
                    <port>65215</port>
                </tcpv4>
            </locator>
```

(continues on next page)

(continued from previous page)

```xml
            </metatrafficUnicastLocatorList>
        </builtin>
      </rtps>
    </participant>

```

The XML basically mimics the *TCP attribute source code* showed above:

- 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.
- metatrafic 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.

Again, note that leaseDuration was set to INFINITY in order to mimic the HelloWorldExample usual participants but can be whatever value without affecting the discovery operation.

## 4.5 HelloWorld_UDP_TCP_config.xml

```xml
        <transport_descriptors>
            <transport_descriptor>
                <transport_id>TCPv4_SERVER</transport_id>
                <type>TCPv4</type>
                <listening_ports>
                    <port>9843</port>
                </listening_ports>
            </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>
```

(continues on next page)

(continued from previous page)

```xml
                            <address>127.0.0.1</address>
                            <port>65215</port>
                        </tcpv4>
                    </locator>
                    <locator>
                        <udpv4>
                            <address>127.0.0.1</address>
                            <port>65215</port>
                        </udpv4>
                    </locator>
                </metatrafficUnicastLocatorList>
            </builtin>
            </rtps>
        </participant>

```

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

- 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.
- metatrafic locators set to the logical TCP listening port and UDP actual IP address and listening port. Note that are both set to 65215 but this doesn't arise any problems because TCP is a logical port and has no physical meaning.

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 (they cannot exchange data although because as HelloWorldExample clients are setup only one transport is selected).

# CHAPTER 5

# Testing

**Table of Contents**

Discovery testing is done resorting to discover-server config files. Each test uses a specific XML file that tests a single or several discovery features. To automatically launch the tests using colcon, please check section installation. To manually launch a test, use the following procedure:

Linux:

```
[BUILD]/install/discovery-server/bin$ . ../../local_setup.bash
[BUILD]/install/discovery-server/bin$ ./discovery-server-X.Y.Z(d)
[SOURCES]/discovery-server/resources/xml/test_XXX.xml
```

Windows:

```
[BUILD]\install\discovery-server\bin>..\..\local_setup.bat
[BUILD]\install\discovery-server\bin>discovery-server-X.Y.Z(d) [SOURCES]\discovery-
↪server\resources\xml\test_XXX.xml
```

To view the full discovery information messages and snapshots in debug configuration, run colcon with the additional flag *-DLOG_LEVEL_INFO=ON*.

A brief description of each test is given below. Note that a detailed explanation of the XML syntax is given in section configuration files.

## 5.1 test_1_PDP_UDP.xml

This is the most simple scenario: a single server manages the discovery information of four clients. The server prefix and listening ports are given in the profiles **UDP server** and **UDP client**. A snapshot is taken after 3 seconds to assess that all clients are aware of the existence of each other.

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

<clients>
  <client name="client1" profile_name="UDP client"/>
  <client name="client2" profile_name="UDP client"/>
  <client name="client3" profile_name="UDP client"/>
  <client name="client4" profile_name="UDP client"/>
</clients>

<snapshots>
  <snapshot time="3">Check all clients met the server and know each other</snapshot>
</snapshots>
```

The snapshot information output would be something like:

```
2019-04-24 12:58:36.936 [DISCOVERY_SERVER Info] Snapshot taken at 2019-04-24 12:58:36␣
↪description: Check all clients
met the server and know each other
5 participants report the following discovery info:
Participant 1.f.1.30.ac.12.0.0.2.0.0.0|0.0.1.c1 discovered:
                 Participant client2 1.f.1.30.ac.12.0.0.3.0.0.0|0.0.1.c1
                 Participant client3 1.f.1.30.ac.12.0.0.4.0.0.0|0.0.1.c1
                 Participant client4 1.f.1.30.ac.12.0.0.5.0.0.0|0.0.1.c1
                 Participant server 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1

Participant 1.f.1.30.ac.12.0.0.3.0.0.0|0.0.1.c1 discovered:
                 Participant client1 1.f.1.30.ac.12.0.0.2.0.0.0|0.0.1.c1
                 Participant client3 1.f.1.30.ac.12.0.0.4.0.0.0|0.0.1.c1
                 Participant client4 1.f.1.30.ac.12.0.0.5.0.0.0|0.0.1.c1
                 Participant server 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1

Participant 1.f.1.30.ac.12.0.0.4.0.0.0|0.0.1.c1 discovered:
                 Participant client1 1.f.1.30.ac.12.0.0.2.0.0.0|0.0.1.c1
                 Participant client2 1.f.1.30.ac.12.0.0.3.0.0.0|0.0.1.c1
                 Participant client4 1.f.1.30.ac.12.0.0.5.0.0.0|0.0.1.c1
                 Participant server 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1

Participant 1.f.1.30.ac.12.0.0.5.0.0.0|0.0.1.c1 discovered:
                 Participant client1 1.f.1.30.ac.12.0.0.2.0.0.0|0.0.1.c1
                 Participant client2 1.f.1.30.ac.12.0.0.3.0.0.0|0.0.1.c1
                 Participant client3 1.f.1.30.ac.12.0.0.4.0.0.0|0.0.1.c1
                 Participant server 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1

Participant 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1 discovered:
                 Participant client1 1.f.1.30.ac.12.0.0.2.0.0.0|0.0.1.c1
                 Participant client2 1.f.1.30.ac.12.0.0.3.0.0.0|0.0.1.c1
                 Participant client3 1.f.1.30.ac.12.0.0.4.0.0.0|0.0.1.c1
```

(continues on next page)

(continued from previous page)

```
Participant client4 1.f.1.30.ac.12.0.0.5.0.0.0|0.0.1.c1
```

We'll only get this with the debug binary. On Release mode, we can resort to provide a filename to the **snapshots** tag. Then an XML file will be generated with the same info (note that generating an XML automatically disables validation thus, it cannot be used in singleton tests).

```
    <DS_Snapshot timestamp="11684334716598" someone="true">
        <description>Check all clients met the server and know each other</
↪description>
        <ptdb guid_prefix="1.f.74.42.80.35.0.0.2.0.0.0" guid_entity="0.0.1.c1">
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.3.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client2"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.4.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client3"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.5.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client4"/>
            <ptdi guid_prefix="4d.49.47.55.45.4c.5f.42.41.52.52.4f" guid_entity="0.0.
↪1.c1" server="true" alive="true" name="server"/>
        </ptdb>
        <ptdb guid_prefix="1.f.74.42.80.35.0.0.3.0.0.0" guid_entity="0.0.1.c1">
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.2.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client1"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.4.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client3"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.5.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client4"/>
            <ptdi guid_prefix="4d.49.47.55.45.4c.5f.42.41.52.52.4f" guid_entity="0.0.
↪1.c1" server="true" alive="true" name="server"/>
        </ptdb>
        <ptdb guid_prefix="1.f.74.42.80.35.0.0.4.0.0.0" guid_entity="0.0.1.c1">
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.2.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client1"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.3.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client2"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.5.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client4"/>
            <ptdi guid_prefix="4d.49.47.55.45.4c.5f.42.41.52.52.4f" guid_entity="0.0.
↪1.c1" server="true" alive="true" name="server"/>
        </ptdb>
        <ptdb guid_prefix="1.f.74.42.80.35.0.0.5.0.0.0" guid_entity="0.0.1.c1">
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.2.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client1"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.3.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client2"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.4.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client3"/>
            <ptdi guid_prefix="4d.49.47.55.45.4c.5f.42.41.52.52.4f" guid_entity="0.0.
↪1.c1" server="true" alive="true" name="server"/>
        </ptdb>
        <ptdb guid_prefix="4d.49.47.55.45.4c.5f.42.41.52.52.4f" guid_entity="0.0.1.c1
↪">
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.2.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client1"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.3.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client2"/>
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.4.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client3"/>
```

(continues on next page)

(continued from previous page)

```
            <ptdi guid_prefix="1.f.74.42.80.35.0.0.5.0.0.0" guid_entity="0.0.1.c1"␣
↪server="false" alive="true" name="client4"/>
        </ptdb>
    </DS_Snapshot>
```

Here we see how all participants reported the discovery of all the others. Note that, because there is no Fast-RTPS discovery callback from a participant to report its own discovery, participants do not report themselves. This must be taken into account when a snapshot is checked. Note, however, that participants do discover themselves when they create a publisher or subscriber because there are callbacks associated with those cases.

## 5.2 test_2_PDP_TCP.xml

Resembles the previous scenario but uses TCP transport instead of the default UDP one. A single server manages the discovery info of four clients. The server prefix and listening ports are given in the profiles **TCP server** and **TCP client**. A snapshot is taken after 3 seconds to assess that all clients are aware of the existence of each other.

Specific transport descriptor must be created for server and clients:

```

      <transport_descriptor>
        <transport_id>TCPv4_SERVER</transport_id>
        <type>TCPv4</type>
        <listening_ports>
          <port>02049</port>
        </listening_ports>
        <calculate_crc>false</calculate_crc>
        <check_crc>false</check_crc>
      </transport_descriptor>

      <transport_descriptor>
        <transport_id>TCPv4_CLIENT</transport_id>
        <type>TCPv4</type>
        <calculate_crc>false</calculate_crc>
        <check_crc>false</check_crc>
      </transport_descriptor>

```

Client and server participant profiles must reference this transport and discard builtin ones.

```
<participant profile_name="TCP client" >
  <rtps>
        <userTransports>
          <transport_id>TCPv4_CLIENT</transport_id>
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
        ...
  </rtps>
</participant>

<participant profile_name="TCP server" >
  <rtps>
        ....
        <userTransports>
          <transport_id>TCPv4_SERVER</transport_id>
```

(continues on next page)

(continued from previous page)

```
        </userTransports>
        <useBuiltinTransports>false</useBuiltinTransports>
        ...
  </rtps>
</participant>
```

## 5.3 test_3_PDP_UDP.xml

Here we test the discovery capacity of handling late joiners. A single server is created, which manages the discovery information of four clients with different lifespans. The server prefix and listening ports are given in the profiles **UDP server** and **UDP client**. A snapshot is taken whenever there is a participant creation or removal to assess all entities are aware of it.

```

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

  <clients>
    <client name="client1" profile_name="UDP client" removal_time="8"/>
    <client name="client2" profile_name="UDP client" creation_time="2" removal_time=
↪"10" />
    <client name="client3" profile_name="UDP client" creation_time="4" removal_time=
↪"12" />
    <client name="client4" profile_name="UDP client" creation_time="6" removal_time=
↪"14" />
  </clients>

  <snapshots>
    <snapshot time="1">Check server and client1 known each other</snapshot>
      <snapshot time="3">Check server and client1 acknowledge client2 creation</
↪snapshot>
      <snapshot time="5">Check server, client1 and client2 acknowledge client3␣
↪creation</snapshot>
      <snapshot time="7">Check server, client1, client2 and client3 acknowledge␣
↪client4 creation</snapshot>
      <snapshot time="9">Check client1 removal is acknowledge by all remaining</
↪snapshot>
      <snapshot time="11">Check client2 removal is acknowledge by all remaining</
↪snapshot>
      <snapshot time="13">Check client3 removal is acknowledge by all remaining</
↪snapshot>
      <snapshot time="15">Check client4 removal is acknowledge by all remaining</
↪snapshot>
  </snapshots>

```

## 5.4 test_4_PDP_UDP.xml

Here we test the capability of one server to exchange information with another one. Two servers are created and each one has different associated clients. We take a snapshot to assess all clients are aware of the other server's clients existence. Note that we don't need to modify the previous tests profiles, as we can rely on the *server* and *client* tag attributes to avoid create redundant boilerplate profiles:

- *server* **prefix** attribute is used to supersede the profile specified one and uniquely identifies each server.
- *server* **ListeningPorts** and **ServerList** tags allow us to link servers between them without creating specific server profiles.
- *client* **server** attribute is used to link a client with its server without using a new profile or a **ServerList**.

```
<servers>
  <server name="server1" prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f" profile_name=
↪"UDP server" />
  <server name="server2" prefix="4D.49.47.55.45.4c.7e.42.41.52.52.4f" profile_name=
↪"UDP server">
    <ListeningPorts>
      <metatrafficUnicastLocatorList>
        <locator>
          <udpv4>
            <address>127.0.0.1</address>
            <port>31569</port>
          </udpv4>
        </locator>
      </metatrafficUnicastLocatorList>
    </ListeningPorts>
    <ServersList>
      <RServer prefix="4D.49.47.55.45.4c.5f.42.41.52.52.4f" />
    </ServersList>
  </server>
</servers>

<clients>
  <client name="client1" profile_name="UDP client" server="4D.49.47.55.45.4c.5f.42.
↪41.52.52.4f"/>
  <client name="client2" profile_name="UDP client" server="4D.49.47.55.45.4c.7e.42.
↪41.52.52.4f" />
  <client name="client3" profile_name="UDP client" server="4D.49.47.55.45.4c.7e.42.
↪41.52.52.4f" />
  <client name="client4" profile_name="UDP client" server="4D.49.47.55.45.4c.7e.42.
↪41.52.52.4f" />
</clients>

<snapshots>
  <snapshot time="3">Check all clients known each other</snapshot>
</snapshots>
```

## 5.5 test_5_EDP_UDP.xml & test_5_EDP_TCP.xml

These tests introduce dummy publishers and subscribers to assess proper EDP discovery operation over UDP and TCP transport. A server and two clients are created, and each participant (server included) creates publishers and subscribers with different types and topics. In the end, a snapshot is taken to verify all publishers and subscribers have been reported by all participants. Note that the tags *publisher* and *subscriber* have attributes to supersede topics specified in profiles.

```
<servers>
  <server name="server" profile_name="TCP server">
      <subscriber topic="topic 1" />
    <publisher topic="topic 2" />
  </server>
</servers>

<clients>
  <client name="client1" profile_name="TCP client" listening_port="56858">
      <subscriber /> <!-- defaults to helloworld type -->
      <subscriber topic="topic 2" />
      <subscriber profile_name="Sub 1" />
      <publisher profile_name="Pub 2" />
  </client>
  <client name="client2" profile_name="TCP client" listening_port="56859">
      <publisher />  <!-- defaults to helloworld type -->
    <publisher topic="topic 1" />
    <publisher profile_name="Pub 1" />
    <subscriber profile_name="Sub 2" />
  </client>
</clients>

<snapshots>
  <snapshot time="3">Check all publishers and subscribers are properly discovered␣
↪by everybody</snapshot>
</snapshots>
```

Snapshots with EDP information are far more verbose:

```
2019-04-24 14:52:44.300 [DISCOVERY_SERVER Info] Snapshot taken at 2019-04-24 14:52:44␣
↪description: Check all
publishers and subscribers are properly discovered by everybody
3 participants report the following discovery info:
Participant 1.f.1.30.64.47.0.0.2.0.0.0|0.0.1.c1 discovered:
         Participant 1.f.1.30.64.47.0.0.2.0.0.0|0.0.1.c1 has:
                1 publishers:
                        Publisher 1.f.1.30.64.47.0.0.2.0.0.0|0.0.1.3 TypeName: sample_
↪type_2 TopicName: topic_2
                3 subscribers:
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.2.4 TypeName:␣
↪HelloWorld TopicName: HelloWorldTopic
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.3.4 TypeName:␣
↪sample_type_2 TopicName: topic_2
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.4.4 TypeName:␣
↪sample_type_1 TopicName: topic_1

         Participant client2 1.f.1.30.64.47.0.0.3.0.0.0|0.0.1.c1 has:
                3 publishers:
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.1.3 TypeName:␣
↪HelloWorld TopicName: HelloWorldTopic
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.2.3 TypeName: sample_
↪type_1 TopicName: topic_1
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.3.3 TypeName: sample_
↪type_1 TopicName: topic_1
                1 subscribers:
```

(continues on next page)

(continued from previous page)

```
                        Subscriber 1.f.1.30.64.47.0.0.3.0.0.0|0.0.4.4 TypeName:␣
↪sample_type_2 TopicName: topic_2

         Participant server 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1 has:
                1 publishers:
                        Publisher 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.3␣
↪TypeName: sample_type_2 TopicName: topic_2
                1 subscribers:
                        Subscriber 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.2.4␣
↪TypeName: sample_type_1 TopicName: topic_1


Participant 1.f.1.30.64.47.0.0.3.0.0.0|0.0.1.c1 discovered:
         Participant client1 1.f.1.30.64.47.0.0.2.0.0.0|0.0.1.c1 has:
                1 publishers:
                        Publisher 1.f.1.30.64.47.0.0.2.0.0.0|0.0.1.3 TypeName: sample_
↪type_2 TopicName: topic_2
                3 subscribers:
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.2.4 TypeName:␣
↪HelloWorld TopicName: HelloWorldTopic
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.3.4 TypeName:␣
↪sample_type_2 TopicName: topic_2
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.4.4 TypeName:␣
↪sample_type_1 TopicName: topic_1

         Participant 1.f.1.30.64.47.0.0.3.0.0.0|0.0.1.c1 has:
                3 publishers:
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.1.3 TypeName:␣
↪HelloWorld TopicName: HelloWorldTopic
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.2.3 TypeName: sample_
↪type_1 TopicName: topic_1
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.3.3 TypeName: sample_
↪type_1 TopicName: topic_1
                1 subscribers:
                        Subscriber 1.f.1.30.64.47.0.0.3.0.0.0|0.0.4.4 TypeName:␣
↪sample_type_2 TopicName: topic_2

         Participant server 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1 has:
                1 publishers:
                        Publisher 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.3␣
↪TypeName: sample_type_2 TopicName: topic_2
                1 subscribers:
                        Subscriber 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.2.4␣
↪TypeName: sample_type_1 TopicName: topic_1


Participant 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1 discovered:
         Participant client1 1.f.1.30.64.47.0.0.2.0.0.0|0.0.1.c1 has:
                1 publishers:
                        Publisher 1.f.1.30.64.47.0.0.2.0.0.0|0.0.1.3 TypeName: sample_
↪type_2 TopicName: topic_2
                3 subscribers:
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.2.4 TypeName:␣
↪HelloWorld TopicName: HelloWorldTopic
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.3.4 TypeName:␣
↪sample_type_2 TopicName: topic_2
                        Subscriber 1.f.1.30.64.47.0.0.2.0.0.0|0.0.4.4 TypeName:␣
↪sample_type_1 TopicName: topic_1
```

(continues on next page)

(continued from previous page)

```
        Participant client2 1.f.1.30.64.47.0.0.3.0.0.0|0.0.1.c1 has:
                3 publishers:
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.1.3 TypeName:␣
↪HelloWorld TopicName: HelloWorldTopic
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.2.3 TypeName: sample_
↪type_1 TopicName: topic_1
                        Publisher 1.f.1.30.64.47.0.0.3.0.0.0|0.0.3.3 TypeName: sample_
↪type_1 TopicName: topic_1
                1 subscribers:
                        Subscriber 1.f.1.30.64.47.0.0.3.0.0.0|0.0.4.4 TypeName:␣
↪sample_type_2 TopicName: topic_2

        Participant 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.c1 has:
                1 publishers:
                        Publisher 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.1.3␣
↪TypeName: sample_type_2 TopicName: topic_2
                1 subscribers:
                        Subscriber 4d.49.47.55.45.4c.5f.42.41.52.52.4f|0.0.2.4␣
↪TypeName: sample_type_1 TopicName: topic_1
```

## 5.6 test_6_EDP_UDP.xml

Here we test how the discovery handles EDP late joiners. It's the same scenario with a server and two clients with different lifespans. Each participant (server included) creates publishers and subscribers with different lifespans, types, and topics. Snapshots are taken whenever an endpoint is created or destroyed to assess every participant shares the same discovery info.

```
<servers>
  <server name="server" profile_name="UDP server" creation_time="1" >
    <subscriber topic="topic 1" creation_time="3" />
    <publisher topic="topic 2" creation_time="5" />
  </server>
</servers>

<clients>
  <client name="client1" profile_name="UDP client" removal_time="16">
    <subscriber creation_time="7" removal_time="14" /> <!-- defaults to helloworld␣
↪type -->
    <subscriber topic="topic 2" creation_time="7" removal_time="14" />
    <subscriber profile_name="Sub 1" creation_time="7" removal_time="14" />
    <publisher profile_name="Pub 2" creation_time="12" />
  </client>
<client name="client2" profile_name="UDP client" creation_time="9">
    <publisher creation_time="10" />  <!-- defaults to helloworld type -->
    <publisher topic="topic 1" creation_time="10" />
    <publisher profile_name="Pub 1" creation_time="10" />
    <subscriber profile_name="Sub 2" creation_time="10" />
  </client>
</clients>

<snapshots>
    <snapshot time="2">Check client1 detects the server</snapshot>
```

(continues on next page)

(continued from previous page)

```
    <snapshot time="4">Check client1 detects server's subscriber</snapshot>
    <snapshot time="6">Check client1 detects server's publisher</snapshot>
    <snapshot time="8">Check server detects client1's subscribers</snapshot>
    <snapshot time="11">Check server and client1 detects client2's and its entities
↪</snapshot>
    <snapshot time="13">Check everybody detects new client1's publisher</snapshot>
    <snapshot time="15">Check everybody detects client1's subscribers' removal</
↪snapshot>
    <snapshot time="17" someone="false">Check server and client2 detects client1␣
↪removal</snapshot>
  </snapshots>
```

## 5.7 test_7_PDP_UDP.xml

Here we test how the discovery handles server shutdown and reboot. This is a clean shutdown made from the Fast RTPS API `Domain::removeParticipant`. Each time the server dies it notifies this fact to all its clients which automatically begin pinging on the server again trying to reconnect when its rebooted. Snapshots check that clients are aware of server absence after shutdown and presence after reboot.

```

  <servers>
    <server name="server" profile_name="UDP server" removal_time="2" />
    <server name="server" profile_name="UDP server" creation_time="4" removal_time="6
↪" />
    <server name="server" profile_name="UDP server" creation_time="8" removal_time="10
↪" />
  </servers>

  <clients>
    <client name="client1" profile_name="UDP client" />
    <client name="client2" profile_name="UDP client" />
    <client name="client3" profile_name="UDP client" />
    <client name="client4" profile_name="UDP client" />
  </clients>

  <snapshots>
    <snapshot time="1">Check server-clients awareness</snapshot>
    <snapshot time="3">Check server demise has been reported to all clients</snapshot>
    <snapshot time="5">Check server-clients awareness has been recovered</snapshot>
    <snapshot time="7">Check server demise has been reported to all clients</snapshot>
    <snapshot time="9">Check server-clients awareness has been recovered</snapshot>
    <snapshot time="11">Check server demise has been reported to all clients</
↪snapshot>
  </snapshots>
```

## 5.8 test_8_lease_client.xml & test_8_lease_server.xml

Standard lease duration mechanism no longer makes sense on the client-server architecture. Clients no longer multicast `DATA(p)` messages in order to make all other clients aware of its presence as in PDP standard mechanism, thus, this periodical messages can no longer be used to assert participant liveliness. In the client-server architecture:

- clients only track its server liveliness by sending periodical messages to them. If a server dies because of lease-duration its client must resume pinging on it in order to reconnect.
- servers track clients and linked servers liveliness by sending periodical messages to them. If a client dies the server must propagate a `DATA(p[UD])` for that client over its PDP network. This way all server's clients have a shared lease duration capability.

In order to test this a python script is used to launch two discovery-servers instances:

- A server with several clients. These instances will take a snapshot at the beginning and another at the end.
- A client who references the server on the first instance. This process would be killed from python between the snapshots.

The first snapshot must show how all clients (remote one included) known each other. After killing process 2 (and its client) the server must kill its proxy by lease duration time out and report it to all other clients. The second snapshot must show how all participants have removed the remote client from its discovery database.

# CHAPTER 6

# Version 1.0.0

First release.

- server creation and setup capabilities.
- discovery testing capabilities.
- single process suite of tests added.
- multiprocess-machine test capability added (generation and validation of snapshots serialized as XML).
- Hello World example over UDP and TCP.
- extended schema that simplifies setup for server creation and testing purposes.