RTI Cloud Discovery Service

User's Manual

Version 7.2.0

Contents

Welcome to *RTI® Cloud Discovery Service* an out-of-the-box solution for provisioning discovery in cloud-based environments.

Chapter 1

Table of Contents

1.1 Introduction

RTI® Cloud Discovery Service is a stand-alone application that deploys *RTI Connext®* applications in dynamic environments where UDP/IP multicast is not available. This is typical of wide-area networks or some cloud-based environments where the routers and switches may disable IP multicast forwarding. *Cloud Discovery Service* also works in conjunction with the *RTI Real-Time WAN Transport* to provide peer-to-peer communication between *DomainParticipants* situated behind Network Address Translators (NATs).

DDS has a builtin Discovery Service that allows all DDS applications to automatically detect the presence of other applications and discover the Topics they publish and subscribe along with the associated data types and *Quality of Service* (QoS).

The builtin discovery service primarily relies on UDP/IP multicast to bootstrap the detection of other DDS applications and learn their network addresses. The use of UDP/IP multicast allows DDS discovery to be completely peer-to-peer. That is, it can operate without requiring any additional services or brokers. The applications themselves can discover each other directly.

However, if *Connext* applications run in environments where UDP/IP multicast is not available, then builtin (peer-to-peer) discovery is not sufficient. *Connext* offers two mechanisms to help with those scenarios:

- For static environments where the network addresses of all the applications are known a-priori, you can configure your application to automatically check on these addresses for the presence of other applications. This is accomplished by configuring the Initial Peers.
- For dynamic environments where the network addresses are not known in advance, or in cases where the list is too large or cumbersome to manage, you can leverage *RTI Cloud Discovery Service*. This external service acts as a reliable "rendezvous" point for *Connext* applications to learn about the presence and network addresses of other DDS applications. You can do this by setting the Initial Peers to include *Cloud Discovery Service*.
- For WAN deployments where applications require Network Address Translator (NAT) traversal. *Cloud Discovery Service* can aid in this process in combination with *RTI Real-Time WAN Transport*. See RTI Real-Time WAN Transport.

Figure 1.1 shows a simple representation of the operation of *Cloud Discovery Service*. It acts as a "relay" service that forwards the bootstrap (participant announcement) messages that allow *DomainParticipants* to learn about

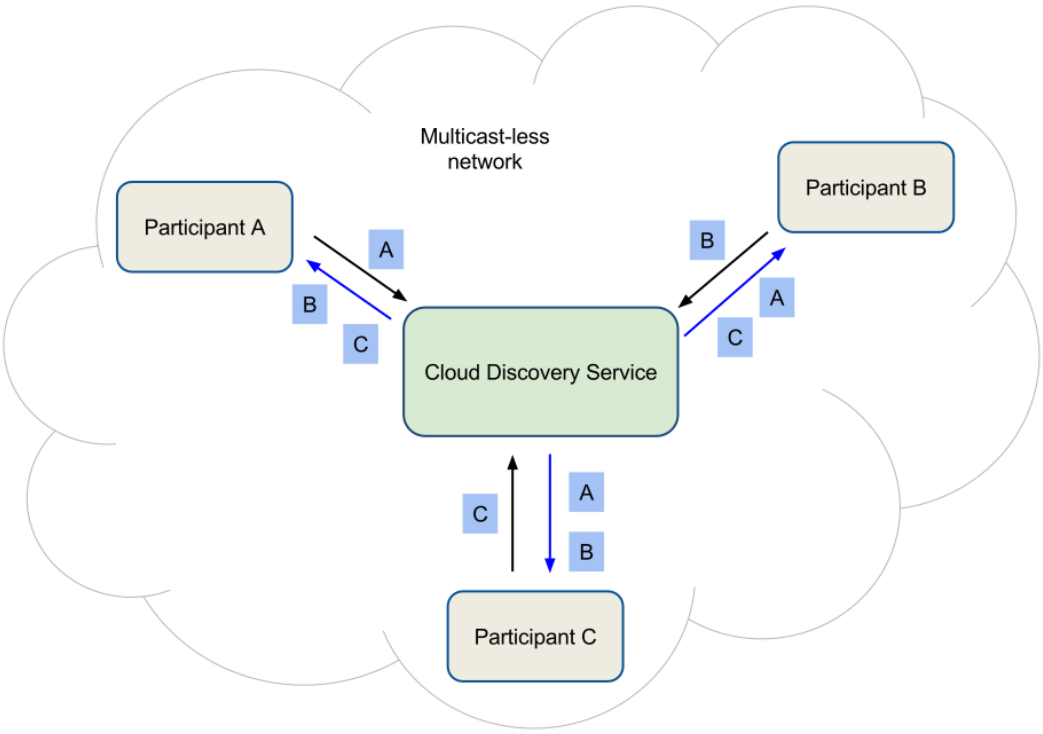

Figure 1.1: Cloud Discovery Service Overview

the presence of other *DomainParticipants*.

1.1.1 The Basics

Cloud Discovery Service operates using the default *Simple Participant Discovery Protocol* (SPDP)[1] or the new *Simple Participant Discovery Protocol 2.0* (SPDP2)[2]. In either case, *DomainParticipants* initially announce their presence to all the specified Initial Peers. These *participant announcements* contain the necessary information for other *DomainParticipants* to discover their presence and bootstrap communications. In *Simple Participant Discovery Protocol*, these messages are also used to maintain participant liveliness.

Cloud Discovery Service listens for participant announcements to dynamically learn about the current list of *DomainParticipants*, their DDS domain IDs, and their network addresses.

Figure 1.2 illustrates that each *DomainParticipant* includes a *Cloud Discovery Service* instance in its Initial Peers. Hence, the *DomainParticipant* will send *participant announcements* to *Cloud Discovery Service*, which will forward those announcements to the list of *DomainParticipants* it knows about, enabling them to initiate the discovery process among themselves.

Figure 1.3 illustrates that once a *DomainParticipant* discovers the presence of another one (via the forwarded message from *Cloud Discovery Service*) it sends its Participant Announcement messages directly. This step is

[1] *Simple Participant Discovery Protocol* discovers all other *DomainParticipants* in the same *DDS Domain* by sending participant announcements. Such announcements include the *DomainParticipant* unique identifying key, transport locators, and QoS. These announcements are sent on a periodic basis using best-effort communication.

[2] *Simple Participant Discovery Protocol 2.0* is an alternative to the original *Simple Participant Discovery Protocol*. SPDP2 is designed to decrease bandwidth usage and improve the reliability of the participant discovery and update process.

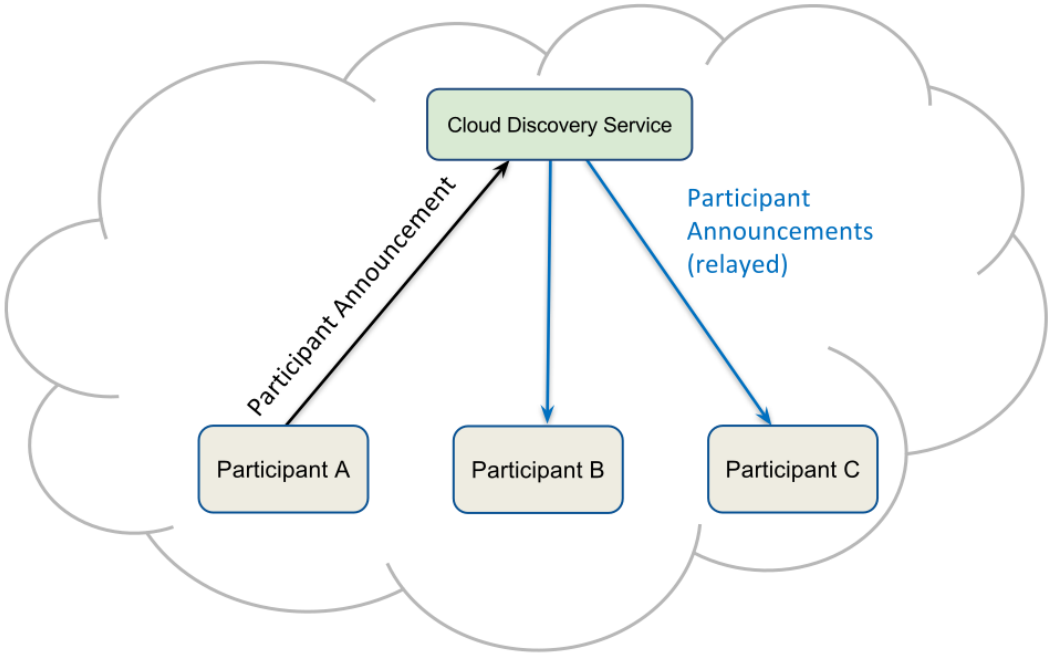

Figure 1.2: Cloud Discovery Service forwards Participant Announcement messages

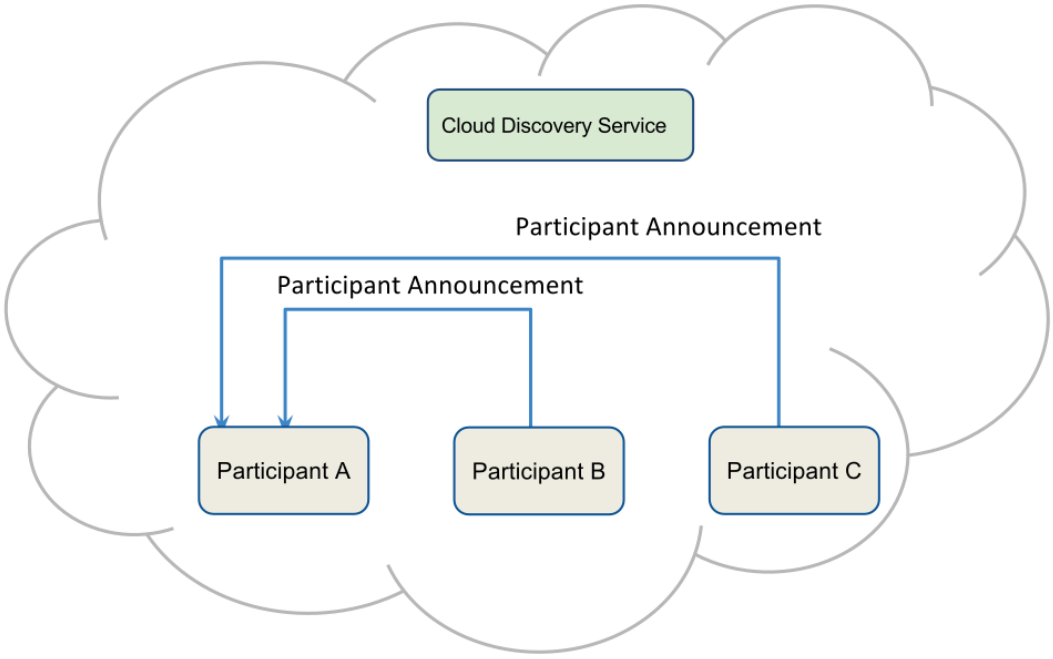

Figure 1.3: *DomainParticipants* exchange Participant Announcement messages

the same as if the *DomainParticipant* had included the other *DomainParticipant* in its Initial Peers or was using multicast to announce its presence. These messages are also used to maintain participant liveliness.

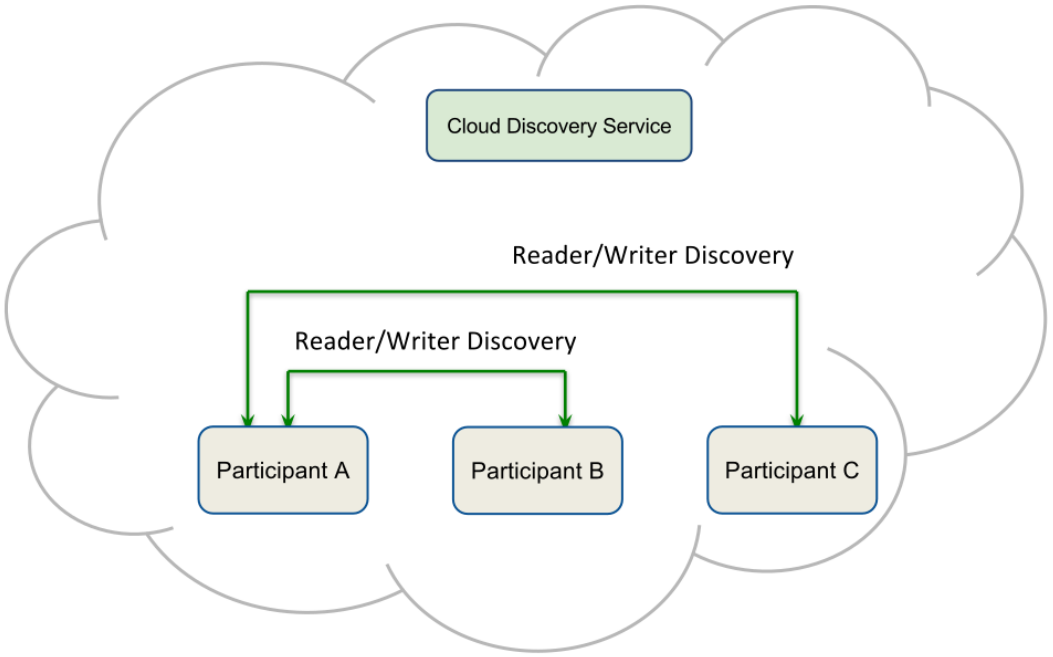

Figure 1.4: *DomainParticipants* exchange Endpoint Discovery messages

Figure 1.4 illustrates that once *DomainParticipants* know about each other, they exchange Endpoint discovery information. That is, they exchange information about the DataWriters and DataReaders each one has. This step is unaltered by the presence of *Cloud Discovery Service*.

Direct benefits of *Cloud Discovery Service*:

- Dynamic discovery remains possible even if multicast is not available, without the need to anticipate or maintain a list of peers.
- *Cloud Discovery Service* integrates seamlessly in a DDS environment. Because it operates at the RTPS[3] level, *Connext* applications do not need any special behavior or protocol. They just need to configure their Initial Peers to contain the location where *Cloud Discovery Service* is running. Since this is all configurable at runtime, there is no need to recompile your application. This implies that *Cloud Discovery Service* will also work with existing services such as *RTI Routing Service*, *RTI Connext Micro*, or other implementations of DDS-RTPS.
- Discovery remains a distributed process performed among all the *DomainParticipants*. This allows you to scale the system dynamically. It also avoids having centralized and bottleneck servers.

For a deeper understanding of discovery, refer to Discovery Overview in the *RTI Connext Core Libraries User's Manual*.

[3] Real-Time Publish-Subscribe Protocol

1.1.2 Available Documentation

In this manual you can find:

- *Introduction*
- *Installation*
- *Core Concepts*
- *Configuration*
- *Usage*
- *Tutorials*
- *Troubleshooting*
- *Release Notes*

1.1.3 Paths Mentioned in Documentation

This documentation refers to:

- `<NDDSHOME>` This refers to the installation directory for *Connext*. The default installation paths are:
  - macOS® systems: `/Applications/rti_connext_dds-version`
  - Linux® systems, non-root user: `/home/your user name/rti_connext_dds-version`
  - Linux systems, root user: `/opt/rti_connext_dds-version`
  - Windows® systems, user without Administrator privileges: `<your home directory>\rti_connext_dds-version`
  - Windows systems, user with Administrator privileges: `C:\Program Files\rti_connext_dds-version`

  You may also see `$NDDSHOME` or `%NDDSHOME%`, which refers to an environment variable set to the installation path.

  Whenever you see `<NDDSHOME>` used in a path, replace it with your installation path.

  Note for Windows Users: When using a command prompt to enter a command that includes the path `C:\Program Files` (or any directory name that has a space), enclose the path in quotation marks. For example: `"C:\Program Files\rti_connext_dds-version\bin\rticlouddiscoveryservice.bat"`

  Or if you have defined the `NDDSHOME` environment variable: `"%NDDSHOME%\bin\rticlouddiscoveryservice.bat"`

- `<path to examples>` By default, examples are copied into your home directory the first time you run *RTI Launcher* or any script in `<NDDSHOME>/bin`. This document refers to the location of the copied examples as `<path to examples>`.

Wherever you see <path to examples>, replace it with the appropriate path. Default path to the examples:

  - macOS systems: /Users/your user name/rti_workspace/version/examples
  - Linux systems: /home/your user name/rti_workspace/version/examples
  - Windows systems: your Windows documents folder\rti_workspace\version\examples. Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 7, the folder is C:\Users\your user name\Documents; on Windows Server 2003, the folder is C:\Documents and Settings\your user name\Documents.

- <RTIMEHOME> This environment variable must be set to the installation directory path for *Connext Micro*. If you installed *Connext Professional* with default settings, this will be in: <path_to_connext_dds_installation>/rti_connext_dds-version/rti_connext_micro-version. If you have copied *Connext Micro* to another place, set RTIMEHOME to point to that location.

References

1.2 Installation

Cloud Discovery Service is not installed as part of a *Connext* package, unless you are installing an evaluation ("eval") or LM ("lm") package (such as rti_connext_dds-|CONNEXT_currentVersion|-eval-x64Win64VS2017.exe or rti_connext_dds-|CONNEXT_currentVersion|-lm-x64Win64VS2017.exe), which includes *Cloud Discovery Service*. Unless you are installing an "eval" or "lm" package, *Cloud Discovery Service* must be downloaded and installed separately. Contact support@rti.com for information on how to obtain a *Cloud Discovery Service* package for your platform.

1.2.1 Installing an Evaluation or LM Version

Install the evaluation ("eval") or LM ("lm") package as described in the RTI Connext Installation Guide. The package includes *Cloud Discovery Service*.

After installing the **eval** or **lm** package, see *How to use a License File with RTI Services*.

1.2.2 Installing a Regular Version

There are two ways to install *Cloud Discovery Service*: using *RTI Launcher* or the rtipkginstall command-line utility. For each of the methods, make sure that you install the correct host and target for your environment (see *Cloud Discovery Service now shipped as host and target*). Additional information about installation can be found at Installing Connext, in the Getting Started Guide.

Note: *Cloud Discovery Service* is supported with *Connext* 5.3.0 and higher. Before 6.1.0, *Cloud Discovery Service* was an RTI Labs product that came installed with *Connext*. As of 6.1.0, it must be downloaded and

installed separately (unless you have an lm package).

Installing from RTI Launcher

To install *Cloud Discovery Service* from *RTI Launcher*:

1. Start *RTI Launcher* from the Start menu or from the command line, by running: `<NDDSHOME>/bin/rtilauncher`.
2. From *Launcher's* **Configuration** tab, click on *Install RTI Packages*.
3. Use the **+** sign to add the `.rtipkg` file(s) that you want to install.
4. Click **Install**.

Installing from the command line

To install *Cloud Discovery Service* from the command line, run:

```
$NDDSHOME/bin/rtipkginstall \
    /<path-to-cloud-discovery-service>/rtipkgfile.rtipkg
```

Other dependencies

Cloud Discovery Service may have dependencies on other packages as follows:

- RTI TLS support and OpenSSL: If using TLS as a transport.
- RTI Security and OpenSSL: If using the *Security* provisions provided by *Cloud Discovery Service*.

1.3 Core Concepts

This section aims to provide a deeper understanding of what *Cloud Discovery Service* is made of, to give you the required insight to configure and use it effectively.

You will learn about:

- *Domain lists*: Sets the context for *Cloud Discovery Service* in DDS domains.
- *Transport*: Shows how the transports are modeled in *Cloud Discovery Service*, which is the basis for communication with *DomainParticipants*.
- *Forwarder*: Explains the active logic of *Cloud Discovery Service* that is in charge of processing the incoming participant announcements, building the system discovery state, and providing the information to all *DomainParticipants*.
- *Database*: Describes the collection of discovered *DomainParticipant* information that *Cloud Discovery Service* maintains to represent the current state of the system.

1.3.1 Domain Lists

A single *Cloud Discovery Service* can forward participant announcements in *multiple DDS domains*. Yet *Cloud Discovery Service* preserves the isolation of *DDS domain IDs* such that a *DomainParticipant* only discovers other *DomainParticipants* with the same *DDS domain ID*.

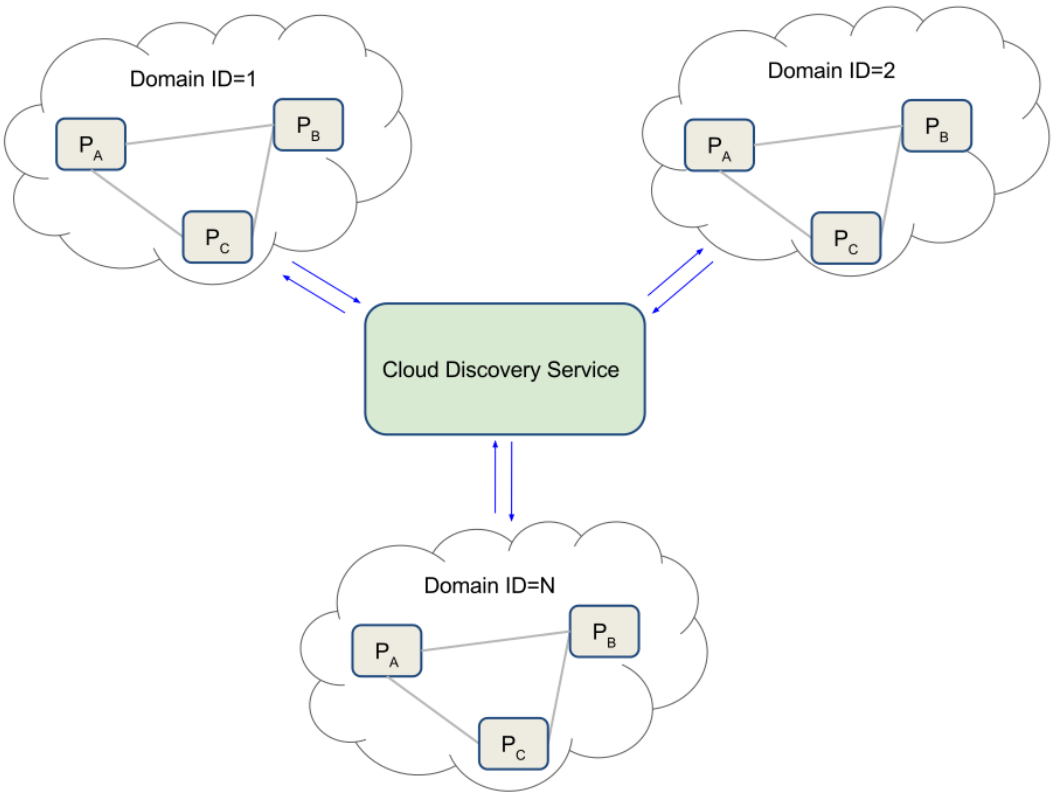

Figure 1.5: Cloud Discovery Service works with Multiple Domains

By providing a *domain list*, you can control on which domains *Cloud Discovery Service* operates. A domain list is a representation of domains by their IDs. You can learn more about configuring domain lists in *Domain List*.

Support for multiple domain IDs can simplify system deployments, allowing a few *Cloud Discovery Service* instances to provide discovery services for all possible expected domain IDs, even if all these domain IDs are not known in advance.

Domain Tags

In large-scale systems or multi-tenant networks, such as those found in *Cloud* deployments, the isolation provided by the DDS domain ID may not be sufficient:

- The maximum number of domain IDs is limited and not sufficiently large. In deployments that require many independent DDS systems in a common network, the domain ID would not be sufficient to provide isolation.
- Managing the assignment of numeric domain IDs to independent systems or projects can be cumbersome. Numeric IDs cannot easily leverage existing organizational or project boundaries such as project name,

department, organization, etc.

For these reasons, the DDS domain concept has been extended to include *Domain Tags*.

A domain tag is a logical sub-division within a domain. It is defined as a string. *DomainParticipants* associated to different domain tags will not discover each other even if they are in the same *domain ID*. You can think of a *Domain Tag* as the DDS equivalent of a network VLAN.

Figure 1.6 shows an overview of the domain tag concept.

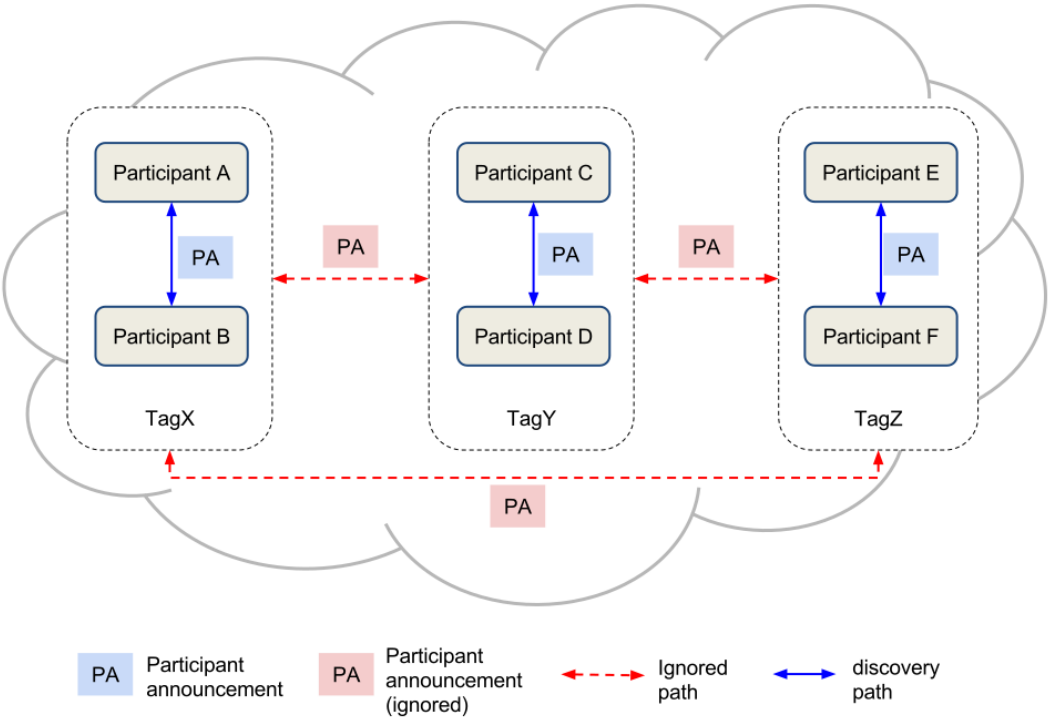

Figure 1.6: Domain Tags

A *DomainParticipant* can be associated with only one domain tag. In a given domain, all the *DomainParticipants* may exchange participant announcements in order to initiate the discovery process between them. However, only *DomainParticipants* that have **the same domain tag** will discover each other.

Domain tags allow you to divide your domain into as many logically isolated spaces as you need. A domain tag is represented by a string *tag name*. You can define as many tags as you need. The tag name may include any ASCII character.

Note: The absence of a domain tag is treated like a special value of the tag. *DomainParticipants* that do not specify a domain tag will communicate only with each other.

Characteristics of domain tags:

1. **Single Tag per DomainParticipant**: A *DomainParticipant* can be associated with a single domain tag.
2. **Immutability**: Domain tags are immutable. *DomainParticipants* specify the domain tag at creation time and it cannot be modified.

Cloud Discovery Service supports domain tags and forwards discovery information only to *DomainParticipants* with matching tuple of *(domainID, domainTag)*. Figure 1.7 shows the domain tags' effect when *Cloud Discovery Service* drives the discovery process.

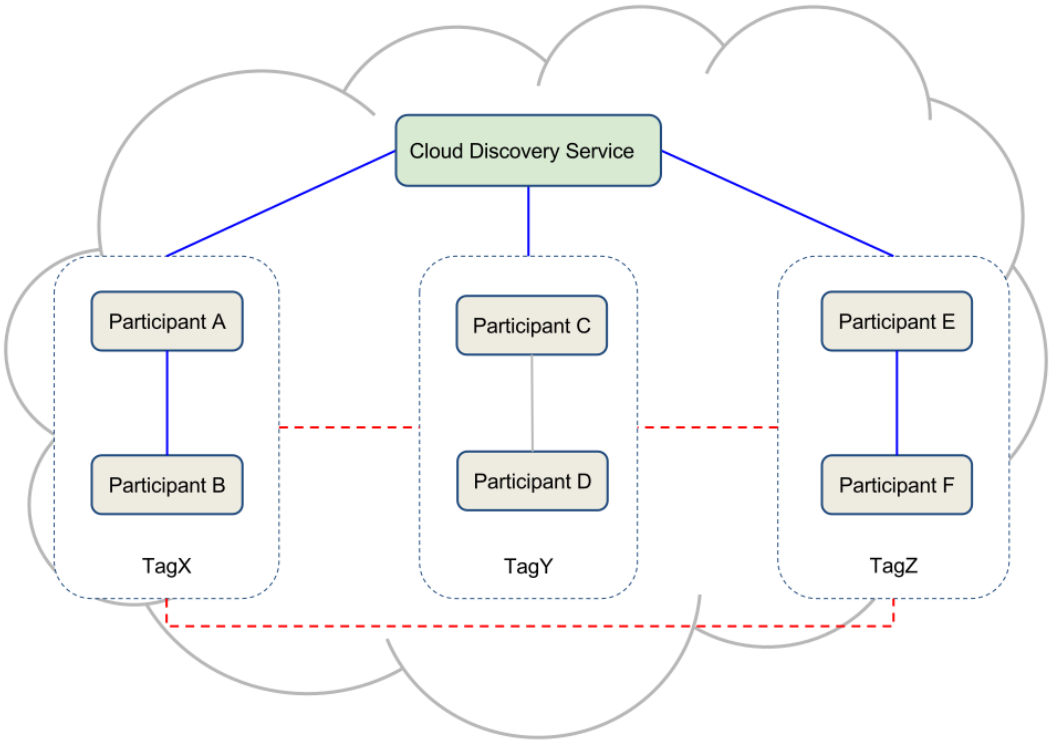

Figure 1.7: *Cloud Discovery Service* with Domain Tags

Cloud Discovery Service detects the domain tag of each *DomainParticipant* and remembers that association. That way it can forward the participant announcements only to those *DomainParticipants* that belong to the same domain ID and tag.

You can specify the domain tag for a *DomainParticipant* at creation time via *DomainParticipantQos*. In particular, you need to propagate the well-known property `dds.domain_participant.domain_tag`, whose value contains the name of the tag associated with the *DomainParticipant*. You can specify this property through the *PropertyQosPolicy*. For more information, see Choosing a Domain Tag in *RTI Connext Core Libraries User's Manual*.

Specifying the Domain Tag in XML

The following XML snippet shows how to specify the domain tag within `<domain_participant_qos>`.

```
<domain_participant_qos>
    <property>
        <element>
            <name>dds.domain_participant.domain_tag</name>
            <value>TagX</value>
            <propagate>true</propagate>
```

(continues on next page)

(continued from previous page)

```
        </element>
    </property>
</domain_participant_qos>
```

Domain Participant Partitions

Domain Participant Partitions provide a way to create different communication planes within a *(domain ID, domain Tag)*. *DomainParticipants* can join and leave these communication planes at any time while they are running. Partitioning at the *DomainParticipant* level can be particularly useful in large, WAN, distributed systems (with thousands of *DomainParticipants*) in which not all *DomainParticipants* need to know about each other at any given time. Partitioning at the *DomainParticipant* level helps reduce network, CPU, and memory utilization, because *DomainParticipants* without matching partitions will not exchange information about their *DataWriters* and *DataReaders*.

To summarize, the characteristics of *Domain Participant Partitions* are:

1. **Mutable**: A *DomainParticipant* can change its partition values at runtime. This allows flexibility within a running system.
2. **Multiple Values**: A *DomainParticipant* can contain multiple values including wildcard characters for its partitions. This allows a *DomainParticipant* to be a member of multiple partitions simultaneously.
3. **Efficient**: They are more efficient than *Endpoint Partitions*, since traffic is restricted only until the *Participant Discovery* level. No *Endpoint Discovery* takes place if partition values for two *DomainParticipants* don't match.

Cloud Discovery Service supports systems that utilize *Domain Participant Partitions*. It forwards announcements to *DomainParticipants* with matching tuple of *(domainID, domainTag)* **and** matching *Domain Participant Partitions*. In this way, *Cloud Discovery Service* increases bandwidth reduction even further, because an announcement from a *DomainParticipant* is only forwarded to *DomainParticipants* with a matching partition. (*Cloud Discovery Service* accomplishes this targeted forwarding by maintaining an internal state to help with the filtering related to partitions.)

Figure 1.8 shows the *Domain Participant Partitions*' effect when *Cloud Discovery Service* drives the discovery process.

Cloud Discovery Service detects the *Domain Participant Partitions* of each *DomainParticipant* and maintains a mapping for its matching *DomainParticipants*. That way it can forward the participant announcements only to those *DomainParticipants* that pass the matching criteria highlighted above. When a *DomainParticipant* changes its *Domain Participant Partitions* value, *Cloud Discovery Service* updates the mapping for the new value before forwarding the incoming participant announcement.

You can specify the *Domain Participant Partitions* for a *DomainParticipant* at creation time via *DomainParticipantQos* in its *Partition QoS Policy*. For XML, you need to set the `<partition>` tag, whose value contains the list of partitions associated with the *DomainParticipant*. You can also change the value for *Domain Participant Partitions* at runtime while the *DomainParticipant* is running. For more information, see Domain Participant Partitions in *RTI Connext Core Libraries User's Manual*.

Note: All *DomainParticipants* are members of at least one concrete partition. If a *DomainParticipant* does

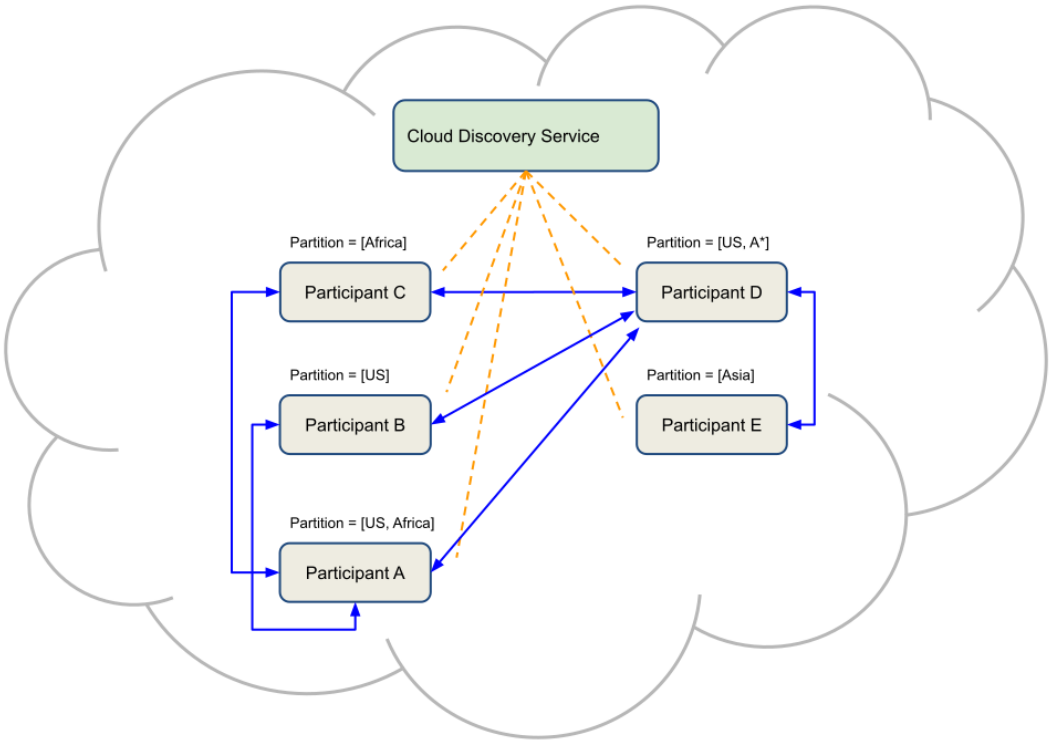

Figure 1.8: *Cloud Discovery Service* with Domain Participant Partitions

not specify any partition or if it only specifies wildcard partitions, the *DomainParticipant* is considered a member of the empty partition. The absence of a partition tag is treated like a special value (empty partition). *DomainParticipants* that do not specify a partition tag will communicate only with each other.

Specifying Domain Participant Partitions in XML

The following XML snippet shows how to specify the partition values within `<domain_participant_qos>`.

```
<domain_participant_qos>
    <partition>
        <name>
            <element>US</element>
            <element>A*</element>
        </name>
    </partition>
</domain_participant_qos>
```

Changing Domain Participant Partitions in code

The following code snippet shows how to change the *Domain Participant Partitions* value at runtime via code.

C

```
DDS_DomainParticipant *participant = NULL;
struct DDS_DomainParticipantQos participantQos =
        DDS_DomainParticipantQos_INITIALIZER;

/* QoS configuration happens here */

/* Create the participant */
participant = DDS_DomainParticipantFactory_create_participant(
        DDS_TheParticipantFactory,
        domainId,
        &participantQos,
        NULL,
        DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    /* Handle error */
}

/* Participant is now up and running */

/* Get the QoS */
if (DDS_DomainParticipant_get_qos(participant, &participantQos))
        != DDS_RETCODE_OK) {
    /* Handle error */
}

/* Set the sequence length and values */
if (!DDS_StringSeq_ensure_length(
        &participantQos.partition.name,
        3,
        3)) {
    /* Handle error */
}

*DDS_StringSeq_get_reference(&participantQos.partition.name, 0) =
        DDS_String_dup("US");
if (*DDS_StringSeq_get_reference(&participantQos.partition.name, 0)
        == NULL) {
    /* Handle error */
}

*DDS_StringSeq_get_reference(&participantQos.partition.name, 1) =
        DDS_String_dup("Africa");
if (*DDS_StringSeq_get_reference(&participantQos.partition.name, 1)
        == NULL) {
    /* Handle error */
}
```

(continues on next page)

(continued from previous page)

```
*DDS_StringSeq_get_reference(&participantQos.partition.name, 2) =
        DDS_String_dup("Asia");
if (*DDS_StringSeq_get_reference(&participantQos.partition.name, 2)
        == NULL) {
    /* Handle error */
}

/* Assign the new QoS */
if (DDS_DomainParticipant_set_qos(participant, &participantQos))
        != DDS_RETCODE_OK) {
    /* Handle error */
}
```

C++

```
using namespace dds::domain;
using namespace dds::domain::qos;
using namespace dds::core::policy;
using namespace dds::core;

DomainParticipantQos qos = DomainParticipant::default_participant_qos();

// QoS configuration happens here

// Create the participant
DomainParticipant participant(domain_id(), qos);

// Participant is now up and running

// Create a new partition value
Partition newPartitionValues(StringSeq({ "US", "Africa", "Asia" }));

// Update the QoS
qos << newPartitionValues;

// Assign the new QoS
participant << qos;
```

1.3.2 Transport

Cloud Discovery Service allows you to select which transports to use to send and receive discovery traffic. The selection of transports defines the possible locations where the service can be reached. *DomainParticipants* require this information to communicate with the service.

In a *Connext* application, *DomainParticipants* automatically configure the underlying transport ports based on the domain ID and the network capabilities of the host machine where the application runs. In *Cloud Discovery Service*, this configuration is manual and explicit.

Note: *Cloud Discovery Service* only works with *unicast* transports. In this manual, any reference to the

transport will always imply unicast.

In particular, *Cloud Discovery Service* allows you to choose:

- **Transport class instance**: A *transport class* is a concrete realization of a networking transport (e.g., UDP, TCP, etc.). You can instantiate this class and uniquely identify it with a *transport alias*. In addition, for each instantiation you can configure properties specific to the implementation (e.g., network interfaces, receive buffer sizes, etc.).
- **Transport Address**: A *transport address* represents an interface for a specific transport class (for example, an IP address for the UDP or TCP transport).
- **Receive Port**: Identifies where the service listens for incoming data. For more information about ports, see *About Ports*.

Each transport instance-address tuple constitutes a Transport *Locator*, which is a unique data-reception endpoint. *Cloud Discovery Service* creates send resources to put discovery traffic on the wire. *Connext* uses ephemeral ports for outbound data, hence a send resource is specified by a transport instance only. *Cloud Discovery Service* creates a send resource for each transport instance specified.

Cloud Discovery Service relies on the same pluggable transport framework that is available for *Connext* applications to create and access transport resources. This implies you can use not only *RTI Connext* builtin transports, but also your own transport implementations. For more information, see transport plugins in *RTI Connext Core Libraries User's Manual*.

DomainParticipants can communicate with *Cloud Discovery Service* as long as they are configured with the proper transport settings. Then they can reach a specific service instance in two ways:

1. By addressing the service instance through an *RTPS peer descriptor* which allows you to identify a DDS service in a generic way by its *locator*. *DomainParticipants* can include these in their lists of initial peers.
2. By addressing the service instance through a peer participant descriptor such that the resulting destination port, computed from the domain ID of the *DomainParticipant* and well-known ports configuration, matches the listening port of the service.

Cloud Discovery Service determines how to reach *DomainParticipants* based on the locator information they propagate as part of their participant announcements. *Cloud Discovery Service* sends discovery traffic to a *DomainParticipant* by sending data to each of its locators.

RTPS Peer Descriptor

A peer descriptor is a string representation of a set of locators for DDS *DomainParticipants*. It provides a compact way to indicate a list of locators where a *DomainParticipant* can find other *DomainParticipants*. This is known as a **participant peer descriptor**, or simply a **peer descriptor**.

RTI Connext applications use the *peer descriptor* to bootstrap the participant discovery process with other *DomainParticipants*. Refer to discovery peer configuration in the *RTI Connext Core Libraries User's Manual*.

The **RTPS peer descriptor** is another kind of peer descriptor that allows addressing a service with which you communicate through the *RTPS* protocol, and that does not necessarily imply the existence of a *DomainParticipant*. *Cloud Discovery Service* is an example of such a service.

Note: The *RTPS peer descriptor* format is supported only by applications using *RTI DDS Connext* versions 5.3.0 and higher.

The RTPS peer descriptor format is shown below:

Figure 1.9: RTPS Peer Descriptor Format

Table 1.1 describes all the elements in the RTPS peer descriptor.

Table 1.1: RTPS Peer Descriptor Elements

Element	Description	Required	Default
`rtps`	Keyword to indicate the RTPS descriptor kind.	Yes	
`@`	Separator.	Only when `<locator>` is specified.	
`<locator>`	Specifies a transport and an address. See locator format.	No	udpv4://localhost
`:`	Separator.	Only when `<port>` is specified.	
`<port>`	RTPS Peer receive port. See *About Ports*.	No	7400

Example: RTPS Peer Descriptors

Table 1.2: RTPS Peer Descriptor for **UDP/IP** Version 4 Transport

`rtps`	`@`	`udpv4://192.169.1.1`	`:`	`7400`

Table 1.3: RTPS Peer Descriptor for a Generic **Starfabric** Transport

`rtps`	`@`	`starfabric://FA::0#0/0/R`		

Example: Transport Setup and Resulting RTPS Descriptor

Assume *Cloud Discovery Service* is configured with the following transport settings and running on host with address `CDS-Host-IP-Address`:

```
<transport>
    <element>
        <alias>udpv4</alias>
        <receive_port>57410</receive_port>
    </element>
</transport>
```

DomainParticipants can send discovery traffic to this service by adding the following peer to their initial peer lists:

```
rtps@udpv4://<CDS-Host-IP-Address>:57410
```

About Ports

Discovery traffic through RTPS relies on well-known ports to establish communication between peers. These ports are a logical concept that allow multiplexing communications at the middleware layer.

It is up to the underlying transport implementation to decide how to map these logical port numbers into the physical transport address scheme. For instance, the RTI builtin UDP transport directly maps the logical port number into the physical UDP port.

The port mapping is especially important in *Cloud* environments where UDP and TCP dominate the transport layer. The presence of these protocols often requires knowing details about the communication ports to properly set up application services.

In a *DomainParticipant*, unless explicitly configured, ports are determined automatically based on the domain ID and participant ID. Also, the port mapping is fully controlled by the underlying transport. In *Cloud Discovery Service*, the ports must be configured explicitly. Moreover, for the direct known RTI transport implementations that rely on UDP and TCP, *Cloud Discovery Service* maps the (logical) receive port directly to the transport physical port.

See RTPS Ports Used for Communication to learn more about RTPS ports.

1.3.3 Forwarder

The forwarder is the *Cloud Discovery Service* component where all the discovery logic resides. Its responsibility is to build/maintain the discovery state and forward participant announcements to the peer *DomainParticipants* so that they can discover each other.

Figure 1.10 shows a representation of the forwarder element of *Cloud Discovery Service*.

The forwarder is composed of four main blocks:

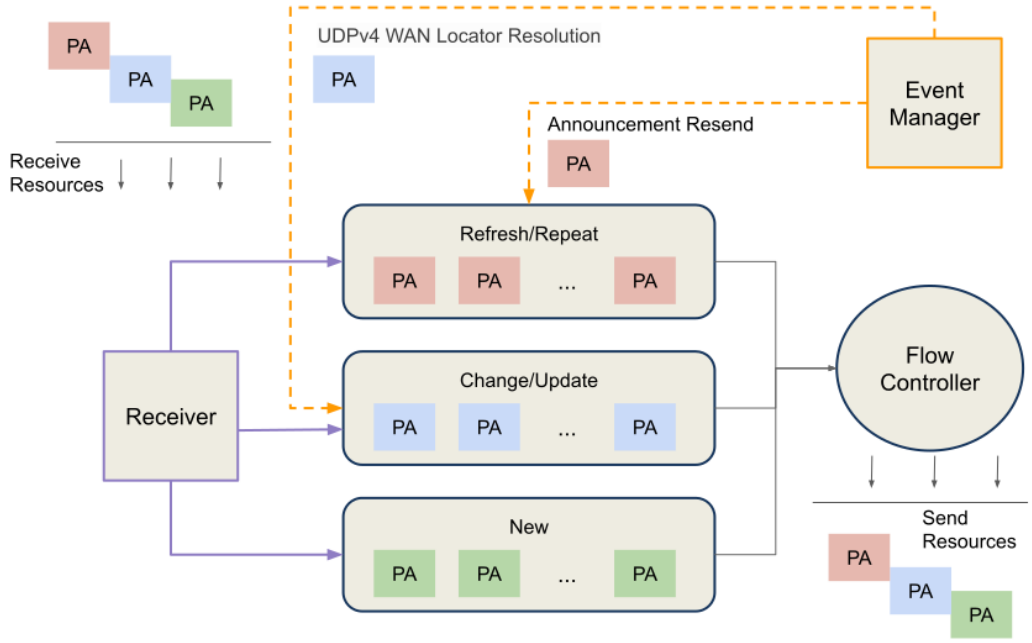

Figure 1.10: Participant Announcement Forwarder

- **Receiver**: The element in charge of retrieving incoming participant announcements. This element interfaces with the *receive resources*, which provide the participant announcements received from all the peer *DomainParticipants*.
- **Announcement Queues**: The received participant announcements are placed in queues so they can be forwarded to their proper destinations. Participant announcements can be categorized in three classes:
  - *New*: Announcements from *DomainParticipants* that are considered new from the service perspective.
  - *Change/Update*: Announcements from *DomainParticipants* that the service has already seen but that contain content changes relative to the previous announcement from the same *DomainParticipant*. For example, QoS or Locator changes or Locator resolutions for the *DomainParticipant*.
  - *Refresh/Repeat*: Announcements from *DomainParticipants* that the service has already seen and that do not contain any content changes from the previously received announcement.

  This classification allows the forwarder to process the different categories differently, allowing you to prioritize and regulate the bandwidth used by each traffic class.
- **Flow Controller**: The entity that removes announcements from the queue and forwards them to the proper destinations. The flow controller regulates the output announcement traffic. The flow controller uses the send resources to direct the announcements to the proper *DomainParticipants*.
- **Event Manager**: This element is in charge of handling events like UDPv4 WAN locator resolution and announcement resend. Each of these events is described below:
  - *UDPv4 WAN Locator Resolution*: This event happens when the *Real-Time WAN Transport* is used as a transport in a WAN scenario. In such an environment due to the presence of a NAT, locators

in the incoming participant announcement are resolved asynchronously to their public IP addresses. Once the locator is resolved by *Cloud Discovery Service*, it creates a *job* of the *Change* type. For more details on this scenario refer to section *NAT Traversal*.

– *Announcement Resend*: This event is triggered when *Cloud Discovery Service* has been configured to perform resends. Resends are performed when a *New* or *Change* announcement is received. It is also performed after a *UDPv4 WAN Locator Resolution* event occurs, which is considered as a *Change*. Resend artifically generates *jobs* of the announcement kind *Repeat* that send out multiple copies of an incoming or resolved participant announcement.

  Resends are particularly useful in networks with high probability of packet loss that hampers discovery speed. This is especially true when using the *Real-Time WAN Transport* for connectivity over the internet. For more details on configuring this mechanism refer to section *Forwarder*.

Flow Controller

Cloud Discovery Service incorporates a configurable flow controller, which allows shaping the generated output traffic as a result of forwarding participant announcements. The parameters that configure a flow controller are:

- **Output capacity**: The maximum amount of announcements forwarded in a period of time. It is measured in jobs-per-time units (e.g., announcements per second). Each received announcement represents a *job* to forward the announcement to all discovered *DomainParticipants*. This parameter allows setting an upper bound to the output traffic to avoid network congestion.
- **Maximum job burst**: The maximum amount of consecutive jobs that can be forwarded in a period of time. This parameter allows setting an upper limit to the output traffic peak.
- **Flush period**: The period at which the flow controller attempts to forward pending announcements.

Note: Because forwarding an announcement requires sending each announcement to multiple discovered participants, the actual output bandwidth depends on the number of participants.

Operation Mode

The flow controller generates *job tokens* at the rate specified by the *output capacity*. Forwarding an announcement takes exactly one job token. The forwarder queues announcements upon reception and attempts to forward them as soon as tokens are available.

If tokens are not available, the announcement jobs will remain in the queue in a pending state. The forwarder wakes up at every *flush period* to check for available tokens to forward the pending announcements.

If tokens are generated faster than jobs are received, the flow controller accumulates the tokens for future jobs. The flow controller will accumulate no more than the *maximum job burst* tokens.

1.3.4 Database

Cloud Discovery Service uses an internal database to keep information about remote entities. This is the information that represents the discovery state of the system.

This state is maintained upon reception of discovery information. Remote entries in the database are added or removed based on the received information. The database will release the resources for each removed entry as needed.

Cloud Discovery Service relies on a dedicated thread, which periodically cleans up any removed state from the database. This model enhances concurrency while maintaining thread safety.

This element is equivalent to the Database Thread of a *DomainParticipant*.

1.4 NAT Traversal

1.4.1 Introduction

RTI Cloud Discovery Service incorporates functionality that allows *Connext* applications situated behind Network Address Translators (NATs) to discover and communicate with each other using UDP. This is possible when *Cloud Discovery Service* is configured to use *RTI® RTI Real-Time WAN Transport*[1]. For further details, see the Real-Time WAN Transport in the Core Libraries User's Manual.

Note: For a better understanding of this section, we recommend that you become familiar with the basic concepts explained in *The Basics*. We also assume that you are familiar with NAT traversal concepts and challenges. You can refer to *Key Terms* for a list of definitions and conventions used in this section.

Cloud Discovery Service **can assist in the discovery and NAT traversal** in the scenario depicted in Figure 1.11. Communication between two remote *DomainParticipants* that sit behind NATs cannot occur by any means, because they are unable to know how to reach each other.

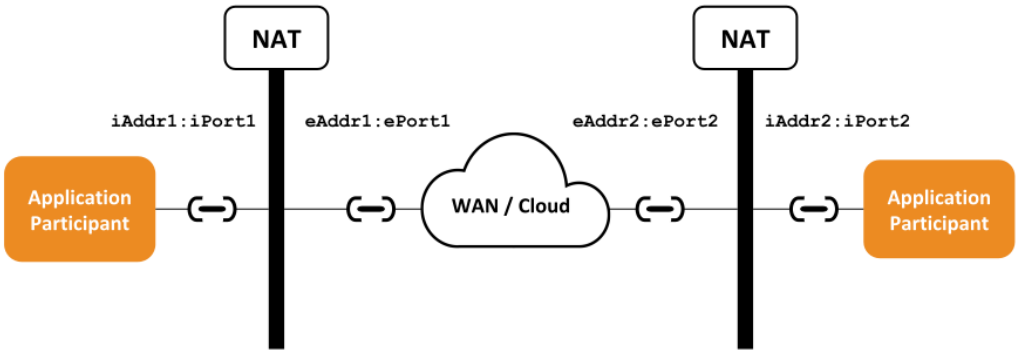

Figure 1.11: Communication challenge between applications behind NATs

[1] *RTI Real-Time WAN Transport* may require an additional license. Contact support@rti.com or your sales representative at RTI for further information.

DomainParticipants sitting behind a NAT are only aware of their own private IP transport addresses (`iAddr:iPort`). These are the only addresses they can exchange as part of the *locators list* they announce in their *DomainParticipant* announcements. In addition, their actual public IP transport addresses depend on the *NAT forwarding rules*, which are for the most part dynamic and impossible to know beforehand – unless a static configuration is set in the NAT-enabled routers.

In this situation, the only alternative is to consider a third player that can inform the remote *DomainParticipants* how they can reach each other through their public addresses. This third player is *Cloud Discovery Service*, in combination with the *RTI Real-Time WAN Transport* (RWT), which is used in the *DomainParticipants*.

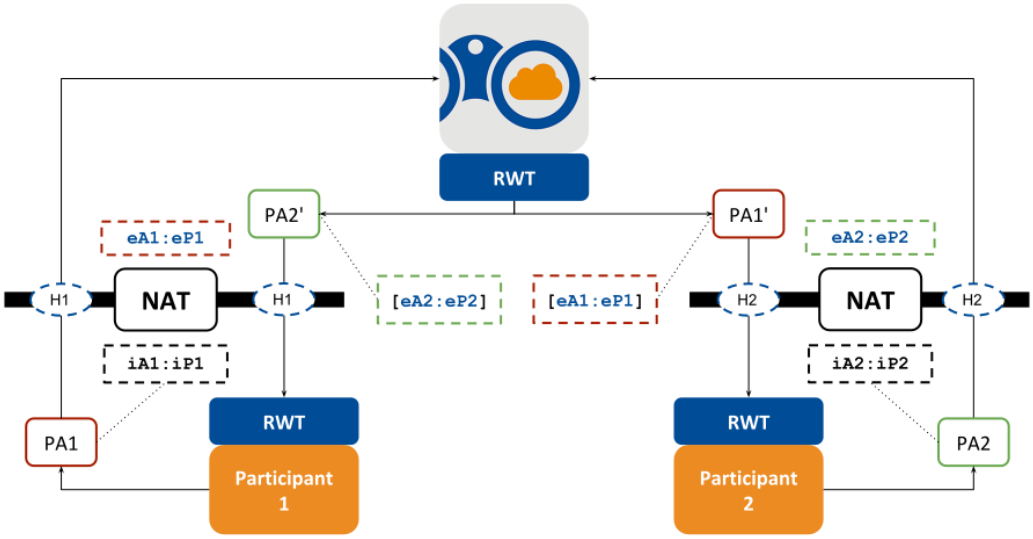

Figure 1.12: Public address resolution through *Cloud Discovery Service*

Figure 1.12 shows how *Cloud Discovery Service* acts as the entity that provides the **resolution of public IP transport addresses**. *Cloud Discovery Service* acts as an externally reachable service that obtains the public IP transport addresses from a *DomainParticipant* and provides them to the other *DomainParticipants* using RTPS locators. A resolved public IP transport address is called a *service reflexive address (SRA)*.

Let's examine Figure 1.12 in more detail to understand how *Cloud Discovery Service* assists with the NAT traversal. The scenario depicts two *DomainParticipants* behind NATs. Each *DomainParticipant* sends a *participant announcement* (or *PA*) to *Cloud Discovery Service* containing WAN locators. These locators can be UUID locators or UUID+PUBLIC locators depending on the configuration of *RTI Real-Time WAN Transport*. *Cloud Discovery Service* cannot forward the original *PAs* to peer *DomainParticipants* if they contain UUID locators, which are unreachable. Instead, *Cloud Discovery Service* will obtain the service reflexive address `eA:eP`, where `eA` is the public IPv4 address and `eP` is the public UDP port, and use it to replace the original UUID locator in the *PA* with a UUID+PUBLIC locator. *DomainParticipants* receive the forwarded and modified *PAs* from *Cloud Discovery Service*. These contain the public service reflexive addresses that are **potentially reachable, thus making peer-to-peer communication possible**.

Note that we say the *SRAs* are *potentially reachable*. The point is that these addresses will be reachable depending on the type of NAT that the *DomainParticipant* sits behind. We will see in further sections that only *Cone NATs* allow this communication, whereas *Symmetric NATs* don't.

To help you understand this behavior, see *Example: Using RTI Real-Time WAN Transport*, where you will emulate the scenario described above in Figure 1.12. In that example, we will analyze step-by-step what you

need to configure in *Cloud Discovery Service* and the application *DomainParticipants*. You can run the example first if you want to see it in action.

See also:

Network Address Translation

UDP Hole Punching

1.4.2 Running *Cloud Discovery Service* with *RTI Real-Time WAN Transport*

To perform the public address resolution needed to assist in the discovery and communication of remote *DomainParticipants* behind NATs, you will need to run a *Cloud Discovery Service* instance on a publicly reachable host and enable *RTI Real-Time WAN Transport* (RWT). For that, you will need to configure the following *Cloud Discovery Service* parameters:

- Service or host port `host_port`: This is the UDP port number where the *Cloud Discovery Service* instance will listen for *PAs*. This number shall be within the valid range for UDP ports and not taken by any other application running on the host.
- Public IPv4 address `public_addr`: This is the publicly reachable address that external *DomainParticipants* can use to access the service host.
- Public port `public_port`: This is the UDP port number that external remote *DomainParticipants* use to communicate with the service running on `host_port`.

If the service host is directly accessible to the WAN with no firewalls/NAT, the `host_port` is the same as the `public_port`, and `public_addr` also matches the local host address. If the service host is behind a NAT, then `public_port` represents the *forwarded port* statically configured in the NAT device, and **can be a different number than** `host_port`; also the `public_addr` **will be different than the local host address**.

Cloud Discovery Service configuration

To run *Cloud Discovery Service* using RWT, we will need to specify the `host_port` and `public_addr`. For example, in the following configuration:

```
<cloud_discovery_service name="CdsWanUdp">
    <transport>
        <element>
            <alias>builtin.udpv4_wan</alias>
            <receive_port>7400</receive_port>
            <property>
                <element>
                    <name>dds.transport.UDPv4_WAN.builtin.public_address</
↪name>
                    <value>216.58.194.174</value>
                </element>
            </property>
        </element>
```

(continues on next page)

(continued from previous page)

```
    </transport>
</cloud_discovery_service>
```

- <receive_port> is set to the host_port.
- The RWT property public_address is set to the public_addr.

In our example, we use one of the builtin configurations that enables RWT and parameterizes the values for the <receive_port> and public_address property using XML configuration variables. This way allows you to reuse the configuration with different values for each deployed *Cloud Discovery Service* instance.

Note: With the above configuration, if CDS is behind a NAT-enabled router, the host_port must be the same as the public_port. If you want to use a different public_port, it will be necessary to configure the property dds.transport.UDPv4_WAN.builtin.comm_ports. For additional information on this property, see the Real-Time WAN Transport in the Core Libraries User's Manual.

Application *DomainParticipant* configuration

This is a very simple step where the application *DomainParticipant* is configured to use RWT and include the *Cloud Discovery Service* instance as part of the initial peers.

To enable RWT, simply select the corresponding builtin transport element in the *DomainParticipant* QoS configuration:

```
<domain_participant_qos>
    ...

    <transport_builtin>
        <mask>UDPv4_WAN</mask>
    </transport_builtin>
</domain_participant_qos>
```

Then when you run the application, set the initial peers to the *Cloud Discovery Service*, which is identified as follows:

```
rtps@public_addr:public_port
```

where public_addr and public_port are the properties described above and determine the public IP transport address that shall be reachable by external applications. To set the initial peers, you can use two methods:

- Using the NDDS_DISCOVERY_PEERS environment:

  ```
  export NDDS_DISCOVERY_PEERS=rtps@udpv4_wan://216.58.194.174:7400
  ```

- Configuring the initial_peers QoS:

```
<domain_participant_qos>
    ...

    <discovery>
        <initial_peers>
            <element>rtps@udpv4_wan://216.58.194.174:7400</element>
        </initial_peers>
    </discovery>
</domain_participant_qos>
```

Warning: The assumption is that `<accept_unknown_peers>` is set to true (which is the default value). This is important or else communication between *DomainParticipants* will not occur.

Then just run your applications with this setup and they will communicate over the WAN using DDS, purely peer-to-peer. To understand how this is possible, let's look at Figure 1.13.

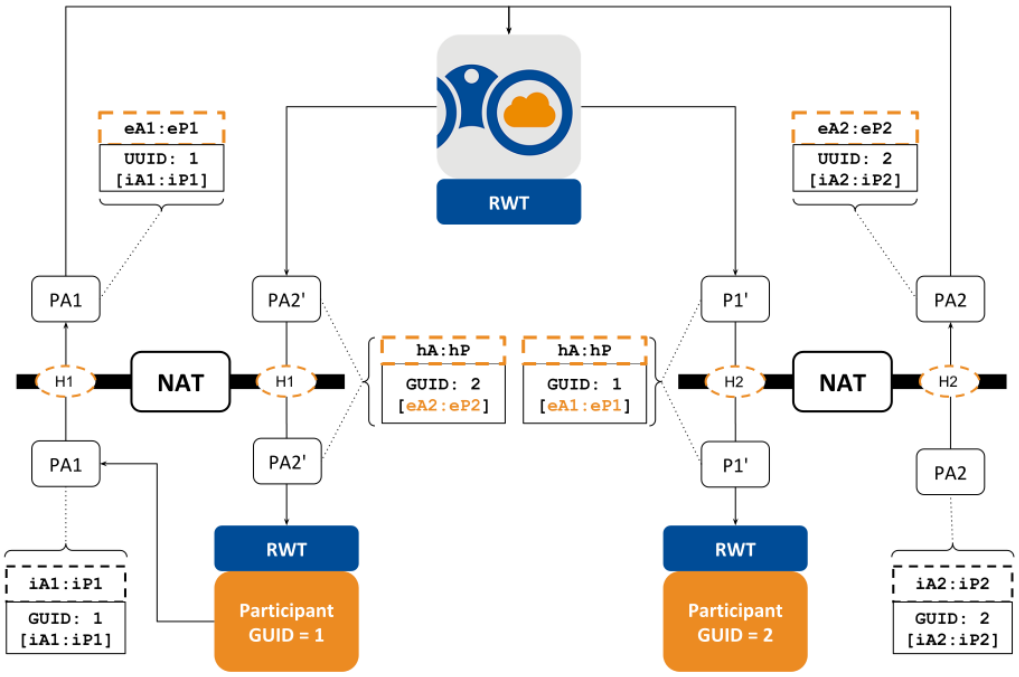

Figure 1.13: Resolution of public address for *DomainParticipants* behind Cone NATs

In this scenario, there are two *DomainParticipants* behind Cone NATs, each uniquely identified by a *global unique identifier* (GUID). The *DomainParticipant* with `GUID=1` has two private IP transport addresses, `iA1:iP1` and `iA1:iP2` (for simplicity, Figure 1.13 only shows `iA1:iP1`). `iA1:iP1` is used to exchange DDS discovery traffic, `iA1:iP2` is used to exchange user data traffic. *DomainParticipant* `GUID=1` sends a *DomainParticipant* announcement `PA1` containing two UUID locators, each one associated with one of the private IP transport addresses.

Upon reception of `PA1`, *Cloud Discovery Service* inspects the received UDP packet that contains the announcement and extracts the public IP transport addresses (`eA1:eP1` and `eA2:eP2`) for each one of the UUID

locators. These are the service reflexive addresses (SRAs) described above. The key step occurs when *Cloud Discovery Service* **modifies the original announcement** `PA1` to extend each UUID WAN locator so it contains the associated SRA. This extended announcement `PA1'` is the one forwarded to the *DomainParticipant* with `GUID=2`, which can then use the SRA addresses to communicate peer-to-peer with *DomainParticipant* `GUID=1`. The same process applies to *DomainParticipant* `GUID=2` and, in general, to any remote *DomainParticipant* in the domain.

In some cases, it is desirable to use a single public address for communication. This can be achieved by updating the configuration of RWT in the *DomainParticipants* as follows:

```
<domain_participant_qos>
    ...
    <transport_builtin>
        <udpv4_wan>
            <comm_ports>
                <default>
                    <host>16000</host>
                </default>
            </comm_ports>
        </udpv4_wan>
    </transport_builtin>
</domain_participant_qos>
```

With this configuration, the *DomainParticipant* with `GUID=1` will have only one private IP transport address, `iA1:iP1`, where `iP1` is 16000. The communication with the *DomainParticipant* will occur on a single public address `eA1:eP1`.

For additional details on how to configure RWT, see the Real-Time WAN Transport in the Core Libraries User's Manual.

Communication between *DomainParticipants*

So far, we have explained the address resolution step that *Cloud Discovery Service* provides so that application *DomainParticipants* that sit behind NATs can communicate between themselves.

The address resolution is just the first step in providing the *DomainParticipants* with the public addresses of their peers. The second step is to establish communication between them: that occurs solely through RWT, without needing *Cloud Discovery Service*. Figure 1.14 provides a simplified view of how the communication is possible when the *DomainParticipants* are configured to use a single UDP port for communication (single WAN locator).

A local *DomainParticipant* `P1` will receive the resolved public address of a peer application *DomainParticipant* `P2` from *Cloud Discovery Service*, that is, `eA2:eP2`. This is the address *DomainParticipant* `P1` uses to reach the peer remote *DomainParticipant* `P2`. In the process of sending DDS traffic `D1` to *DomainParticipant* `P2`, the NAT will temporarily open a hole `H1`. When `D1` arrives at the host of `P2`, the NAT will let the incoming traffic come through hole `H2`. This is possible because:

- *DomainParticipant* `P2` is in parallel performing the same communication protocol, hence opening a temporary hole `H2`.
- Both *DomainParticipants* sit behind NATs that always perform the same mapping between a private address `iA:iP` and a public address `eA:eP`, no matter the destination. This category of NATs includes:

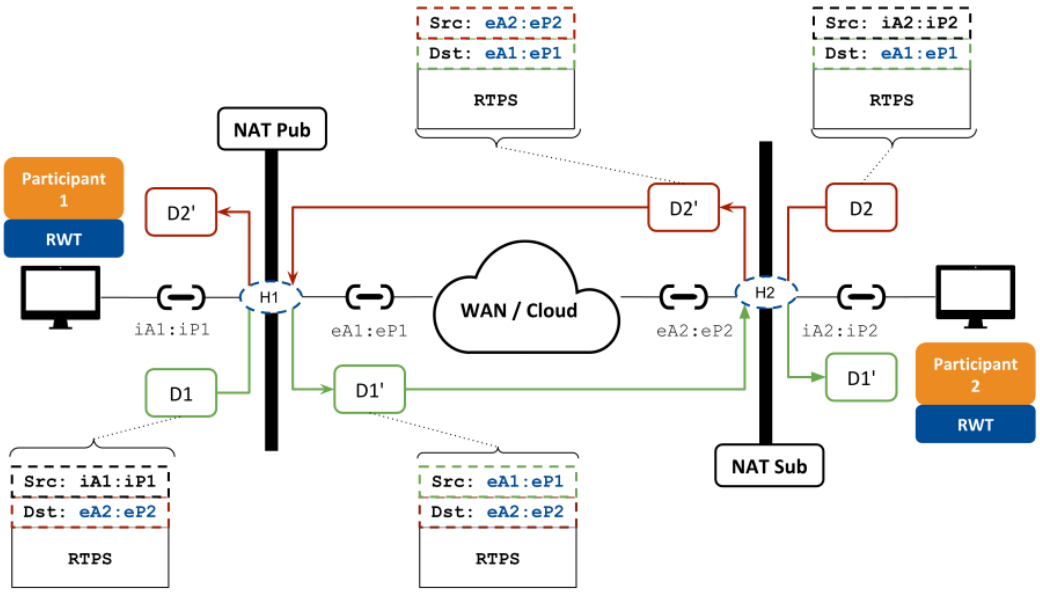

Figure 1.14: Direct communication over the WAN between *DomainParticipants* behind Cone NATs

- *Full-cone*
- *Restricted-cone*
- *Port-restricted cone*

This is not true for Symmetric NATs, which change the mapping based on the destination, so it's not possible to use the SRA that *Cloud Discovery Service* obtains.

1.4.3 Communication scenarios using *Cloud Discovery Service*

Cloud Discovery Service support two basic WAN connectivity secenarios:

1. Communication between *DomainParticipants* behind Cone-NATs.
2. Communication with a *DomainParticipant* with well-known, reachable public IP addresses. If the *DomainParticipant* is behind a NAT, these public addresses must be statically configured in the NAT-enabled router.

For additional information see the Real-Time WAN Transport in the Core Libraries User's Manual.

1.4.4 Debugging *Cloud Discovery Service* with the UDP WAN Transport

You can find out how *Cloud Discovery Service* is resolving addresses and for which *DomainParticipants*. Locator content depends on the transport implementation. The locators for *RTI Real-Time WAN Transport* convey the following information:

- **Flags**: Indicate the type of WAN locator and included information.
- **UUID**: Unique identifier that typically refers to a single network interface.
- **Public address**: Contains the public IPv4 network address and UDP public port.

See the locator format section in *RTI Connext DDS Core Libraries User's Manual* for more information about locators.

You can obtain locator resolution information in two ways: **logging** and **remote administration**.

Logging

Run *Cloud Discovery Service* with `-verbosity ALL` to enable the output of resolution log messages. The WAN locators will be represented with the following string format:

```
f=<flags>,u={<uuid>,P=<eAddr:ePort>}
```

You will find two type of messages:

- **New resolution**: This is a log message generated in the event of resolving the IP public address for a UUID WAN locator. The log message will provide two important pieces of information. In the log context, in the content between the `[]`, you will find the full WAN locator address in hexadecimal; in the message portion of the log, you will find the UUID and its resolved IP public address. For example, this log:

  ```
  [...|RESOLVE{UUID+PUBLIC=0x07BD73FC,0x9ED500AB,0x6D40DCD4:0xBC4E31D3}] \
          f=7,u={BD,73,FC,9E,D5,00,AB,6D,40},P=188.78.49.211:56532
  ```

  shows the following information:

  - UUID: `BD,73,FC,9E,D5,00,AB,6D,40`. Note that these nine bytes are the same as the first portion of the hexadecimal address in the log context.
  - Public IPv4 Address: `188.78.49.211`
  - Public UDP Source Port: `56532`

- **Resolution status**: This is a log message displayed upon reception and forwarding of a *DomainParticipant* announcement (PA). This message displays the set of *DomainParticipant* locators in JSON format, for both meta-traffic and user-traffic, including the original locators received in the announcement and their resolved equivalent. For example:

  ```
  [...|DATAP{GUID=0x0101C372,0x17D4CED6,0xEA7B3DA5:0x000001C1}]
  {
     "locators": {
  ```

(continues on next page)

(continued from previous page)

```
    "original":{
        "metatraffic": [
                "udpv4_wan://f=1,u={BD,73,FC,9E,D5,00,AB,6D,40},P=10.0.
↪2.15:0:32410"
            ],
            "default":[
                "udpv4_wan://f=1,u={BD,73,FC,9E,D5,00,AB,6D,40},P=10.0.
↪2.15:0:32411"
            ]
    },
    "resolved":{
        "metatraffic": [
            "udpv4_wan://f=7,u={BD,73,FC,9E,D5,00,AB,6D,40},P=188.78.49.
↪211:56532:32410"
        ],
        "default": [
            "udpv4_wan://f=7,u={BD,73,FC,9E,D5,00,AB,6D,40},P=188.78.49.
↪211:51788:32411"
        ]
    }
  }
}
```

The last part of the context portion of the log identifies the remote *DomainParticipant* for which the locators are displayed; in this case it provides the GUID in hexadecimal format (`GUID=0x0101C372, ...`). The message portion of the log shows that there are two UUID locators, one for each type of traffic, and it shows that each of them has been resolved properly, displaying the resolved public address in a format similar to the resolution log shown above.

Note that the number of items between the `original` and `resolved` lists may differ and, in fact, `length(resolved) <= length(original)`. If a UUID locator is not resolved or the resolved list is empty, it may be an indication of a connectivity problem. For example, the following log indicates that UUID locators have not been resolved:

```
[...|DATAP{GUID=0x0101C372,0x17D4CED6,0xEA7B3DA5:0x000001C1}]
{
  "locators": {
    "original": {
        "metatraffic": [
            "udpv4_wan://f=1,u={BD,73,FC,9E,D5,00,AB,6D,40},P=10.0.2.
↪15:0:32410"
        ],
        "default": [
            "udpv4_wan://f=1,u={BD,73,FC,9E,D5,00,AB,6D,40},P=10.0.2.
↪15:0:32411"
        ]
    },
    "resolved":{
        "metatraffic": [],
        "default": []
    }
```

(continues on next page)

(continued from previous page)

```
    }
  }
```

If you run into this situation, you can check that:

- *Cloud Discovery Service* and the host it runs on are properly configured to receive external traffic on the receive port.
- The remote *DomainParticipants* have properly set their initial peers to the public transport address where *Cloud Discovery Service* is reachable. Note that this address will be different than the host address of *Cloud Discovery Service* if it runs behind a NAT.

Administration

You can obtain a one-time snapshot of the locator resolution state for a given *DomainParticipant* using the remote administration capabilities (*Remote Administration*). *Cloud Discovery Service* offers a remote operation to retrieve the locator resolution state for a specified remote *DomainParticipant* (see *the API specification*).

For example, consider the resolution state log shown above. You could obtain the *DomainParticipant* locators by issuing the following command:

```
method: GET
resource: /cloud_discovery_services/[name]/database/locators
string_body: "0x0101C372,0x17D4CED6,0xEA7B3DA5:0x000001C1"
```

and you will get a response containing the resolution state in JSON format:

```
{
  "locators": {
     "original":{
        "metatraffic":[
            "udpv4_wan://f=1,u={BD,73,FC,9E,D5,00,AB,6D,40},P=10.0.2.
↪15:0:32410"
        ],
        "default":[
            "udpv4_wan://f=1,u={BD,73,FC,9E,D5,00,AB,6D,40},P=10.0.2.
↪15:0:32411"
        ]
    },
    "resolved":{
        "metatraffic":[
            "udpv4_wan://f=7,u={BD,73,FC,9E,D5,00,AB,6D,40},P=188.78.49.
↪211:56532:32410"
        ],
        "default":[
            "udpv4_wan://f=7,u={BD,73,FC,9E,D5,00,AB,6D,40},P=188.78.49.
↪211:51788:32411"
        ]
    }
  }
}
```

Identifying the NAT type

As we have mentioned, peer-to-peer communication between the remote *DomainParticipants* can only occur if they sit behind Cone NATs. Communication will not be possible if any of them is behind a Symmetric NAT. Therefore, it is mandatory that you verify the type of NAT your applications run.

There are multiple third-party utilities that you can download to find out about the NAT shielding your computer. One example is natat, a small open-source utility you can run locally to find out your NAT kind.

1.4.5 Key Terms

NAT Traversal A mechanism to establish peer-to-peer connections across gateways that sit behind NATs.

IP Transport Address (or Address) The combination of the IPv4 address and the UDP port where an application accepts incoming traffic. Sometimes you will also see the term "address" being used to refer to an IP transport address when the context is clear.

Public IP Transport Address (or Public Address) An IP transport address for an application that is routable on a WAN. When the WAN is the Internet, the term "Internet-routable address" can be used instead.

Private IP Transport Address (or Private Address) The IP transport address of an application that sits behind a NAT. This address is not reachable from external applications running outside the NAT.

`eAddr:ePort` A public IP transport address, where `eAddr` is the public or external IPv4 address and `ePort` is the UDP public or external port.

`iAddr:iPort` A private IP transport address, where `iAddr` is the private IPv4 address and `iPort` is the UDP host port.

Service Reflexive Address The public IP transport address that *Cloud Discovery Service* obtains from the incoming UDP packets and is used for address resolution.

RTPS Locator (or Locator) A DDS endpoint (DataWriter or DataReader) address unit that consists of a transport class, RTPS port, and locator transport address (128-bit).

Reachable locator: Locator associated with a DDS endpoint (DataWriter or DataReader) to which another DDS endpoint can send data.

RTPS WAN Locator (or WAN Locator) A locator for *RTI Real-Time WAN Transport*.

RTPS UUID WAN Locator (or UUID Locator) A WAN locator for *RTI Real-Time WAN Transport* that is not reachable. *Cloud Discovery Service* transforms UUID locators into UUID+PUBLIC locators by associating a public IP transport address to the UUID. The public IP transport address for the UUID locator is the service reflexive address.

RTPS UUID+PUBLIC WAN Locator (or UUID+PUBLIC Locator) A WAN locator for *RTI Real-Time WAN Transport* that is reachable. The locator encapsulates a public IP transport address as part of the locator address.

Address Resolution The process of identifying the public address at which an application behind a NAT is reachable.

NAT forwarding mapping A static configuration in the NAT device that allows mapping a public address to a private address, so external applications can send and receive data.

Application or Remote *DomainParticipant* A *DomainParticipant* that is part of a *Connext* application.

UDP Hole-Punching A NAT traversal mechanism that consists of creating a temporary UDP forwarding mapping for an internal address.

GUID string representation A representation of a *DomainParticipant* GUID, in a hexadecimal string with the following notation: `host_id,app_id,instance_id:object_id`.

1.5 Usage

This chapter explains how to run *Cloud Discovery Service* either from the distributed command-line executable or as a library within your application.

1.5.1 Command-Line Executable

Cloud Discovery Service runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin`.

```
rticlouddiscoveryservice [options]
```

This section explains how to run *Cloud Discovery Service* from a command-line tool. In particular, it describes:

- Starting *Cloud Discovery Service* (Section 1.5.1).
- Stopping *Cloud Discovery Service* (Section 1.5.1).
- Command-line Options (Section 1.5.1).

Starting Cloud Discovery Service

Cloud Discovery Service runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin`.

```
rticlouddiscoveryservice [options]
```

To start *Cloud Discovery Service* with a default configuration, enter:

```
$NDDSHOME/bin/rticlouddiscoveryservice
```

This command will run *Cloud Discovery Service* indefinitely until you stop it. See Section 1.5.1.

Note: *Cloud Discovery Service* is pre-loaded with a builtin configuration that has default settings. See Section 1.6.3.

Table 1.4 describes the command-line parameters.

Stopping Cloud Discovery Service

To stop *Cloud Discovery Service*, press Ctrl-c. *Cloud Discovery Service* will perform a clean shutdown.

Command-Line Options

The following table describes all the command-line parameters available in *Cloud Discovery Service*. To list the available commands, run `rticlouddiscoveryservice -h`.

Table 1.4: Cloud Discovery Service Command-line Options

Parameter	Description
`-allowDomain <string>`	Subset of domain IDs where *Cloud Discovery Service* operates. Use this argument in order to indicate which domain IDs *Cloud Discovery Service* should process announcements for. Remaining ones will be automatically ignored. **Default:** `DOMAIN_LIST_ALL` (forward all domains)
`-appName <string>`	Assigns a name to the execution of the *Cloud Discovery Service*. Remote commands and status information will refer to the instances using this name. **Default:** (same as configuration name).
`-cfgFile <string>`	Path to the configuration file. **Default:** (unspecified)
`-cfgName <string>`	Name of the *Cloud Discovery Service* configuration to be loaded. It must match a `<cloud_discovery_service>` tag in the configuration file. **Default:** rti.cds.builtin.config.default.
`-denyDomain <string>`	Subset of domain IDs ignored by *Cloud Discovery Service*. Use this argument in order to indicate which domain IDs *Cloud Discovery Service* shouldn't process announcements from. **Default:** Empty string (no domain IDs are ignored) **Overrides:** `domain_list/allow_domain_id` or `-allowDomain`
`-heapSnapshotDir <dir>`	Output directory where the heap monitoring snapshots are dumped. The filename format is RTI_heap_<appName>_<processId>_<index>. Used only if heap monitoring is enabled. **Default**: current working directory
`-heapSnapshotPeriod <sec>`	Period at which heap monitoring snapshots are dumped. Enables heap monitoring if > 0. **Default**: 0 (disabled)
`-help`	Prints this help and exits.
`-ignoreXsdValidation`	Loads the configuration even if the XSD validation fails.
`-listConfig`	Prints the available configurations and exits.
`-logFormat <format>`	A mask to configure the format of the log messages for both the service and DDS. It allows the following values: • `DEFAULT` - Print message, method name, log level, activity context, and logging category • `VERBOSE` - Print `DEFAULT` information, plus: module, thread ID, and message location (and spread the message over two lines) • `TIMESTAMPED` - Print `VERBOSE` information, timestamped • `MINIMAL` - Print only message number and message location • `MAXIMAL` - Print all available fields **Default:** DEFAULT
`-maxObjectsPerThread <int>`	Maximum number of thread-specific objects that can be created. **Default:** Same as the Connext default for max_objects_per_thread
`-remoteAdministrationDomainId <int>`	Enables remote administration and sets the domain ID for communication. **Overrides:** `administration/domain_id`
`-remoteMonitoringDomainId <int>`	Enables remote monitoring and sets the domain ID for status publication. **Overrides:** `monitoring/domain_id`
`-stopAfter <int>`	Number of seconds the *Cloud Discovery Service* runs before it stops. **Default:** (infinite).
`-transport <string>`	A comma separated list of transport resources, where each resource is specified in the form: `[alias:]receive_port` **Default:** udpv4:7400 **Overrides:** `transport`
`-licenseFile <string>`	Specifies the license file path to be used. See *How to use a License File with RTI Services*. **Default:** Empty string
`-D<name>=<value>`	Defines a user variable that can be used as an alternative replacement for

All the command-line options are optional; if specified, they override the values of their corresponding settings in the loaded XML configuration. See Section 1.6 for the set of XML elements that can be overriden with command-line options.

1.5.2 Cloud Discovery Service as a Library

Cloud Discovery Service can be deployed as a library linked into your application on selected architectures (see Section 1.14). This allows you to create, configure, and start *Cloud Discovery Service* instances from your application.

To build your application, add the dependency with the *Cloud Discovery Service* library under `<NDDSHOME>/lib/<ARCHITECTURE>`, where `<ARCHITECTURE>` is a valid and installed target architecture.

Here is an example of using *Cloud Discovery Service* as a library with the Library API:

C

```
struct RTI_CDS_Property property =
        RTI_CDS_Property_INITIALIZER;
struct RTI_CDS_Service *service = NULL;

/* Initialize property */
property.cfg_file     = "my_cds_service_cfg.xml";
property.service_name = "my_cds_service";
...

service = RTI_CDS_Service_new(&property);
if(service == NULL) {
    /* Log error */
    ...
}

if(!RTI_CDS_Service_start(service)) {
    /* Log error */
    ...
}

while(keep_running) {
    sleep();
    ...
}
...

RTI_CDS_Service_delete(service);
```

C++

```
using namespace rti::cds;

CloudDiscoveryService service(
        ServiceProperty()
                .cfg_file("my_cds_service_cfg.xml")
```

(continues on next page)

(continued from previous page)

```
                .service_name("my_cds_service"));

service.start();
```

1.5.3 Operating System Daemon

See generic instructions in *How to Run as an Operating System Daemon*.

1.6 Configuration

1.6.1 Configuring Cloud Discovery Service

This section provides a reference for the XML elements that conform a *Routing Service* configuration. For details on how to provide XML configurations to *Routing Service*. refer to *Configuring RTI Services*. This chapter describes how to write an XML configuration.

1.6.2 XML Tags for Configuring RTI Cloud Discovery Service

This section describes the XML tags you can use in a *Cloud Discovery Service* configuration file. The following diagram and Table 1.5 describe the top-level tags allowed within the root `<dds>` tag.

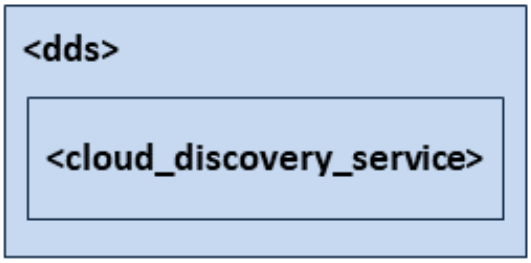

Figure 1.15: Top-level Tags in the Configuration File

Table 1.5: Top-level Tags in the Configuration File

Tags within <dds>	Description	Multiplicity
<cloud_discovery_service>	Specifies a *Cloud Discovery Service* configuration. **Attributes** • name: uniquely identifies a service configuration. **Required**. **Example** <cloud_discovery_service name="ExampleService"> <!-- your service settings ... --> </cloud_discovery_service>	1..*.

Cloud Discovery Service

A configuration file must have at least one <cloud_discovery_service> tag. This tag is used to configure an execution of *Cloud Discovery Service*. A configuration file may contain multiple <cloud_discovery_service> tags.

When you start *Cloud Discovery Service*, you can specify which <cloud_discovery_service> tag to use to configure the service using the -cfgName command-line option.

Because a configuration file may contain multiple <cloud_discovery_service> tags, one file can be used to configure multiple *Cloud Discovery Service* executions.

Figure 1.16 and Table 1.6 describe the tags allowed within a <cloud_discovery_service> tag.

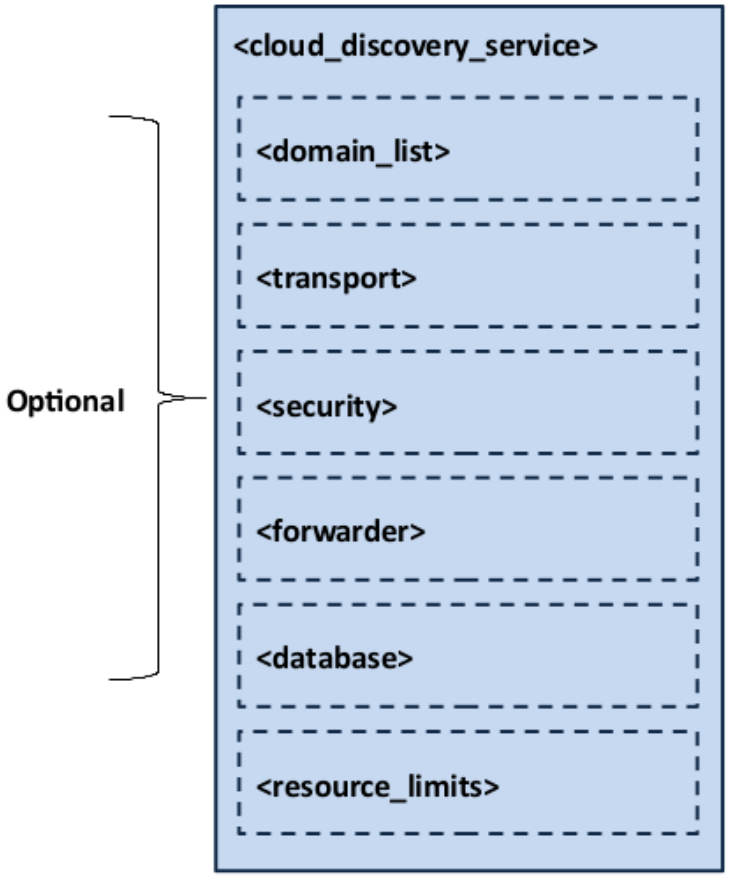

Figure 1.16: Cloud Discovery Service Tag Structure

Table 1.6: Cloud Discovery Service Tag

Tags within <cloud_discovery_service>	Description	Multiplicity
<administration>	Enables remote administration. When administration is enabled, monitoring is also enabled by default. If no domain ID is specified for monitoring, *Cloud Discovery Service* will use the same domain as administration by default. See *Administration*.	0..1
<monitoring>	Enables monitoring for *Cloud Discovery Service*, including the periodic publication of statistics. See Section 1.6.2.	0..1
<domain_list>	Set of domains for which the service forwards participant announcements. See *Domain List*.	0..1
<transport>	Selection of unicast transport resources where the service receives and sends participant announcements. See *Transport*.	0..1
<security>	Configures the security features provided by the *RTI Security Plugins*. See *Security*.	0..1
<forwarder>	Configures the behavior of the participant announcement forwarding logic. See *Forwarder*.	0..1
<database>	Configures the behavior of the service's internal database, which contains information about the discovery state. See *Database*.	0..1
<resource_limits>	Service resource management policies. See *Resource Limits*.	0..1

Example: Specify a Configuration in XML

```
<dds>
    <cloud_discovery_service name="EmptyConfiguration"/>
    <cloud_discovery_service name="ShapesDemoConfiguration">
    <!--...-->
    </cloud_discovery_service>
</dds>
```

Starting *Cloud Discovery Service* with the following command will use the <cloud_discovery_service> tag with the name EmptyConfiguration.

```
$NDDSHOME/bin/rticlouddiscoveryservice \
        -cfgFile file.xml -cfgName EmptyConfiguration
```

Administration

The <administration> tag allows you to enable and configure remote administration of *Cloud Discovery Service*, including stopping, starting, and deleting a *Cloud Discovery Service* instance.

See *Remote Administration* for details on using remote administration.

Table 1.7: Administration Tags in Cloud Discovery Service's Configuration File

Tags within <administration>	Description	Multiplicity
<domain_id>	Domain ID used for remote administration. Also used for monitoring by default.	0..1
<domain_participant_qos>	QoS used by the administration DomainParticipant. If the tag is not defined, *Connext* defaults will be used.	0..1
<publisher_qos>	QoS used by the administration Publisher. If the tag is not defined, *Connext* defaults will be used.	0..1
<subscriber_qos>	QoS used by the administration Subscriber. If the tag is not defined, *Connext* defaults will be used.	0..1
<datawriter_qos>	QoS used by administration DataWriter(s). If the tag is not defined, *Connext* defaults will be used, with the following changes: • history.kind = DDS_KEEP_ALL_HISTORY_QOS • resource_limits.max_samples = 32	0..1
<datareader_qos>	Quality of Service (QoS) used by administration DataReader(s). If the tag is not defined, the *Connext* defaults will be used, with the following changes: • reliability.kind = DDS_RELIABLE_RELIABILITY_QOS (this value cannot be changed) • history.kind = DDS_KEEP_ALL_HISTORY_QOS • resource_limits.max_samples = 32	0..1
<distributed_logger>	When you enable *Distributed Logger*, *Cloud Discovery Service* will publish its log messages to *Connext*. See *Enabling Distributed Logger*.	0..1

The contents of the tags for configuring QoS are specified in the same manner as for the *Connext* XML QoS Profiles file. See Configuring QoS with XML, in the *RTI Connext* Core Libraries User's Manual.

Monitoring

The <monitoring> tag allows you to enable and configure remote monitoring of *Cloud Discovery Service*. See *Monitoring*.

Table 1.8: Monitoring Tags in *Cloud Discovery Service*'s Configuration File

Tags within <monitoring>	Description	Multiplicity
<enabled>	Whether to enable monitoring of the service. Default: Disabled, unless administration is enabled.	0..1
<domain_id>	Domain ID used for monitoring. Default: The domain ID specified for monitoring.	0..1
<datawriter_qos>	QoS used by monitoring DataWriter(s)	0..1
<publisher_qos>	QoS used by monitoring Publisher(s)	0..1
<domain_participant_qos>	QoS used by monitoring DomainParticipant	0..1
<statistics_sampling_period>	How frequently to sample the service's statistics, using the tags <sec> and <nanosec>. For example, <sec>1</sec> samples the service's statistics every second.	0..1
<status_publication_period>	How frequently to publish the service status, using the tags <sec> and <nanosec>. For example, <sec>1</sec> publishes the service's status every second.	0..1

The contents of the tags for configuring QoS are specified in the same manner as for the *Connext* XML QoS Profiles file. See Configuring QoS with XML, in the *RTI Connext* Core Libraries User's Manual.

Domain List

A <domain_list> allows you to control for which domains the discovery traffic is propagated. Table 1.9 describes the tags allowed.

Figure 1.17 and Table 1.9 describe the tags allowed within a <domain_list> tag.

Table 1.9: Domain List Tag

Tags within <domain_list>	Description	Multiplicity
<allow_domain_id>	Set of domain IDs where the service forwards participant announcements. **Default:** DOMAIN_LIST_ALL	0..1
<deny_domain_id>	Subset of the allowed domain IDs for which the service ignores announcements. **Default:** [empty string]	0..1

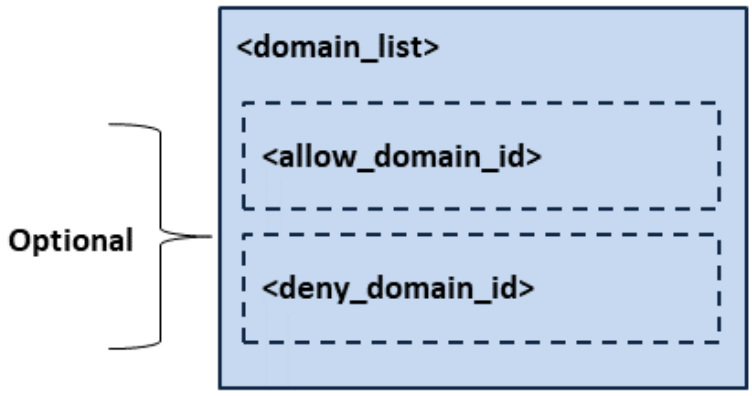

Figure 1.17: Domain List Tag Structure

The `<allow_domain_id>` and `<deny_domain_id>` filters both allow the same syntax, in which a subset of domains can be specified as a comma-separated list containing one or more of the following elements:

- Individual domains: `5, 6, 7`
- Domain Range: `[1 - 10]`
- **Special values:**
  - `DOMAIN_LIST_ALL`: specifies all the available domains.
  - (empty string): specifies none of the domains.

Example: Deny a Few Specific Domains

```
<domain_list>
   <allow_domain_id>DOMAIN_LIST_ALL</allow_domain_id>
   <deny_domain_id>5,7,10</deny_domain_id>
</domain_list>
```

Example: Allow a Subset of Domains

```
<domain_list>
   <allow_domain_id>[10 - 30], 40, [50 - 100]</allow_domain_id>
</domain_list>
```

Transport

The `<transport>` element allows you to select and configure the resources used to receive and send discovery traffic. A receive resource is uniquely specified by a transport class-receive port pair. For each different transport instance specified, *Cloud Discovery Service* creates a send resource.

Figure 1.18 and Table 1.10 shows the description of this element.

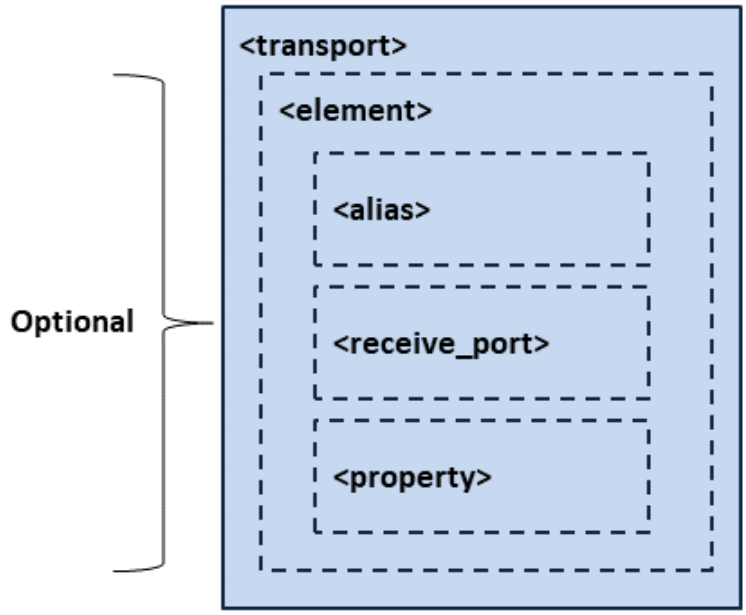

Figure 1.18: Transport Tag Structure

Table 1.10: Transport Tag

Tags within `<transport>`	Description	Multiplicity
`<element>`	Individual transport receive resource. See Table 1.11.	0..*

Each `<element>` within `<transport>` is a transport unicast receive resource, specified by a transport alias and a receive port. Table 1.11 describes the tags allowed.

Table 1.11: Transport Element Tag

Tags within <element> of a <transport>	Description	Multiplicity
<alias>	Identifies a concrete transport class instantiation. **Default:** udpv4 If the transport supports it, you can reuse the same transport instance by specifying the same alias. **Note:** If you attempt to reuse a transport instance of a class that does not support reuse, *Cloud Discovery Service* will log a warning and continue loading.	0..1
<receive_port>	Port that the service listens on to receive participant announcements. **Default:** 7400 **Attributes:** • kind: Specifies the port representation. Optional. – **PORT**: An integer that represents a port number within the valid range allowed by the transport. – **DOMAIN_ID_DERIVED**: A domain ID that represents the discovery receive port computed from the well-known RTPS port mapping: * participant Id: 0 * well-known ports: DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS	0..1
<property>	A sequence of name-value string pairs that allows configuring the underlying transport instance. **Example:** `<property><value><element><name>dds.transport.UDPv4.builtin.allow_interfaces_list</name><value>eth0</value></element></value></property>` **Note:** When reusing a transport instance, if you specify the same property twice, *Cloud Discovery Service* will log a warning.	0..1

Cloud Discovery Service comes with the following preconfigured transport instances, which you can use and configure directly.

Note: You can override any of the preset transport properties. In such a case, *Cloud Discovery Service* will log a warning.

Table 1.12: Available Transports

Alias	Description	Prefix
`udpv4` or `builtin.udpv4`	Builtin implementation of UDPv4. • **Class ID**: NDDS_TRANSPORT_CLASSID_UDPv4 • **Reuse**: Yes	`dds.transport.UDPv4.builtin`
`udpv4_wan` or `builtin.udpv4_wan`	Implementation of UDPv4 for WAN networks. • **Class ID**: NDDS_TRANSPORT_CLASSID_UDPv4_WAN • **Reuse**: No **Note:** Your library path requires libraries from *RTI Real-Time WAN Transport* Support (`nddsrwt`).	`dds.transport.UDPv4_WAN.builtin`
`udpv6` or `builtin.udpv6`	Builtin implementation of UDPv6. • **Class ID**: NDDS_TRANSPORT_CLASSID_UDPv6 • **Reuse**: Yes	`dds.transport.UDPv6.builtin`
`tcpv4_lan`, `tcpv4_wan`, `tlsv4_lan` or `tlsv4_wan`	Implementation of TCPv4 for LAN and WAN networks. • **Class ID**: See **parent.classid** in Table 1.13 • **Reuse**: No **Note:** Your library path requires libraries from RTI TCP Support (`nddstransporttcp`) and additionally, if enabling TLS, RTI TLS Support (`nddstls`) and OpenSSL.	`dds.transport.cds.tcp1`

Preregistered UDP Transports

Cloud Discovery Service registers an instance for the following UDP transports:

- UDPv4
- UDPv4 WAN or the *RTI Real-Time WAN Transport*
- UDPv6

There are no preset properties for any of these instances.

See the following links for properties for each of these UDP transports respectively:

- Connext UDPv4 configuration

- Connext Real-Time WAN Transport configuration
- Connext UDPv6 configuration

Preregistered TCP Transport

Cloud Discovery Service registers an instance of the RTI TCP transport. Table 1.13 shows a list of preset properties.

See RTI TCP Transport configuration for a list of available properties.

Table 1.13: TCP Transport preset properties

Property name (prefix with `dds.transport.cds.tcp1`)	Property value
`library`	`nddstransporttcp`
`create_function`	`NDDS_Transport_TCPv4_create`
`server_bind_port`	The value of the corresponding `<receive_port>`
`parent.classid`	• **`NDDS_TRANSPORT_CLASSID_TCPV4_LAN`** – For alias `tcpv4_lan` • **`NDDS_TRANSPORT_CLASSID_TCPV4_WAN`** – For alias `tcpv4_wan` • **`NDDS_TRANSPORT_CLASSID_TLSV4_LAN`** – For alias `tlsv4_lan` • **`NDDS_TRANSPORT_CLASSID_TLSV4_WAN`** – For alias `tlsv4_wan`

Example: Reusing UDP Transport Instance for Multiple Receive Resources

```
<transport>
    <element>
        <alias>udpv4</alias>
        <receive_port>7400</receive_port>
    </element>
    <element>
        <alias>udpv4</alias>
        <receive_port>7500</receive_port>
    </element>
</transport>
```

Example: A Receive Resource for Each UDP and TCP Transport

This example shows how to specify different receive resources from different transport instances. Additionally, it shows how to extend the behavior of the preregistered TCP transport through the specification of additional transport properties using the transport prefix `dds.transport.cds.tcp1`.

```
<transport>
    <element>
        <alias>builtin.UDPv4</alias>
        <receive_port>7400</receive_port>
    </element>
    <element>
        <alias>tcpv4_wan</alias>
        <receive_port>8400</receive_port>
        <property>
            <element>
                <name>dds.transport.cds.tcp1.tcp1.public_address</name>
                <value>35.6.9.10</value>
            </element>
        </property>
    </element>
</transport>
```

Security

The `<security>` element allows you to enable and configure the security features provided by the *RTI Lightweight Security Plugins*. For further details, see *Security*.

Table 1.14 shows the description of this element.

Table 1.14: Security Tag

Tags within <security>	Description	Multiplicity

<property>	A sequence of name-value string pairs that allows configuring security. These are the valid properties you can configure within this tag:
	
	Attention: The following properties are deprecated and should only be used to communicate with legacy systems:
	• com.rti.serv.secure.authentication.participant_discovery_protection_key
	• com.rti.serv.secure.cryptography.rtps_protection_key
	
	• com.rti.serv.secure.cryptography.rtps_protection_preshared_key
	• com.rti.serv.secure.cryptography.rtps_protection_preshared_key_algorithm
	• dds.participant.discovery_config.signature_validation_persistent_state_file
	For further details, see *Security*.
	Example:
	`<property>`
	`  <element>`
	`  <name>com.rti.serv.secure.authentication.rtps_protection_preshared_key</name>`
	`  <value>str:10:KeySeed</value>`
	`  </element>`
	`  <element>`
	`  <name>com.rti.serv.secure.authentication.rtps_protection_preshared_key_algorithm</name>`
	`  <value>AES256+GCM</value>`
	`  </element>`
	`  <element>`
	`  <name>dds.participant.discovery_config.signature_validation_persistent_state_file</name>`
		0..1

Protocol Mode

The `<protocol_mode>` element allows you to select the discovery protocol that *Cloud Discovery Service* should support. *DomainParticipants* using a different discovery protocol than the one used by *Cloud Discovery Service* will be completely ignored.

The valid values for `<protocol_mode>` are:

- **SPDP** - *Cloud Discovery Service* forwards the messages from *DomainParticipants* that use *Simple Participant Discovery Protocol*. **This is the default value.**
- **SPDP2** - *Cloud Discovery Service* forwards only the bootstrap messages from *DomainParticipants* that use *Simple Participant Discovery Protocol 2.0*. *Cloud Discovery Service* does not forward the configuration or liveliness messages, since these are exchanged peer-to-peer after the bootstrapping completes. *DomainParticipants* using *Simple Participant Discovery Protocol 2.0* do not send the configuration or liveliness messages to *Cloud Discovery Service*.

  For further details about the new *Simple Participant Discovery Protocol 2.0*, refer to Simple Participant Discovery 2.0.

Note: When you choose `SPDP2` for the `<protocol_mode>` in XML, you need to ensure that the *DomainParticipants* are configured to use *Simple Participant Discovery Protocol 2.0*. Here is the snippet that you need to use in the XML QoS configuration file:

```
<domain_participant_qos>
    <discovery_config>
        <builtin_discovery_plugins>SDP2</builtin_discovery_
↪plugins>
    </discovery_config>
    . . .
</domain_participant_qos>
```

Forwarder

The `<forwarder>` element allows you to configure the attributes and behavior of the active component involved in the participant announcement forwarding process.

Figure 1.19 and Table 1.15 describe this element.

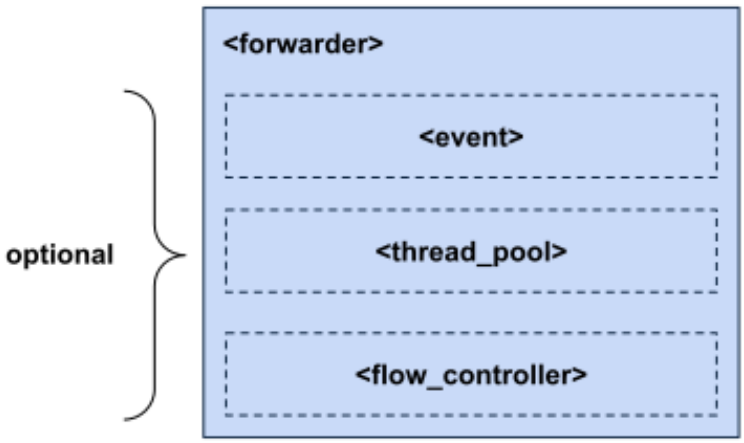

Figure 1.19: Forwarder Tag Structure

Table 1.15: Forwarder Tag

Tags within `<forwarder>`	Description	Multiplicity
`<event>`	Configures the behavior of the *Event Manager* resend events. Resend involves *Cloud Discovery Service* forwarding multiple copies of an incoming participant announcement. The `<event>` tag handles the count and interval between the announcement copies. It only applies to a *New* or *Change* announcement type. For more details refer to section *Forwarder*. **Note:** The `<refresh_period>` tag from previous versions has been replaced by the following new tags. Using this new mechanism allows for higher efficiency in bandwidth usage without sacrificing discovery speed. Elements: • **`<new_or_change_participant_announcements>`:** Sets the number of times a *New* or *Change* announcement is resent by the forwarder. This can be set to a value greater than 0 in networks where the probability of packet loss is high. Default: 5. • **`<min_new_or_change_participant_announcements_period>`:** Sets the minimum duration for the interval between successive resends from the forwarder. A random interval is generated by the forwarder to schedule a resend between `<min_new_or_change_participant_announcements_period>` and `<max_new_or_change_participant_announcements_period>`. Default: 1s. • **`<max_new_or_change_participant_announcements_period>`:** Sets the maximum duration for the interval between successive resends from the forwarder. A random interval is generated by the forwarder to schedule a resend between `<min_new_or_change_participant_announcements_period>`	0..1

Flow Controller

The `<flow_controller>` element allows you to configure the output traffic of *Cloud Discovery Service*. With the flow controller, you can limit the output capacity, assign more of the output capacity to certain participant announcements, and also throttle the output traffic.

Figure 1.20 and Table 1.16 describe this element.

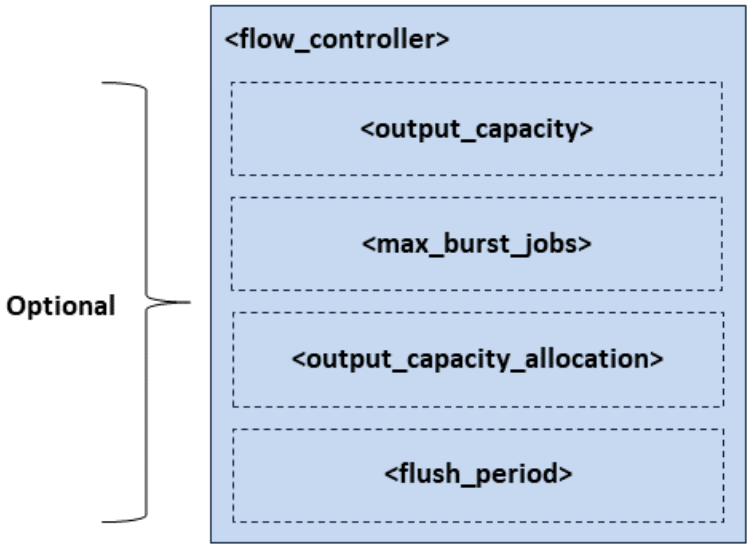

Figure 1.20: Flow Controller Tag Structure

Table 1.16: Flow Controller Tag

Tags within <flow_controller>	Description	Multiplicity
<output_capacity>	Specifies a limit for the output capacity in jobs per second. **Default:** LENGTH_UNLIMITED	0..1
<max_burst_jobs>	Maximum amount of announcement jobs that can be dispatched at once. **Default:** <output_capacity> * 1 second	0..1

<output_capacity_allocation>	Configures the output capacity distribution depending on the participant announcement class. See Table 1.17.
	Example
	`<output_capacity_allocation>`
	`  <new>50</new>`
	`  <update>30</update>`
	`  <refresh>20</refresh>`
	`</output_capacity_allocation>`
		0..1

<flush_period>	Configures the maximum period at which the forwarder will attempt to send pending announcements. **Default:** DURATION_INFINITE.
	Example
	`<flush_period>`
	`  <sec>1</sec>`
	`  <nanosec>0</nanosec>`
	`</flush_period>`
		0..1

Note: Configuring only one flow controller parameter in isolation may result in inaccurate performance of the forwarder. It is recommended to configure all the flow controller parameters, to guarantee the expected behavior.

The <output_capacity_allocation> allows you to distribute the output capacity to each participant announcement class. Each class element is an integer that represents the percentage of the total output capacity used to forward announcements of such class.

Table 1.17 describes this element.

Table 1.17: Output Capacity Allocation Tag

Tags within <output_capacity_allocation>	Description	Multiplicity
<new>	Percentage of the maximum output capacity dedicated to the **new** participant announcement class. **Default:** 40	0..1
<update>	Percentage of the maximum output capacity dedicated to the **update** participant announcement class. **Default:** 30	0..1
<refresh>	Percentage of the maximum output capacity dedicated to the **refresh** participant announcement class. **Default:** 30	0..1

Note that the sum of the percentages from the three classes can never be greater than **100**. Otherwise *Cloud Discovery Service* will log an error message and fail to start. Unless default values are used, if the allocation for one or more classes are not specified, *Cloud Discovery Service* will equally split the remaining of the output capacity among them.

Example: Flow Controller

This example shows how to set up a flow controller with a maximum output capacity of 5000 jobs/s, of which half is reserved only for new participant announcements, and the other half is equally distributed among the update and refresh classes (25/25).

```
<flow_controller>
    <output_capacity>5000</output_capacity>
    <output_capacity_allocation>
        <new>50</new>
    </output_capacity_allocation>
</flow_controller>
```

Database

The `<database>` element allows you to configure the behavior of the internal database that contains the information that represents the discovery state of the system.

Figure 1.21 and Table 1.18 describe this element.

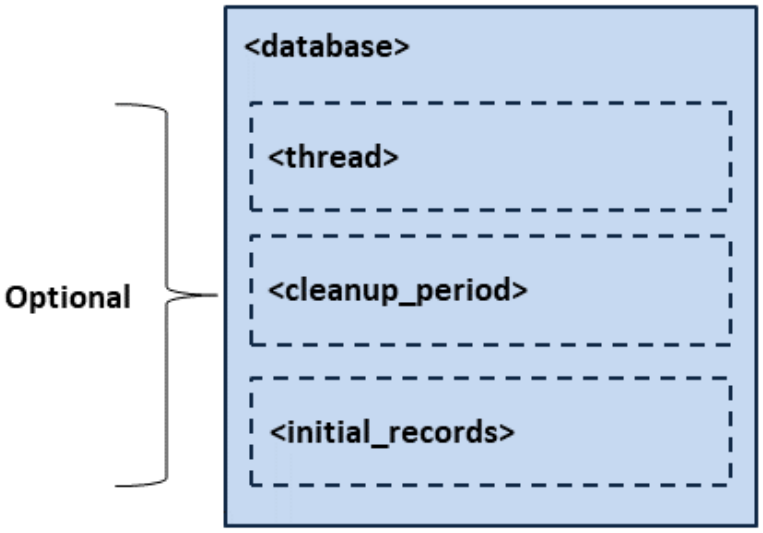

Figure 1.21: Database Tag Structure

Table 1.18: Database Tag

Relevant tags within <database>	Description	Multiplicity
<thread>	Database thread settings. See **thread** in DatabaseQosPolicy.	0..1
<cleanup_period>	The period at which the service database thread wakes up to clean up expired information. See **cleanup_period** in DatabaseQosPolicy. **Example** <cleanup_period> <sec>1</sec> <nanosec>0</nanosec> </cleanup_period>	0..1
<initial_records>	The initial number of total records. See **initial_records** in DatabaseQosPolicy.	0..1

Resource Limits

The <resource_limits> element allows you to specify upper limits of memory consumption. In general, *Cloud Discovery Service* incorporates mechanisms to clean up unused memory and maintain the execution within bounds when possible. Nevertheless, you may need to tune the memory usage if you have special memory requirements.

Figure 1.22 and Table 1.19 describe this element.

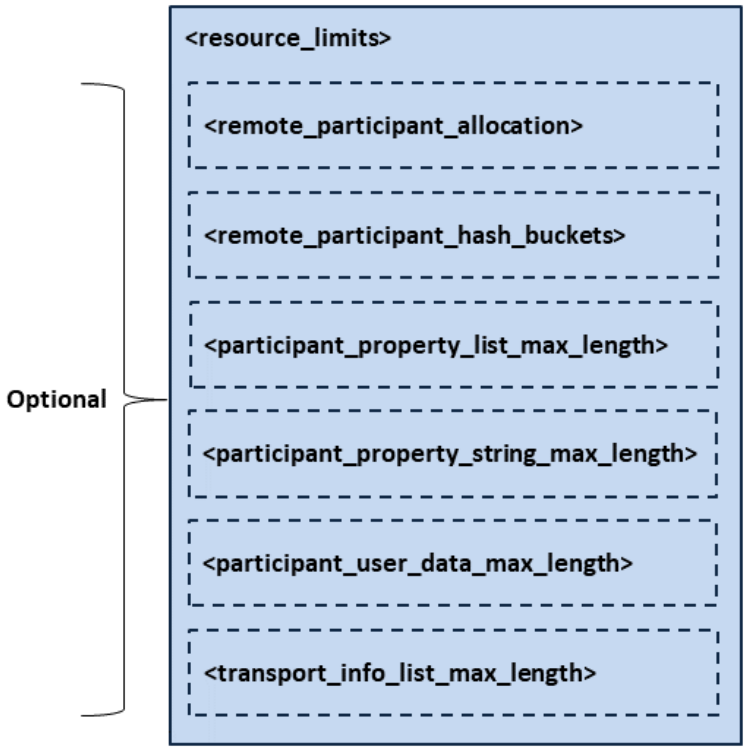

Figure 1.22: Resource Limits Tag Structure

Table 1.19: Resource Limits Tag

Tags within <resource_limits>	Description	Multiplicity
<remote_participant_allocation>	Allocation settings applied to remote DomainParticipants. See **remote_participant_allocation** in DomainParticipantResourceLimitsQosPolicy.	0..1
<remote_participant_hash_buckets>	Number of hash buckets for remote DomainParticipants. See **remote_participant_hash_buckets** in DomainParticipantResourceLimitsQosPolicy.	0..1
<participant_property_list_max_length>	Maximum number of properties associated with the DomainParticipant. See **participant_property_list_max_length** in DomainParticipantResourceLimitsQosPolicy.	0..1
<participant_property_string_max_length>	Maximum string length of the properties associated with the DomainParticipant. See **participant_property_string_max_length** in DomainParticipantResourceLimitsQosPolicy.	0..1
<participant_user_data_max_length>	Maximum length of user data to send and receive in a participant announcement. See **participant_user_data_max_length** in DomainParticipantResourceLimitsQosPolicy.	0..1
<transport_info_list_max_length>	Maximum number of installed transports to send and receive information about a participant announcement. See **transport_info_list_max_length** in DomainParticipantResourceLimitsQosPolicy.	0..1

Enabling Distributed Logger

Distributed Logger is included in *Connext* but it is not supported on all platforms; see the *RTI Connext* Core Libraries Platform Notes for the set of platforms that support *Distributed Logger*.

When you enable *Distributed Logger*, the *Cloud Discovery Service* will publish its log messages to *Connext*. Then you can use *RTI Admin Console* to visualize the log message data. Since the data is provided in a topic, you can also use *rtiddsspy* or even write your own visualization tool.

To enable *Distributed Logger*, use the tag <distributed_logger> within <adminstration>. For example:

```
<cloud_discovery_service name="MyCDS">
      <administration>
            ...
            <distributed_logger>
                <enabled>true</enabled>
            </distributed_logger>
        </administration>
        ...
</cloud_discovery_service>
```

For more details, see Enabling Distributed Logger in RTI Services, in the *RTI Connext* Core Libraries User's

Manual.

1.6.3 Builtin Configuration

Cloud Discovery Service comes pre-loaded with a builtin configuration, which is selected on startup if no configuration name is specified. This builtin configuration is equivalent to the following:

```
<cloud_discovery_service name="rti.cds.builtin.config.default">
    <transport>
        <element>
            <alias>builtin.udpv4</alias>
            <receive_port>7400</receive_port>
        </element>
    </transport>
    <domain_list>
        <allow_domain_id>DOMAIN_LIST_ALL</allow_domain_id>
    </domain_list>
    <forwarder>
        <thread_pool>
            <size>2</size>
        </thread_pool>
    </forwarder>
    <database>
        <cleanup_period>
            <sec>50</sec>
        </cleanup_period>
    </database>
</cloud_discovery_service>
```

The builtin configuration has the name `rti.cds.builtin.config.default`, which is reserved; no additional configurations can have this name.

1.6.4 Overriding XML Settings

Cloud Discovery Service allows you to override certain XML settings through the use of the command-line options. When these options are explicitly specified, their values will override the values of the equivalent XML elements.

See *Command-Line Options* for a list of the available options that can override XML settings.

1.7 Remote Administration

A control client (such as *RTI Admin Console*) can use this interface to remotely control *Cloud Discovery Service*.

Note: *Cloud Discovery Service* remote administration is based on the *RTI Remote Administration Platform* described in *Remote Administration Platform*. Please refer to that manual for a detailed discussion on the workings of remote administration in *Cloud Discovery Service*.

Below you will find an API reference for all the supported operations.

1.7.1 Enabling Remote Administration

By default, remote administration is disabled in *Cloud Discovery Service*.

To enable remote administration you can use the `<administration>` XML tag (see *Administration*) or the `-remoteAdministrationDomainId` command-line parameter (see *Command-Line Options*). Both of these methods enable remote administration and set the domain ID for remote communication.

1.7.2 Available Service Resources

Table 1.20 lists the public resources specific to *Cloud Discovery Service*. Each resource identifier is expressed as a hierarchical sequence of identifiers, including parent and target resources. (See *Resource Identifiers* for details.)

In the table below, the element `(cds)` refers to the name of an entity of the corresponding class as specified in the configuration in the `name` attribute. For example, in the following configuration:

```
<cloud_discovery_service name="MyCDS">...</cloud_discovery_service>
```

The resource identifier is:

```
/cloud_discovery_services/MyCDS
```

In the table below, the resource identifier is written as /cloud_discovery_services/(cds), where (cds) is the service name. This nomenclature is used in the table to give you an idea of the structure of the resource identifiers. For actual example resource identifier names, see the example section that follows.

Table 1.20: Resources and their identifiers in *Cloud Discovery Service*

Resource	Resource Identifier
Cloud Discovery Service	/cloud_discovery_services/(cds)
Database	/cloud_discovery_services/(cds)/database/[subset_info] where `[subset_info]` represents a specific smaller piece of the stored discovery information. The current available subsets are: • `locators`: List of original and resolved locators for a specified participant.

Example

This example shows you how to address a resource of each possible resource class in *Cloud Discovery Service*.

Cloud Discovery Service

Entity with name "MyCDS":

```
<cloud_discovery_service name="MyCDS">...</cloud_discovery_service>
```

Resource identifier:

```
/cloud_discovery_services/MyCDS
```

1.7.3 Remote API Overview

Table 1.21: Remote Interface Overview

Resource	Operation	Description
CloudDiscoveryService	*DELETE /cloud_discovery_services/(cds)*	Shuts down a running *Cloud Discovery Service* instance.
	UPDATE /cloud_discovery_services/(cds)/state	Sets a *Cloud Discovery Service* state.
	GET /cloud_discovery_services/(cds)/state	Gets a *Cloud Discovery Service* state.
	GET /cloud_discovery_services/(cds)/database/locators	Gets the list of original and resolved locators for a concrete participant.

1.7.4 Cloud Discovery Service

DELETE /cloud_discovery_services/(cds)

Operation `shutdown`

This operation will cause *Cloud Discovery Service* to shutdown.

UPDATE /cloud_discovery_services/(cds)/state

Operation `set_state`

Sets the state of a *Cloud Discovery Service* object. The action is parameterized on octet_body, which could have the following values:

See *Set Resource State*.

Valid requested states:

- `STARTED`
- `STOPPED`
- `ENABLED`
- `DISABLED`

- Example

  To stop an instance of *Cloud Discovery Service* with the name "MyCDS":

Request Field	Value
`action`	UPDATE
`resource_identifier`	/cloud_discovery_services/MyCDS/state
`octet_body`	`to_cdr_buffer(RTI::Service::EntityStateKind::STOPPED)` For example code refer to *Example: Controlling services remotely from a Connext Application*.

GET /cloud_discovery_services/(cds)/state

Operation `get_state`

Gets the state of a *Cloud Discovery Service* object. The value is contained in the reply octet_body, which could have the following values:

See *Get Resource State*

Valid reply states:

- `STARTED`
- `STOPPED`

- Example

To inspect the current state of an instance of *Cloud Discovery Service* with the name "MyCDS":

Request Field	Value
`action`	GET
`resource_identifier`	/cloud_discovery_services/MyCDS/state

Reply Field	Value
`octet_body`	`from_cdr_buffer(RTI::Service::Admin::CommandReply::octet_body)` For example code refer to *Example: Controlling services remotely from a Connext Application*.

1.7.5 Database

`GET /cloud_discovery_services/(cds)/database/locators`
Operation `get_participant_locators`

Gets the list of resolved and original locators for a specified participant. The result is returned in the reply's string body as a JSON string with the following format:

```
{
  "locators": {
    "original":{
       "metatraffic":[
           "original_mt_locator1",
           "original_mt_locator1",
            ...
       ],
       "default":[
           "original_def_locator1",
           "original_def_locator2",
           ...
       ]
    },
    "resolved":{
       "metatraffic":[
           "resolved_mt_locator1",
           "resolved_mt_locator2",
           ...
       ],
       "default":[
           "resolved_def_locator1",
           "resolved_def_locator2",
           ...
       ]
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

The original locator set represent the locators contained in the participant announcement as it is received. The resolved locator set represent the locators *Cloud Discovery Service* updates or extend to contain additional information necessary for the peer participants. This is for example important when using the |*RTI_RWT*|.

The participant is provided in the string body as the string representation of the GUID with format:

```
0x0101C372,0x17D4CED6,0xEA7B3DA5:0x000001C1
```

- Example

  To locator information for a specific participant existing in the database of a *Cloud Discovery Service* instance with the name "MyCDS":

Request Field	Value
`action`	GET
`resource_identifier`	/cloud_discovery_services/MyCDS/database/locators
`string_body`	`0x0101C372,0x17D4CED6,` `↪0xEA7B3DA5:0x000001C1`

Reply Field	Value

string_body	{
	  "locators": {
	  "original":{
	  "metatraffic":[
	  "udpv4_wan://f=1,u={BD,73,FC,
	↪9E,D5,00,AB,6D,40},P=10.0.2.15:0:32410"
	  ],
	  "default":[
	  "udpv4_wan://f=1,u={BD,73,FC,
	↪9E,D5,00,AB,6D,40},P=10.0.2.15:0:32411"
	  ]
	  },
	  "resolved":{
	  "metatraffic":[
	  "udpv4_wan://f=7,u={BD,73,
	↪FC,9E,D5,00,AB,6D,40},P=188.78.49.
	↪211:56532:32410"
	  ],
	  "default":[
	  "udpv4_wan://f=7,u={BD,73,
	↪FC,9E,D5,00,AB,6D,40},P=188.78.49.
	↪211:56532:32410"
	  ]
	  }
	  }
	}

1.8 Monitoring

This section provides documentation on *Cloud Discovery Service* remote monitoring.

Note: *Cloud Discovery Service* monitoring is based on the *Monitoring Distribution Platform* described in *Monitoring Distribution Platform*. We recommend that you read *Monitoring Distribution Platform* before using *Cloud Discovery Service* monitoring.

1.8.1 Overview

Enabling Service Monitoring

By default, monitoring is disabled in *Cloud Discovery Service*. To enable monitoring you can use the `<monitoring>` tag (see *Monitoring*) or the `-remoteMonitoringDomainId` command-line parameter, which enables remote monitoring and sets the domain ID for data publication (see *Command-Line Options*).

Monitoring Types

The available *Keyed Resource* classes and their types that can be present in the distribution monitoring topics are listed in Table 1.22. The complete type relationship is shown in Figure 1.23.

Table 1.22: *Cloud Discovery Service Keyed Resources*

Keyed Resource Class	Config	Event	Periodic
Service	`ServiceConfig`	`ServiceEvent`	`ServicePeriodic`
Forwarder	`ForwarderConfig`	`ForwarderEvent`	`ForwarderPeriodic`
Sender	`SenderConfig`	`SenderEvent`	`SenderPeriodic`
Receiver	`ReceiverConfig`	`ReceiverEvent`	`ReceiverPeriodic`
Database	`DatabaseConfig`	`DatabaseEvent`	`DatabasePeriodic`

All the type definitions for *Cloud Discovery Service* monitoring information are in `[NDDSHOME]/resource/idl/ServiceCommon.idl` and `[NDDSHOME]/resource/idl/CdsMonitoring.idl`.

Cloud Discovery Service creates a *DataWriter* for each distribution *Topic*. All *DataWriters* are created from a single *Publisher*, which is created from a dedicated *DomainParticipant*. See *Cloud Discovery Service* for details on configuring the QoS for these entities.

1.8.2 Monitoring Metrics Reference

This section provides a reference to all the monitoring metrics *Cloud Discovery Service* distributes, organized by service resource class.

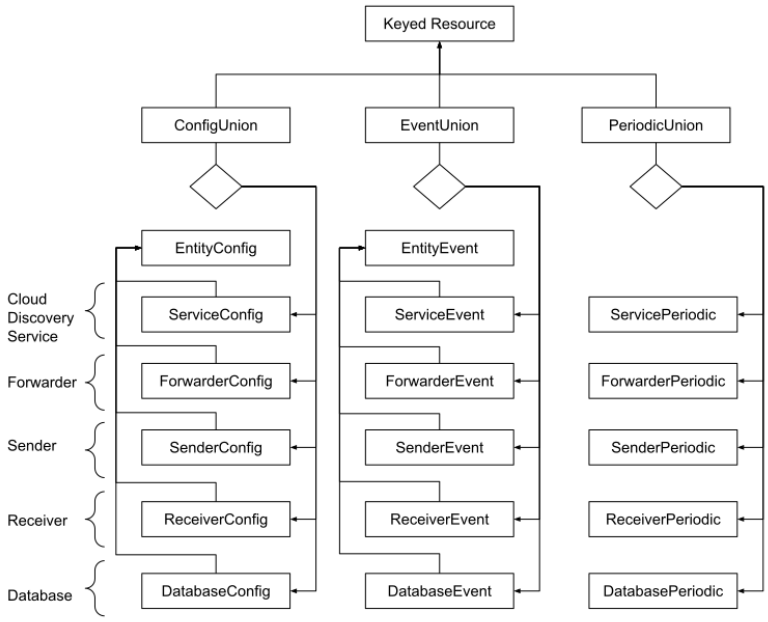

Figure 1.23: Keyed Resource Types for *Cloud Discovery Service* monitoring

Service

Listing 1.1: *Cloud Discovery Service* Types

```
@mutable @nested
struct ServiceConfig : Service::Monitoring::EntityConfig {
    Service::BoundedString application_name;
    Service::Monitoring::ResourceGuid application_guid;
    @optional Service::Monitoring::HostConfig host;
    @optional Service::Monitoring::ProcessConfig process;
};

@mutable @nested
struct ServiceEvent : Service::Monitoring::EntityEvent {
};

@mutable @nested
struct ServicePeriodic {
    @optional Service::Monitoring::HostPeriodic host;
    @optional Service::Monitoring::ProcessPeriodic process;
};
```

Table 1.23: `ServiceConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 1.52.
`application_name`	Name of the *Cloud Discovery Service* instance. The application name is provided through: • `appName` command-line option when run as executable. • `ServiceProperty::service_name` field when run as a library.
`application_guid`	GUID of the *Cloud Discovery Service* instance. Unique across all service instances.
`host`	See Table 1.48.
`process`	See Table 1.50.

Table 1.24: `ServiceEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 1.53.

Table 1.25: `ServicePeriodic`

Field Name	Description
`host`	See Table 1.49.
`process`	See Table 1.51.

Forwarder

Listing 1.2: Forwarder Types

```
@mutable @nested
struct ForwarderConfig : Service::Monitoring::EntityConfig {
};

@mutable @nested
struct ForwarderEvent : Service::Monitoring::EntityEvent {
};

@mutable @nested
struct ForwarderPeriodic {
    @optional Service::Monitoring::NetworkPerformance
            new_participant_announcements;
    @optional Service::Monitoring::NetworkPerformance
            repeat_participant_announcements;
    @optional Service::Monitoring::NetworkPerformance
            change_participant_announcements;
    @optional Service::Monitoring::ThreadPoolPeriodic thread_pool;
};
```

Table 1.26: `ForwarderConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 1.52.

Table 1.27: `ForwarderEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 1.53.

Table 1.28: `ForwarderPeriodic`

Field Name	Description
`new_participant_announcements`	Provides network performance metrics as an aggregation of the same metrics across the contained *Sender* and *Receiver*. See *Network Performance Metrics*.
`repeat_participant_announcements`	Provides network performance metrics as an aggregation of the same metrics across the contained *Sender* and *Receiver*. See *Network Performance Metrics*.
`change_participant_announcements`	Provides network performance metrics as an aggregation of the same metrics across the contained *Sender* and *Receiver*. See *Network Performance Metrics*.
`thread_pool`	Sequence of `ThreadPeriodic` objects, one for each thread of the *Forwarder*'s thread pool. See Table 1.55.

Sender

Listing 1.3: Sender Types

```
@mutable @nested
struct SenderConfig : Service::Monitoring::EntityConfig {
};

@mutable @nested
struct SenderEvent : Service::Monitoring::EntityEvent {
};

@mutable @nested
struct SenderPeriodic {
    @optional Service::Monitoring::NetworkPerformance
            new_participant_announcements;
    @optional Service::Monitoring::NetworkPerformance
            repeat_participant_announcements;
    @optional Service::Monitoring::NetworkPerformance
            change_participant_announcements;
};
```

Table 1.29: `SenderConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 1.52.

Table 1.30: `SenderEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 1.53.

Table 1.31: `SenderPeriodic`

Field Name	Description
`new_participant_announcements`	Provides network performance metrics for the `new` participant announcements class dispatched by the *Sender*. See *Network Performance Metrics*.
`repeat_participant_announcements`	Provides network performance metrics for the `repeat` participant announcements class dispatched by the *Sender*. See *Network Performance Metrics*.
`change_participant_announcements`	Provides network performance metrics for the `change` participant announcements class dispatched by the *Sender*. See *Network Performance Metrics*.

Receiver

Listing 1.4: Receiver Types

```
@mutable @nested
struct ReceiverConfig : Service::Monitoring::EntityConfig {
};

@mutable @nested
struct ReceiverEvent : Service::Monitoring::EntityEvent {
};

@mutable @nested
struct ReceiverPeriodic {
    @optional Service::Monitoring::NetworkPerformance
            new_participant_announcements;
    @optional Service::Monitoring::NetworkPerformance
            repeat_participant_announcements;
    @optional Service::Monitoring::NetworkPerformance
            change_participant_announcements;
};
```

Table 1.32: `ReceiverConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 1.52.

Table 1.33: `ReceiverEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 1.53.

Table 1.34: `ReceiverPeriodic`

Field Name	Description
`new_participant_announcements`	Provides network performance metrics for the `new` participant announcements class received by the *Receiver*. See *Network Performance Metrics*.
`repeat_participant_announcements`	Provides network performance metrics for the `repeat` participant announcements class received by the *Receiver*. See *Network Performance Metrics*.
`change_participant_announcements`	Provides network performance metrics for the `change` participant announcements class received by the *Receiver*. See *Network Performance Metrics*.

Database

Listing 1.5: Database Types

```
@mutable @nested
struct DatabaseConfig : Service::Monitoring::EntityConfig {
};

@mutable @nested
struct DatabaseEvent : Service::Monitoring::EntityEvent {
};

@mutable @nested
struct DatabasePeriodic {
    @optional Service::Monitoring::ThreadPoolPeriodic thread_pool;
};
```

Table 1.35: `DatabaseConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 1.52.

Table 1.36: `DatabaseEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 1.53.

Table 1.37: `DatabasePeriodic`

Field Name	Description
`thread_pool`	Sequence of `ThreadPeriodic` objects, one for each thread of the *Database*'s thread pool. See Table 1.55.

1.9 Security

You can use symmetric cryptography using pre-shared keys to protect the communication between *Cloud Discovery Service* and the user's *DomainParticipants*, as described in Security Considerations when Using Cloud Discovery Service.

Cloud Discovery Service uses the *RTI Lightweight Security Plugins* to protect the integrity and/or confidentiality of RTPS messages. By operating at the RTPS level, the protection is applied to all messages exchanged between the *DomainParticipants* and *Cloud Discovery Service*. These include the participant announcements and the `BINDING_PING` messages when using the *Real-Time WAN Transport*.

Attention: In the prior releases, *Cloud Discovery Service* and *Real-Time WAN Transport* can be protected with <<deprecated>> `com.rti.serv.secure.cryptography.rtps_protection_key` and <<deprecated>> `com.rti.serv.secure.authentication.participant_discovery_protection_key` properties. They are still functional and intended for communicating with legacy systems only. This functionality will be removed in the future and is not suitable for new deployments. For detailed description about legacy properties, please refer to *Connext DDS Secure* and *Cloud Discovery Service* 6.1.2 documentation.

1.9.1 Configuration

To configure security in *Cloud Discovery Service*, you can set the following properties:

- `com.rti.serv.secure.cryptography.rtps_protection_preshared_key` - This is the key value used by the *RTI Lightweight Security Plugins* inside *Cloud Discovery Service* to protect the integrity and/or confidentiality of RTPS messages. The value should be the same on all the *DomainParticipants* and *Cloud Discovery Service*. For further details, see Configuring the Lightweight Security Plugins.
- `com.rti.serv.secure.cryptography.rtps_protection_preshared_key_algorithm` - This is the Pre-Shared Key Protection algorithm used by *DomainParticipants* and *Cloud Discovery Service*. The value should be the same on all the *DomainParticipants* and *Cloud Discovery Service*. For further details, see Configuring the Lightweight Security Plugins.
- `dds.participant.discovery_config.signature_validation_persistent_state_file` - This property allows protection against a *Cloud Discovery Service* participant announcement replay attack. It is useful when a running *Cloud Discovery Service* instance configured with the above security properties could be restarted. For further details, see Protection Against a Cloud Discovery Service Participant Announcement Replay Attack.

In *Cloud Discovery Service*, set the above properties by updating the `<property>` tag inside the `<security>` tag (see *Configuration for Security*).

1.9.2 Pre-Shared Key Mutability

For a running *Cloud Discovery Service* instance, the *RTI Lightweight Security Plugins* also supports mutability for the `com.rti.serv.secure.cryptography.rtps_protection_preshared_key` property. You are allowed to change the pre-shared key dynamically. The reasons for changing the key could be overuse, leaks or compromise, or proactive prevention of these security problems.

To change the pre-shared key, leverage the *Cloud Discovery Service* Library API:

- C Library API - RTI_CDS_Service_update_rtps_protection_preshared_key.
- C++11 Library API - update_rtps_protection_preshared_key.

1.10 Tutorials

Note: The commands shown in these examples are for the `x64Linux3gcc5.4.0` architecture. For simplicity, all the examples assume that all the applications run on the same host machine and the environment variable `NDDSHOME` is set to point to your *RTI Connext* installation.

Note: Some of the examples require OpenSSL libraries and you must ensure they are present in your library path before starting the applications. Example commands assume the existence of the environment variable `OPENSSLHOMELIB` set to the directory that contains the OpenSSL libraries.

If you use the OpenSSL libraries distributed by RTI, the libraries will have the following locations:

- On Linux and macOS systems: `NDDSHOME`/third_party/openssl-<version>/<architecture>/release/lib
- On Windows systems: `NDDSHOME`\third_party\openssl-<version>\<architecture>\release\lib

---

1.10.1 Example: Using a Builtin UDP Transport

This example illustrates the basic functionality of *Cloud Discovery Service*. It shows how *Connext DDS* applications discover each other through an instance of *Cloud Discovery Service*.

Setup

The first step is to have two *Connext* applications that can talk to each other. For that, you can use an IDL file of your choice and run *rtiddsgen* to generate the example. For instance, you can create a file named `CDSHelloWorld.idl` with the following content:

```
struct CDSHelloWorld {
    long count;
};
```

Generate the publisher and subscriber examples by running the following command (you can replace `C++11` with any other supported language of your choice):

```
$NDDSHOME/bin/rtiddsgen -language C++11 -example x64Linux3gcc5.4.0 \
    -ppDisable CDSHelloWorld.idl
```

To compile the generated code, run:

```
make -f makefile_CDSHelloWorld_x64Linux3gcc5.4.0
```

You should have two executables, `CDSHelloWorld_publisher` and `CDSHelloWorld_subscriber`. Now you can run the examples to make sure everything is fine. You need to run the examples from the directory that contains the generated `USER_QOS_PROFILES.xml`.

From one terminal run:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_publisher -s 30
Writing CDSHelloWorld, count 0
Writing CDSHelloWorld, count 1
...
```

From a different terminal, run:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber
CDSHelloWorld subscriber sleeping for 4 sec...
[count: 3]
CDSHelloWorld subscriber sleeping for 4 sec...
[count: 4]
...
```

You should see that the subscriber application receives the samples generated by the publisher application. This occurs because by default, *DomainParticipants* enable multicast. That is, participant announcements are automatically sent to a multicast destination.

Disable Multicast and Shared Memory, and unset default Initial Peers

Stop the applications and disable multicast and the shared memory transport. Then unset the default initial peers, which enable applications on the same host to discover each other. To do that, open the `USER_QOS_PROFILES.xml` file and add the following snippet to `<domain_participant_qos>`.

```
<domain_participant_qos>
. . .

    <!-- Only enable the UDPv4 transport -->
    <transport_builtin>
        <mask>UDPv4</mask>
    </transport_builtin>
    <discovery>
        <!-- This effectively disables multicast -->
        <multicast_receive_addresses />
        <!-- This eliminates the default initial peers -->
        <initial_peers />
    </discovery>
</domain_participant_qos>
```

Now if you run the publisher and subscriber applications again, notice that the subscriber does not receive any samples:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber
CDSHelloWorld subscriber sleeping for 4 sec...
CDSHelloWorld subscriber sleeping for 4 sec...
CDSHelloWorld subscriber sleeping for 4 sec...
...
```

You can stop the applications now.

Cloud Discovery Service in Action

To make the applications communicate, we need to run *Cloud Discovery Service*. From a terminal, run the following command:

```
$NDDSHOME/bin/rticlouddiscoveryservice -verbosity LOCAL
```

This command will run *Cloud Discovery Service* with the builtin configuration that listens on port 7400 over the UDP transport.

Re-run the publisher and subscriber applications, but this time set the initial peers to talk to *Cloud Discovery Service* (see *RTPS Peer Descriptor*). To do that, open the `USER_QOS_PROFILES.xml` file and add the following snippet to `<domain_participant_qos>`.

```
<domain_participant_qos>
. . .

    <discovery>
```

(continues on next page)

(continued from previous page)

```
        . . .
        <initial_peers>
            <element>rtps@localhost:7400</element>
        </initial_peers>
    </discovery>
</domain_participant_qos>
```

Now run the applications and verify that the subscriber receives samples. In addition, you can start more publishers and subscribers, then see how they all discover each other and communicate. You can also stop *Cloud Discovery Service* and see that the applications **continue communicating** because once Discovery has completed, *Cloud Discovery Service* is no longer needed.

1.10.2 Example: Using a Custom Listening Port

This example extends *Example: Using a Builtin UDP Transport* to run *Cloud Discovery Service* on a custom listening port over UDP. For that, you can run *Cloud Discovery Service* as follows:

```
$NDDSHOME/bin/rticlouddiscoveryservice -transport <port>
```

where `<port>` represents the port number you want to use. This example will use `10000`.

```
$NDDSHOME/bin/rticlouddiscoveryservice -transport 10000
```

To see which exact address *Cloud Discovery Service* is using to listen for participant announcements, run the service with `-verbosity LOCAL` and look for the line:

```
...
[/cloud_discovery_services/rti.cds.builtin.config.default/forwarder/
↪receiver|ENABLE] listening for announcements on:
{
  "locator":[
    "udpv4://172.17.0.2:10000"
  ]
}
...
```

Set the initial peers accordingly to indicate where *Cloud Discovery Service* is listening:

```
<domain_participant_qos>
. . .

    <discovery>
        . . .
        <initial_peers>
            <element>rtps@localhost:10000</element>
        </initial_peers>
    </discovery>
</domain_participant_qos>
```

1.10.3 Example: Using RTI TCP Transport

This example extends *Example: Using a Builtin UDP Transport* to use the RTI TCP transport for both discovery and user data. The example shows how to run *Cloud Discovery Service* using the preregistered instance of RTI TCP transport.

Note: To run this example, you need RTI TCP Transport, which is included with *Connext*.

Setup

The first step of this example is to configure the publisher and subscriber applications so they communicate over the RTI TCP transport in LAN mode. For that, you can include the following XML code within the `<domain_participant_qos>` tag of the file `USER_QOS_PROFILES.xml` generated by *rtiddsgen*.

```
<property>
    <value>
        <element>
            <name>dds.transport.load_plugins</name>
            <value>dds.transport.tcp.tcp1</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.library</name>
            <value>nddstransporttcp</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.create_function</name>
            <value>NDDS_Transport_TCPv4_create</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.server_bind_port</name>
            <value>$(BIND_PORT)</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.parent.classid</name>
            <value>NDDS_TRANSPORT_CLASSID_TCPV4_LAN</value>
        </element>
    </value>
</property>
```

The server bind port property value is obtained from the environment variable `$(BIND_PORT)`. This is a convenience for this example so you can reuse the same file to specify a different port for the publisher and subscriber applications, which is required when running on the same host machine.

Now you can run the applications. You need to run from the directory that contains the generated `USER_QOS_PROFILES.xml` you just modified.

From the terminal, run the Publisher application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0
export BIND_PORT=7401
export NDDS_DISCOVERY_PEERS=tcpv4_lan://localhost:7402

./objs/x64Linux3gcc5.4.0/CDSHelloWorld_publisher -s 30
```

Then run the Subscriber application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0
export BIND_PORT=7402
export NDDS_DISCOVERY_PEERS=tcpv4_lan://localhost:7401

./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber
```

You should see the subscriber receive samples. This is possible since in the initial peers list, you indicated how the applications can reach each other.

Cloud Discovery Service in Action

Now you will use *Cloud Discovery Service* to provide the discovery for the applications.

From a terminal, run the following:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0

$NDDSHOME/bin/rticlouddiscoveryservice -transport tcpv4_lan:7400
```

This will run *Cloud Discovery Service* with the builtin configuration and override the transport selection to use RTI TCP with bind port `7400`.

Re-run the publisher and subscriber application, but this time set the initial peers to talk to *Cloud Discovery Service* (see *RTPS Peer Descriptor*). For that, you can set the environment from the application terminal as follows:

For the Publisher application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0
export BIND_PORT=7401
export NDDS_DISCOVERY_PEERS=rtps@tcpv4_lan://localhost:7400
```

For the Subscriber application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0
export BIND_PORT=7402
export NDDS_DISCOVERY_PEERS=rtps@tcpv4_lan://localhost:7400
```

Now run the applications and verify that the subscriber receives samples. You can also stop *Cloud Discovery Service* and see how the applications **continue communicating**.

Configuration for TCP transport in WAN Mode using a public address

In this example, we have seen how to configure the TCP transport in the publisher and subscriber applications, and how to configure *Cloud Discovery Service* to operate in LAN mode. However, there can be situations where *Cloud Discovery Service* or the publisher and subscriber applications are located across LANs or firewalls. In such a use case, it would be typical to have *Cloud Discovery Service* expose a public IP address which can be used for asymmetric discovery by the publisher or subscriber application.

To apply this example in that context, you will need to make the following changes:

USER_QOS_PROFILES.xml

First, change the **classid** for the TCP transport. The publisher and subscriber also need a publicly reachable address when communicating with *Cloud Discovery Service* or other peers in the domain. To achieve both of these changes, add the following to configuration of the RTI TCP transport:

```
<property>
    <value>
        <element>
            <name>dds.transport.tcp.tcp1.parent.classid</name>
            <value>NDDS_TRANSPORT_CLASSID_TCPV4_WAN</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.public_address</name>
            <value>$(PUBLIC_ADDRESS)</value>
        </element>
    </value>
</property>
```

The `$(PUBLIC_ADDRESS)` environment variable allows us to use the same configuration file for the publisher and subscriber applications.

Cloud Discovery Service

Create a configuration file for *Cloud Discovery Service*, select the TCP transport for the WAN alias, and set the public address. This public address will be used by the publisher and the subscriber applications in their discovery peers.

```
<?xml version="1.0" ?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="../../resource/schema/rti_cloud_
↪discovery_service.xsd">
    <cloud_discovery_service name="ExampleTCPWAN">
        <annotation>
            <documentation>
                <![CDATA[
                Example that configures the pre-registered instance of the
                RTI TCP WAN transport with a public IP address.
```

(continues on next page)

(continued from previous page)

```
                ]]>
            </documentation>
        </annotation>
        <transport>
            <element>
                <alias>tcpv4_wan</alias>
                <receive_port>9000</receive_port>
                <property>
                    <element>
                        <name>dds.transport.cds.tcp1.public_address</name>
                        <value>35.6.9.10:4589</value>
                    </element>
                </property>
            </element>
        </transport>
    </cloud_discovery_service>
</dds>
```

You can download the configuration file from here: `rti_cds_example_tcp_wan.xml`

You can read more about the **public_address** property of the RTI TCP transport in RTI TCP Transport configuration. An important thing to note here is the format of the specified address. It follows the `[ip:port]` format, where the IP is the public address of the router that provides access to the WAN. The port is the router port that is used to reach the private `server_bind_port` inside the LAN from the outside. In this example, the private `server_bind_port` is `9000` and the external port is `4589`.

You should change these values as per your network configuration.

To use this configuration, from the terminal, run:

```
$NDDSHOME/bin/rticlouddiscoveryservice \
        -cfgFile <working_dir>/rti_cds_example_tcp_wan.xml -cfgName␣
↪ExampleTCPWAN
```

NDDS_DISCOVERY_PEERS

Run the publisher and subscriber applications exactly the way you did in *Setup*, but with one modification.

```
export NDDS_DISCOVERY_PEERS=rtps@tcpv4_wan://35.6.9.10:4589
```

Make sure you set the `PUBLIC_ADDRESS` environment variable for the publisher and subscriber applications.

```
# Replace the [ip:port] pair depending upon the mapping setup in your router
export PUBLIC_ADDRESS=10.20.30.40:7000
```

You will see that until *Cloud Discovery Service* is started, the publisher and subscriber applications won't discover each other. Once *Cloud Discovery Service* starts, these applications will complete discovery and start communicating with each other.

1.10.4 Example: Using RTI TCP Transport with RTI TLS Support

This example extends *Example: Using a Builtin UDP Transport* to enable TLS for secure communication. The example shows how to run *Cloud Discovery Service* using the preregistered transport instance of the RTI TCP transport with TLS enabled.

Note: To run this example, you need the RTI TCP Transport, which is shipped with *Connext*. Additionally, you will need to install the optional packages RTI TLS support and OpenSSL.

Setup

The first step in this example is to configure the publisher and subscriber applications to communicate over the RTI TCP transport in LAN mode with TLS enabled. For that, you can include the following XML code within the `<domain_participant_qos>` tag of the file `USER_QOS_PROFILES.xml` generated by *rtiddsgen*.

```
<property>
    <value>
        <element>
            <name>dds.transport.load_plugins</name>
            <value>dds.transport.tcp.tcp1</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.library</name>
            <value>nddstransporttcp</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.create_function</name>
            <value>NDDS_Transport_TCPv4_create</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.server_bind_port</name>
            <value>$(BIND_PORT)</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.parent.classid</name>
            <value>NDDS_TRANSPORT_CLASSID_TLSV4_LAN</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.tls.verify.ca_file</name>
            <value>$(PATH_TO_EXAMPLES)/dds_security/cert/tls_rsa01/ca/
↪rsa01RootCaCert.pem</value>
        </element>
        <element>
            <name>dds.transport.tcp.tcp1.tls.identity.certificate_chain_file</
↪name>
            <value>$(PATH_TO_EXAMPLES)/dds_security/cert/tls_rsa01/identities/
↪rsa01Peer01.pem</value>
        </element>
```

(continues on next page)

(continued from previous page)

```
    </value>
</property>
```

Note: This *Cloud Discovery Service* example reuses the security artifacts in other examples shipped with *Connext*. To identify what the value of the environment variable `$(PATH_TO_EXAMPLES)` should be, refer to *Paths Mentioned in Documentation*

The server bind port property value is obtained from the environment variable `$(BIND_PORT)`. This is a convenience for this example so you can reuse the same file to specify different ports for the publisher and subscriber applications, which is required when running on the same host machine.

Now you can run the applications. You need to run from the directory that contains the generated `USER_QOS_PROFILES.xml` that you just modified and the certificates.

Because TLS is enabled, you must ensure that the *RTI TLS Support* and OpenSSL libraries are present in your library path before starting the applications. See the *beginning of this section* for a note on how to locate the OpenSSL libraries.

From the terminal, run the Publisher application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0:$OPENSSLHOMELIB
export BIND_PORT=7401
export NDDS_DISCOVERY_PEERS=tlsv4_lan://localhost:7402

./objs/x64Linux3gcc5.4.0/CDSHelloWorld_publisher -s 30
```

Then run the Subscriber application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0:$OPENSSLHOMELIB
export BIND_PORT=7402
export NDDS_DISCOVERY_PEERS=tlsv4_lan://localhost:7401

./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber
```

You should see the subscriber receives samples. This is possible since in the initial peers list, you indicated how the applications can reach each other.

Cloud Discovery Service in Action

Now you will use *Cloud Discovery Service* to provide the discovery for the applications. In this case, you need to configure the transport with an instance of the RTI TCP transport with TLS enabled.

The configuration required for this example is shown below:

```
<?xml version="1.0"?>
<dds  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="../../resource/schema/rti_cloud_
↪discovery_service.xsd">
```

(continues on next page)

(continued from previous page)

```
    <cloud_discovery_service name="ExampleTls">
        <annotation>
            <documentation><![CDATA[
                Example that configures the pre-registered instance of the
                RTI TCP transport with TLS enabled.
                ]]>
            </documentation>
        </annotation>
        <transport>
            <element>
                <alias>tlsv4_lan</alias>
                <receive_port>7400</receive_port>
                <property>
                    <element>
                        <name>dds.transport.cds.tcp1.tls.verify.ca_file</name>
                        <value>$(PATH_TO_EXAMPLES)/dds_security/cert/tls_
↪rsa01/ca/rsa01RootCaCert.pem</value>
                    </element>
                    <element>
                        <name>dds.transport.cds.tcp1.tls.identity.certificate_
↪chain_file</name>
                        <value>$(PATH_TO_EXAMPLES)/dds_security/cert/tls_
↪rsa01/identities/rsa01Peer01.pem</value>
                    </element>
                </property>
            </element>
        </transport>
    </cloud_discovery_service>

</dds>
```

You can download the configuration file from here: `rti_cds_example_tls.xml`

From a terminal, run:

```
export LD_LIBRARY_PATH=$OPENSSLHOMELIB
$NDDSHOME/bin/rticlouddiscoveryservice \
        -cfgFile <working_dir>/rti_cds_example_tls.xml -cfgName ExampleTls
```

This command will run *Cloud Discovery Service* with the custom configuration for this example, located under the directory `<working_dir>`.

Re-run the publisher and subscriber applications, but this time set the initial peers to talk to *Cloud Discovery Service* (see *RTPS Peer Descriptor*). For that, you can set the environment from the application terminal as follows:

For the Publisher application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0:$OPENSSLHOMELIB
export BIND_PORT=7401
export NDDS_DISCOVERY_PEERS=rtps@tlsv4_lan://localhost:7400
```

For the Subscriber application:

```
export LD_LIBRARY_PATH=$NDDSHOME/lib/x64Linux3gcc5.4.0:$OPENSSLHOMELIB
export BIND_PORT=7402
export NDDS_DISCOVERY_PEERS=rtps@tlsv4_lan://localhost:7400
```

Now run the applications and verify that the subscriber receives samples. You can also stop *Cloud Discovery Service* and see how the applications **continue communicating**.

1.10.5 Example: Using *RTI Real-Time WAN Transport*

This example extends the example in *Example: Using a Builtin UDP Transport* to use *RTI RTI Real-Time WAN Transport* to assist in the communication when the publisher and subscriber applications sit behind Cone NATs. This example shows how to run *Cloud Discovery Service* with the pre-registered transport instance of *RTI Real-Time WAN Transport*.

Note: To run this example, you will need an additional license to run *RTI Real-Time WAN Transport*. Contact support@rti.com or your sales representative for further information.

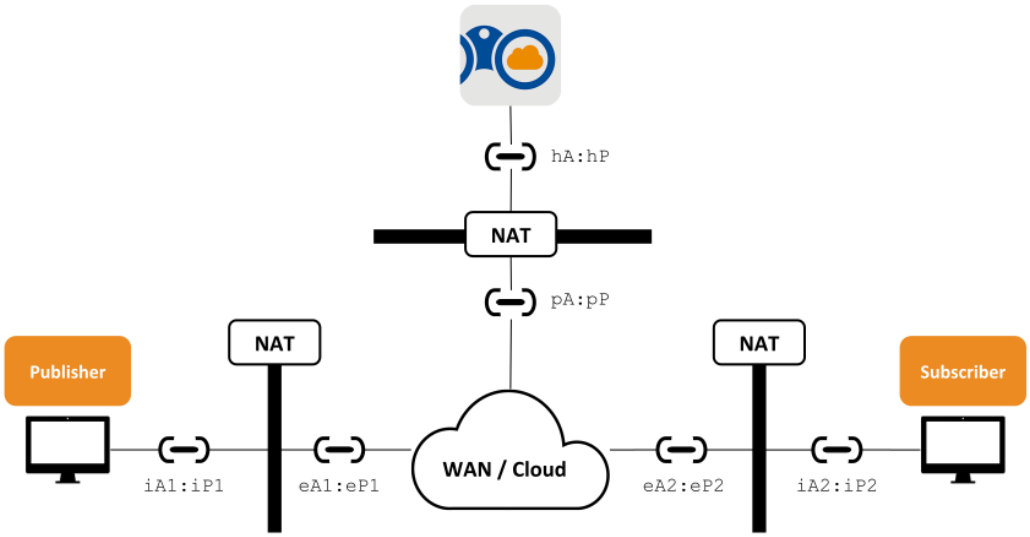

Figure 1.24: Example setup and network layout

Figure 1.24 shows the different hosts and applications involved in this example:

- The publisher and subscriber applications **sit behind different Cone NATs**. You will run these applications in hosts belonging to different networks, in remote locations preferably. These hosts have private transport addresses represented as `iA:iP` and public or external addresses `eA:eP`. The values of these addresses are irrelevant in this example.
- A *Cloud Discovery Service* instance also **sitting behind a NAT of any kind** and that is configured to allow incoming traffic by **statically mapping a public address** `pA:pP` to the private address of *Cloud*

Discovery Service `hA:hP`. You will run *Cloud Discovery Service* in a host from a different network different to the ones the publisher and subscriber applications run. For example, you could run *Cloud Discovery Service* in an EC2 instance from the *Amazon Web Services <https://aws.amazon.com>*.

Warning: This example only works if the publisher and application NATs are of any of the *Cone* type. *Symmetric NATs* are not supported in this setup.

Setup

The first step in this example is to configure the publisher and subscriber applications to communicate over *RTI Real-Time WAN Transport*. You will simply need to configure the corresponding builtin transport mask. For that, you can modify the `<domain_participant_qos>` tag of the file `USER_QOS_PROFILES.xml` generated by *rtiddsgen* to look as follows:

```
<domain_participant_qos>
    ...

    <transport_builtin>
        <mask>UDPv4_WAN</mask>
    </transport_builtin>
</domain_participant_qos>
```

Now you can run the applications making sure each is behind a NAT. You need to run from the directory that contains the generated `USER_QOS_PROFILES.xml`

For the Publisher application in the host behind `NAT Pub`:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_publisher -s 30
```

For the Subscriber application in the host behind `NAT Sub`:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber
```

You will naturally observe that the applications cannot communicate. This is expected since none of them know how reach each other out.

Cloud Discovery Service in Action

Now you will use *Cloud Discovery Service* to provide discovery for the applications. In this case, we will use a builtin configuration part of *Cloud Discovery Service* that enables the *UDP WAN Transport*. The configuration has name `rti.cds.builtin.config.default_wan` and looks as follows:

```
<cloud_discovery_service name="rti.cds.builtin.config.default_wan">
    <annotation>
        <documentation><![CDATA[
            Enables Real-Time WAN Transport.
            XML variables:
```

(continues on next page)

(continued from previous page)

```
                - RTI_CDS_PORT: CDS public and host port number
                - RTI_CDS_PUBLIC_ADDR: CDS WAN public address
            ]]>
        </documentation>
    </annotation>
    <transport>
        <element>
            <alias>builtin.udpv4_wan</alias>
            <receive_port>$(RTI_CDS_PORT)</receive_port>
            <property>
                <element>
                    <name>dds.transport.UDPv4_WAN.builtin.public_address</
↪name>
                    <value>$(RTI_CDS_PUBLIC_ADDR)</value>
                </element>
            </property>
        </element>
    </transport>
</cloud_discovery_service>
```

Note two important elements in this configuration:

- `<receive_port>`: This is the host UDP port where *Cloud Discovery Service* runs. By default this is **also the public port that should be reachable externally**.
- Property `dds.transport.UDPv4_WAN.builtin.public_address`: specifies the **public IPv4 address for the host** where *Cloud Discovery Service* runs. Note that this address may be different than the host private address if *Cloud Discovery Service* sits behind a NAT as well. If this is the case, it's expected that the NAT configuration has a port forwarding rule to map the host port to a public port statically. The default value for `RTI_CDS_PUBLIC_ADDR` is `localhost` (the default host address, typically chosen by the OS).

You will need to deploy *Cloud Discovery Service* in a host machine that is publicly reachable. You may need to configure port forwarding rules if a NAT is present (like the case of EC2 in AWS). For this example we will assume the following values:

- public IPv4 address: `107.21.118.142`
- public UDP port: `39170` (and a forwarding rule to a private port with the same number).

From a terminal in this host machine, run the following command:

```
$NDDSHOME/bin/rticlouddiscoveryservice \
        -cfgName rti.cds.builtin.config.default_wan \
        -DRTI_CDS_PORT=39170 \
        -DRTI_CDS_PUBLIC_ADDR=107.21.118.142
```

Re-run the publisher and subscriber application, but this time set the initial peers to talk to *Cloud Discovery Service* (see *RTPS Peer Descriptor*). For that, you can set the environment in the application terminal as follows, for both the publisher and subscriber applications:

```
export NDDS_DISCOVERY_PEERS=rtps@udpv4_wan://107.21.118.142:39170
```

Now run the applications and verify that the subscriber receives samples. You can also stop *Cloud Discovery Service* and see how the applications **continue communicating**.

1.10.6 Example: Discovering Connext Micro applications with Cloud Discovery Service

This example illustrates how a *Connext Micro* publisher application discovers a subscriber application via an instance of *Cloud Discovery Service*. To reiterate, we are assuming that we are on a network that doesn't support multicast.

Installing Connext Micro

Before we move to how *Cloud Discovery Service* is configured, you first need to have *Connext Micro* installed. This example assumes that you have *Connext Micro* version 3.0.0 or higher installed. This includes the *rtiddsgen* code generator. *rtiddsgen* can be used to generate example applications for publishing and subscribing to data. Earlier versions of *rtiddsgen* shipped with *Connext Micro* didn't support this functionality of generating examples. Please follow the step by step instructions in the *Connext Micro* documentation to build the source code before moving on to the next steps.

Setup

Just like *Connext Professional*, you will need an IDL file to feed into *rtiddsgen* to create an example publisher and subscriber application. You can use the same IDL as used in *Setup*, `CDSHelloWorld.idl`:

```
struct CDSHelloWorld {
    long count;
};
```

Generate the publisher and subscriber applications by running the following command (you can also use C++):

```
$RTIMEHOME/rtiddsgen/scripts/rtiddsgen -micro -language C -example \
        CDSHelloWorld.idl
```

Modify the `CDSHelloWorld_publisher.c` file to increase the `count` variable with each written sample. For instance, you could add the following line to the main writing loop:

```
sample->count = i;
```

The next step is to compile the applications.

Note: *rtiddsgen* creates a README.txt file which contains instructions on how to compile for most standard platforms.

For example:

```
$RTIMEHOME/resource/scripts/rtime-make --config Release --build --name \
        x64Linux3gcc5.4.0 --target Linux --source-dir . \
        -G "Unix Makefiles" --delete
```

This will generate two executables, `CDSHelloWorld_publisher` and `CDSHelloWorld_subscriber`, under the `objs/` folder. Run the examples to make sure everything is as expected. From the top-level directory of the project, run the following commands in two different terminals.

Terminal 1:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_publisher
Written sample 1
Matched a subscriber
Written sample 2
Written sample 3
Written sample 4
Written sample 5
```

Terminal 2:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber
Running for 24 hours, press Ctrl-C to exit
Matched a publisher

Valid sample received

Valid sample received

Valid sample received

Valid sample received
```

You should see that the subscriber application receives the samples generated by the publisher application. This happens because the *Connext Micro* application you created adds the loopback address as a default peer and runs on domain ID 0.

Understanding the Connext Micro Peer Descriptor

Connext Micro follows a different naming convention than *Connext Professional* when it comes to its peer descriptor string. The peer descriptor format is:

```
[participant_index@][interface://]address
```

You can read more about the peer descriptor string in the *Connext Micro* documentation. But the important point to note here is that *Connext Micro* doesn't allow you to directly specify a port number in the peer descriptor. So you need to be aware of which port number is selected based on the combination of participant index and domain ID. As you will see in more detail below, it is essential for the purpose of avoiding port conflicts.

For the use case of *Cloud Discovery Service* working with *Connext Micro*, we have two configuration options:

1. *Configure by Port* - You can compute the port number that the combination of peer descriptor and domain ID will refer to, based on the RTPS well-known ports specification. This idea is useful when one of the *Connext Micro* publishers or subscribers is running on the same machine as *Cloud Discovery Service*.
2. *Configure by Domain ID* - You can configure *Cloud Discovery Service* to run on a particular domain ID and by default it will assume the participant index 0. This idea is useful when *Cloud Discovery Service* is running on a separate machine than the publisher or the subscriber application.

Configure by Port

Setup

This example assumes that *Cloud Discovery Service* runs on the same machine as either the publisher or subscriber application or both. If you want to test a *Cloud Discovery Service* instance running on a different machine, you can still use this example by just changing the IP address part in the peer descriptor or you can check out the simpler method of *Configure by Domain ID*.

Changing the default initial peer

We are going to change the initial peer for the publisher and the subscriber applications so they won't be able to discover each other. To demonstrate this example on the same machine, we will have the publisher and subscriber applications contact a specific participant index that doesn't correspond to either of them. This will allow *Cloud Discovery Service* to come into the picture. To do that, run the publisher and subscriber applications as follows, using the `-peer` and `-domain` command-line arguments.

CDSHelloWorld_publisher:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_publisher -peer [5]@_udp://127.0.0.1 \
        -domain 0
```

CDSHelloWorld_subscriber:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber -peer [5]@_udp://127.0.0.1 \
        -domain 0
```

At this point you should notice that the publisher and subscriber applications cannot discover each other.

Cloud Discovery Service in Action

To make the applications communicate, we need a special configuration for *Cloud Discovery Service*. We need a configuration where *Cloud Discovery Service* listens on the ports referred to by the combination of peer descriptor and domain ID for the publisher and the subscriber applications.

The configuration from file `rti_cds_example_micro.xml` required for this example is shown below:

```
    <cloud_discovery_service name="ExampleMicroByPort">
        <annotation>
            <documentation>
                <![CDATA[
                Example that configures CDS for Micro based on port numbers
                computed from the domain ID and participant index
                ]]>
            </documentation>
        </annotation>
        <transport>
            <element>
                <alias>udpv4</alias>
                <receive_port kind="PORT">7420</receive_port>
            </element>
        </transport>
    </cloud_discovery_service>
```

Here `7420` is the port number corresponding to participant index 5 on domain 0 when the RTPS well-known port mapping is kept to its default setting. You can read more about the defaults in the API documentation. This allows *Cloud Discovery Service* to discover both the publisher and subscriber applications and relay their announcements to each other.

Note: Assuming there are no other *Connext Micro* or *Connext Professional* applications running on the system, if we start the publisher first, it will take the participant index 0 and the subscriber will take the participant index 1 on domain 0. If you have more applications running on your system on the same domain, you will have to increase the participant index in the peer descriptor to be a higher number (than 5 which was used) to avoid port conflicts with other running DDS applications. At the same time you will have to change the port number *Cloud Discovery Service* binds to, based on the corresponding combination of participant index and domain ID.

From the terminal, you can run the following command to use the above configuration:

```
$NDDSHOME/bin/rticlouddiscoveryservice -cfgFile <working_dir>/rti_cds_example_
↪micro.xml \
        -cfgName ExampleMicroByPort
```

Once *Cloud Discovery Service* starts, you should see the publisher and subscriber applications sending data to each other. You can stop *Cloud Discovery Service* and still see the applications continue to communicate.

Configure by Domain ID

Setup

This method assumes that *Cloud Discovery Service* runs on a separate machine. This is because if *Cloud Discovery Service* runs on the same machine as your publisher or subscriber application, you may end up with port conflicts depending on which application (the publisher, subscriber or *Cloud Discovery Service*) is started first. *Cloud Discovery Service* when working in this configuration tries to bind to the port corresponding to

participant index 0. To see *Cloud Discovery Service* in action on the same machine as your *Connext Micro* publisher, subscriber or both, please refer to the *Configure by Port* section.

Changing the default initial peer

We are going to change the initial peer for the publisher and the subscriber application such that they won't be able to discover each other. To do that we will be running the publisher and subscriber applications as follows using the `-peer` and `-domain` command-line arguments.

In the example commands below, note that `10.10.10.10` is a hypothetical address for the machine running *Cloud Discovery Service*. You may want to replace that with the IP address of your machine that is running *Cloud Discovery Service*.

CDSHelloWorld_publisher:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_publisher -peer _udp://10.10.10.10 \
        -domain 0
```

CDSHelloWorld_subscriber:

```
./objs/x64Linux3gcc5.4.0/CDSHelloWorld_subscriber -peer _udp://10.10.10.10 \
        -domain 0
```

Note: *Connext Micro* generated applications have an option to specify the UDP interface name to use for communication. The generated application will rely on a well-known name like `en1` or `eth0` depending on the platform. However, if your interface has a different name, use the command-line option `-udp_intf` to specify it.

At this point you should notice that the publisher and subscriber applications cannot discover each other.

Cloud Discovery Service in Action

To make the applications communicate, we need a special configuration for *Cloud Discovery Service*. We need a configuration where *Cloud Discovery Service* listens on a port computed from the RTPS well-known port mapping for a given domain ID. Note that the participant index cannot be configured and defaults to 0.

The configuration from file `rti_cds_example_micro.xml` required for this example is shown below:

```
    <cloud_discovery_service name="ExampleMicroByDomainID">
        <annotation>
            <documentation>
                <![CDATA[
                Example that configures CDS for Micro based on domain ID␣
↪since Micro
                doesn't allow a peer descriptor to have a port number
                ]]>
            </documentation>
```

(continues on next page)

(continued from previous page)

```
        </annotation>
        <transport>
            <element>
                <alias>udpv4</alias>
                <receive_port kind="DOMAIN_ID_DERIVED">0</receive_port>
            </element>
        </transport>
    </cloud_discovery_service>
```

From the terminal, run the following command:

```
$NDDSHOME/bin/rticlouddiscoveryservice -cfgFile <working_dir>/rti_cds_example_
↪micro.xml \
        -cfgName ExampleMicroByDomainID
```

Once *Cloud Discovery Service* starts you should see the publisher and subscriber applications sending data to each other. As before, you can stop *Cloud Discovery Service* and still see the applications continue to communicate.

1.11 Software Development Kit

You can extend the out-of-the-box behavior of *Cloud Discovery Service* through its *Software Development Kit* (SDK). The SDK provides a set of public interfaces that allow you to control *Cloud Discovery Service* execution.

The SDK is divided in the following modules:

- *RTI Cloud Discovery Service* Library API: This module offers a set of APIs that allow you to create *Cloud Discovery Service* instances in your application. This allows you to run *Cloud Discovery Service* as a library, as described in Section 1.5.2.

Table 1.38 shows which modules are available for each API, along with links to the API documentation.

Table 1.38: API Documentation for the SDK

Language API	Available Modules
RTI Cloud Discovery Service C API	• Library
RTI Cloud Discovery Service C++ API	• Library

1.12 Common Infrastructure

1.12.1 Configuring RTI Services

RTI Services are configured using XML and offer multiple ways to load the configurations. The loading alternatives are in general standard across all RTI Services. This section covers how you can provide XML configurations to RTI Services, as well as specific behaviors on how the XML is parsed, validated, and interpreted.

How to Load and Select an XML Configuration

To run an RTI Service with a specific configuration you need to provide two pieces:

- **XML content with one or more configurations** This is the actual XML code that contains the service-specific configurations. We refer to this as the input XML document. There are two different input sources: File system or in-memory strings.
- **Configuration name** The name of the actual service configuration to be run. Each RTI Service defines a top-level element that shall contain a `name` attribute that uniquely identifies it.

Loading from Files

RTI Services can receive a list of file paths separated by semicolons (`;`):

```
filepath_1;filepath_2; ... filepath_N
```

File paths can be relative or absolute and files are loaded in order from left to right. How you provide the file path list depends on whether you run the service from the shipped executable or embed it into your application using the Library API[1].

Shipped Executable

Use the `-cfgFile` option.

> **Warning:** On some operating systems, `;` is interpreted as a command separator, so you will need to escape the path list with double quotes `"`.

For example on Linux systems:

RTI Routing Service

```
$NDDSHOME/bin/rtiroutingservice -cfgFile "file.xml;/home/file2.xml"
```

RTI Recording Service

[1] Library API may not be available for certain RTI Services.

```
$NDDSHOME/bin/rtirecordingservice -cfgFile "file.xml;/home/file2.xml"
```

RTI Cloud Discovery Service

```
$NDDSHOME/bin/rticlouddiscoveryservice -cfgFile "file.xml;/home/file2.xml"
```

where `[NDDSHOME]` indicates the path to your *Connext* installation.

Library API

Set the `ServiceProperty::cfg_file` member.

For example in C++:

```
ServiceProperty property;
property.cfg_file("file.xml;/home/file2.xml");
...
Service service(property);
```

Loading from In-Memory Strings

If you are embedding RTI Services into your application using the Library API, the input XML document can be also be provided through a string array object. You can do so by setting the `ServiceProperty::cfg_strings` member.

For example in C++:

```
std::vector<std::string> xml_strings;
xml_strings.resize(2);
/* This sample demonstrates using Routing Service */
xml_strings[0] = "<dds><routing_service name=\"MyService\">";
xml_strings[1] = "</routing_service></dds>";
property.cfg_strings(xml_strings);
...
Service service(property);
```

Selecting which Configuration to Run

As stated earlier, the input XML document may contain one or more service configurations. You will need to select which specific configuration to run by providing its configuration name.

How you provide the configuration name depends on whether you run the service from the shipped executable or by embedding it into your application using the Library API.

For example, consider the following input XML document in a file named `MyService.xml` that contains two configurations.

RTI Routing Service

```
<dds>
    <routing_service name="Service1"> ... </routing_service>

    <routing_service name="Service2"> ... </routing_service>
</dds>
```

RTI Recording Service

```
<dds>
    <recording_service name="Service1"> ... </recording_service>

    <recording_service name="Service2"> ... </recording_service>
</dds>
```

RTI Cloud Discovery Service

```
<dds>
    <cloud_discovery_service name="Service1"> ... </cloud_discovery_service>

    <cloud_discovery_service name="Service2"> ... </cloud_discovery_service>
</dds>
```

You can run the configuration for `Service1` as follows:

Shipped Executable

Use the `-cfgName` option.

For example, on Linux systems:

RTI Routing Service

```
$NDDSHOME/bin/rtiroutingservice -cfgFile MyService.xml -cfgName Service1
```

RTI Recording Service

```
$NDDSHOME/bin/rtirecordingservice -cfgFile MyService.xml -cfgName Service1
```

RTI Cloud Discovery Service

```
$NDDSHOME/bin/rticlouddiscoveryservice -cfgFile MyService.xml -cfgName␣
↪Service1
```

Library API

Set the `ServiceProperty::cfg_name` member.

For example in C++:

```
ServiceProperty property;
property.cfg_file("MyService.xml");
property.cfg_name("Service1");
...
Service service(property);
```

Default Files

In addition to manually providing input XML files, RTI Services also attempt to automatically load a set of files from predefined locations:

Table 1.39: RTI Services Default Files

File	Allowed Content
`[working directory]/USER_[SERVICE].xml`	• Service-specific elements • QoS profiles • Types
`[NDDSHOME]/resource/xml/RTI_[SERVICE].xml`	• Service-specific elements • QoS profiles • Types
`[working directory]/USER_QOS_PRORFILES.xml`	• QoS profiles • Types

where `[SERVICE]` refers to the concrete product name in uppercase. For example:

- `ROUTING_SERVICE` for *RTI Routing Service*
- `RECORDING_SERVICE` for *RTI Recording Service*
- `CLOUD_DISCOVERY_SERVICE` for *RTI Cloud Discovery Service*

These files are loaded only if present.

You can disable the loading of default files by using the proper option:

Shipped Executable

Use the `-skipDefaultFiles` option.

Library API

Set the `ServiceProperty::skip_default_files` member to true.

XML Syntax and Validation

The XML representation of DDS-related resources must follow these syntax rules:

- It shall be a well-formed XML document according to the criteria defined in clause 2.1 of the Extensible Markup Language standard.
- It shall use UTF-8 character encoding for XML elements and values.
- It shall use `<dds>` as the root tag of every document.

To validate the loaded configuration, each RTI Service relies on an XSD document that describes the format of the XML content. The validation of the input XML document occurs after all the files and strings have been parsed. If the validation fails, the RTI Service will fail to load the XML and log an error. For example here is an error in the case of *RTI Cloud Discovery Service*:

```
NDDSHOME/bin/rticlouddiscoveryservice
[/cloud_discovery_services/default|CREATE] line 26: Element 'invalid_example_
↪tag': This element is not expected.
[/cloud_discovery_services/default|CREATE] CDSService_loadConfiguration:!
↪validate configuration
[/cloud_discovery_services/default|CREATE] CDSService_initialize:!load␣
↪configuration
[/cloud_discovery_services/default|CREATE] CDSService_new:!init service
main:!create service
```

You can disable the XSD validation process by using the proper option:

Shipped Executable

Use the `-ignoreXsdValidation` option.

Library API

Set the `ServiceProperty::enforce_xsd_validation` member to false.

We recommend including a reference to this document in the XML file that contains the service's configuration; this provides helpful features in code editors such as Visual Studio®, Eclipse®, and NetBeans®, including validation and auto-completion while you are editing the XML file.

The XSD for the RTI Service configuration elements is in `[NDDSHOME]/resource/schema/rti_[service_name].xsd`, where `[service_name]` refers to product name in lower snake case. For example:

- `routing_service` for *RTI Routing Service*
- `recording_service` for *RTI Recording Service*
- `cloud_discovery_service` for *RTI Cloud Discovery Service*

To include a reference to the XSD document in your XML file, use the attribute `xsi:noNamespaceSchemaLocation` in the `<dds>` tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

(continues on next page)

(continued from previous page)

```
    xsi:noNamespaceSchemaLocation="[NDDSHOME]/resource/schema/rti_routing_
↪service.xsd">
    <!-- ... -->
</dds>
```

Warning: The product XSD file provided under `[NDDSHOME]/resource/schema` is to assist you in the process of creating an XML configuration document. RTI Services have the XSD builtin in memory, so making modifications to the reference XSD will not have an impact on the validation process.

Listing Available Configurations

The shipped executables of some RTI Services provide an option to list all the available configurations in the specified input XML document. You can run the service with the `-listConfig` option to list the available configurations and exit. For example, on Linux systems:

RTI Routing Service

```
rtiroutingservice -listConfig
Available configurations:
- default:([NDDSHOME]/resource/xml/RTI_ROUTING_SERVICE.xml)
    Routes all topics from domain 0 to domain 1
- defaultBothWays:([NDDSHOME]/resource/xml/RTI_ROUTING_SERVICE.xml)
    Routes all topics from domain 0 to domain 1 and the other way around
- defaultReliable:([NDDSHOME]/resource/xml/RTI_ROUTING_SERVICE.xml)
    Routes all topics from domain 0 to domain 1 using reliable communication
```

RTI Cloud Discovery Service

```
rticlouddiscoveryservice -listConfig
Available configurations:
- rti.cds.builtin.config.default:(builtin string)
    Empty configuration. Assumes default values.
- rti.cds.builtin.config.default_wan:(builtin string)
    Enables Real-Time WAN Transport.
                XML variables:
                    - RTI_CDS_PORT: CDS public and host port number
                    - RTI_CDS_PUBLIC_ADDR: CDS WAN public address
```

Each listed configuration indicates the input source (file path or string) and the content of the `<documentation>` tag if present. This operation lists all the configurations detected from the specified input XML document from all the locations and files.

Configuration Variables

The builtin XML parser of the RTI Service offers a special mechanism to reuse and customize content at run time through the concept of *Configuration variables*.

A configuration variable is an RTI-specific construct that you can use in the input XML documents to set placeholders for content that **will be expanded at parsing time**. A variable is specified as follows:

```
$(VAR_NAME)
```

where `VAR_NAME` is the name that identifies the variable. You can use configuration variables in your XML content as an attribute value and element text.

```
<element attribute="$(VAR_ATTR)">my expanded $(VAR_TEXT)</element>
```

The possible ways a variable can be expanded are listed below in precedence order:

1. Process environment.

   ```
   export VAR_NAME=my_value
   ```

2. Using a specific option when running the service.

   Shipped Executable

   Use the `-DVAR_NAME=VALUE` option

   ```
   $<rtiservicename> ... -DVAR_NAME=my_value
   ```

   where `<rtiservicename>` is one of `rtiroutingservice`, `rtirecordingservice` or `rticlouddiscoveryservice`.

   Library API

   Set the `ServiceProperty::user_environment` member

   ```
   ServiceProperty property;
   property.user_environment()["VAR_NAME"] = "var_value";
   ...
   ```

3. `<configuration_variables>` section, which represents an unbounded list of variable name-variable value pairs.

   ```
   <configuration_variables>
       <value>
           <element>
               <name>VAR_NAME</name>
               <value>var_value</value>
           </element>
           ...
       </value>
   </configuration_variables>
   ```

All three of these mechanisms can be used in combination or separately. For the above example, you could expand one variable using the process environment and another variable using the command-line option. The following command:

```
export VAR_ATTR=expanded_attr
<rtiservicename> ... -DVAR_TEXT=expanded_text
```

where `<rtiservicename>` is one of `rtiroutingservice`, `rtirecordingservice` or `rticlouddiscoveryservice`, will result in the following actual parsed XML with the expanded variables:

```
<element attribute="expanded_attr">my expanded expanded_text</element>
```

If the RTI Service cannot expand a variable, it will load the XML document and log an error indicating which variable could not be expanded. Here is an example for *RTI Routing Service*:

```
[/routing_services/default|CREATE] RTIXMLUTILSVariableExpansor_
↪expandString:variable with name=ADMIN_DOMAIN_ID not defined
[/routing_services/default|CREATE] RTIXMLUTILSVariableExpansor_visit:!parse␣
↪at line=19 for tag=domain_id: expand environment variable in element text
[/routing_services/default|CREATE] ROUTERXmlVariableExpansor_visit:!parse at␣
↪line=19 for tag=domain_id
...
```

How to Load Default QoS Profiles

Generally, loading a default QoS profile follows the same mechanism as *Connext* applications. The details on how to specify default QoS profiles in XML is explained in the section Overwriting Default QoS in the *RTI Connext Core Libraries User's Manual*.

In short, you will need to mark a profile as the default using the `is_default_qos` attribute. For RTI Services, you will need to do this as part of the default file `USER_QOS_PROFILES.xml` (see *Default Files*). This requirement is necessary since the default QoS profiles are parsed by the underlying *DomainParticipantFactory* and not the service itself.

Warning: Marking as default a QoS profile defined in a different file than `USER_QOS_PROFILES.xml` will have no effect.

How to Set Logging Properties

You can configure different aspects of the logging infrastructure that is part of RTI Services and *Connext*. This section describes different ways to set these logging properties.

Command-Line Options

The shipped executable for an RTI Service typically offers some out-of-the-box options to configure logging. Typically, you will find these options:

- `-verbosity` sets the verbosity level for the messages generated by the service and *Connext*.
- `-logFormat` configures the format of the log messages, such as whether they contain timestamps, thread IDs, etc.
- `-logFile` redirects the logging to a specified text file.

You can refer to the `Usage` section of each individual product user's manual for further details.

Library API

To configure the service-level verbosity, use the `Logger` singleton class part of the Library API. For example, the following sets `WARNING` level for the service logs in *RTI Routing Service*. For other services change the preceding `rti::routing` prefix to match the RTI Service you are working with.

```
rti::routing::Logger::instance().service_verbosity(
        rti::config::Verbosity::WARNING);
```

To configure the *Connext*-level verbosity (for logs generated by the DDS libraries), you can use the *Connext* configuration logger API. For example, the following sets `WARNING` level for the *Connext* logs:

```
rti::config::Logger::instance().verbosity(
        rti::config::Verbosity::WARNING);
```

For the remaining overall logging properties, such as the log format, output file, and so on, you can also use the *Connext* configuration logger API. For example, to redirect the logging to an output file:

```
rti::config::Logger::instance().output_file(my_service_logs.txt);
```

XML Configuration

As an alternative to the previous two methods, you can configure some logging properties through the `LoggingQosPolicy` which can be specified in XML. For more information, see the LOGGING QosPolicy (DDS Extension) in the *RTI Connext Core Libraries User's Manual*.

The Logging QoS is configured within the `<participant_factory_qos>` that is part of a QoS profile. Since multiple profiles can be present in the loaded XML document, to tell *Connext* which one to use, you will need to mark the profile as the default using the `is_default_qos` attribute, or for the DomainParticipantFactory, the `is_default_participant_factory_profile` attribute.

See *How to Load Default QoS Profiles* for details on how to load default QoS profiles with RTI Services. For example, you can set different properties for the logger by placing the XML code seen below in the `USER_QOS_PROFILES.xml` default file:

```
<dds>
    <qos_library name="DefaultLibrary">
        <qos_profile name="DefaultProfile" is_default_participant_factory_
↪profile ="true">
            <participant_factory_qos>
                <logging>
                    <!-- this element affects Connext logs only -->
                    <verbosity>ALL</verbosity>
                    <!-- for all Connext and Service logs -->
                    <category>ENTITIES</category>
                    <print_format>MAXIMAL</print_format>
                    <output_file>LoggerOutput1.txt</output_file>
                </logging>
            </participant_factory_qos>
        </qos_profile>
    </qos_library>
</dds>
```

See also:

Configuring Connext Logging Describes the types of logging messages and how to use the logger to enable them.

Identifying Threads used by Connext DDS Describes the logging messages that provide thread-context information.

How to Run as an Operating System Daemon

Certain Operating Systems offer the capability to run processes in the background and non-interactively. On Linux or macOS systems, this is referred to as *daemon* processes. On Windows systems, this is referred to as a *service*.

How to run a process as a daemon depends on the OS and in some cases there are multiple options. This section describes the most common way to run an RTI Service as a daemon of the main OS.

Linux and macOS Systems

The simplest and more portable way requires you to use the Library API to create your own executable that instantiates the RTI Service and sets the running process as a daemon using the `daemon()` API. For example, for *RTI Routing Service*:

```
#include <stdlib.h>
#include "rti/routing/Service.hpp"

int main(int argc, char **argv)
{
    using namespace rti::routing;

    if (daemon(0,0)) {
```

(continues on next page)

(continued from previous page)

```
        Logger::instance().error("Failed to create daemon process\n");
        return -1;
    }

    // parse arguments and configure ServiceProperty
    ServiceProperty property;
    property.cfg_file(argv[1]);
    ...
    Service service(property);

    service.start();
}
```

The above code generates an executable that runs the process as a daemon with zero-value arguments, indicating that the working directory is `/` and the standard output is redirected to `/dev/null`. You can find more information about the `daemon()` in the user man pages.

Note that if you link the application dynamically, you will need to guarantee that the dependency libraries are available as part of the library path. An alternative is to link the applications statically.

Windows Systems

To run a process as a Windows Service we recommend using the third party tool Non-Sucking Service Manager (NSSM). This tool allows you to run an existing executable as a service, while adjusting environment variables and command-line arguments.

Hence you can use NSSM to run the shipped executable of an RTI Service. For example, for *Routing Service* you can run:

```
nssm install myRouterService <rtiroutingservice> "-cfgName default"
```

The above command will install a service named `myRouterService` on your Windows system that runs *Routing Service* with the default configuration. Then you can manage the service with the `nssm` GUI utility itself or the Windows Services Control Manager (select Control Panel -> Administrative Services -> Services).

The example above causes the service to use the executable directory as the working directory and relies on the default configuration file in `[NDDSHOME]/resource/xml`. You can specify a different working directory as well as different command-line arguments as follows:

```
nssm set myRouterService AppDirectory <my_working_dir>
nssm set myRouterService AppParameters "-cfgFile my_router.xml -cfgName␣
↪MyRoute"
```

Alternatively, you can use the Library API to embed the RTI Service into your own executable and implement the Windows Library APIs to run the executable as a Windows Service. (see How to: Create Windows Services).

Here are some things to consider when running an RTI Service as a Windows Service:

- All `AppParameters` arguments must be enclosed in quotation marks.

- If you specify `-cfgFile` in the Start Parameters field, you must use the full path to the file.
- Some versions of Windows do not allow Windows Services to communicate with other services/applications using shared memory. In such case, you will need to disable the shared memory transport in all *DomainParticipants* created by the RTI Service.
- In some scenarios, you may need to add a multicast address to your discovery peers or simply use *RTI Cloud Discovery Service*.

How to use a License File with RTI Services

If your *RTI Connext* distribution requires a license file, you will receive one from RTI via email. To install the license file, follow the instructions in Installing RTI Connext DDS, in the RTI Connext DDS Installation Guide. Alternatively, you can provide the RTI Service with the path to your license file using either the `-licenseFile` command-line argument or the `license_file_name` field in the Service Property of the Library API.

Note: Some RTI Services do not require a license file.

Check the command line arguments list for the RTI Service to see if a `-licenseFile` argument exists. If it doesn't, you can use the RTI Service without a license file.

Each time your RTI Service starts, it looks for the license file in the following locations, in order, until it finds a valid license:

1. The file specified in the environment variable `RTI_LICENSE_FILE`, which you may set to point to the full path of the license file, including the filename. For example, on Linux:

   ```
   export RTI_LICENSE_FILE=/home/username/my_rti_license.dat
   ```

2. The file `rti_license.dat` in the current working directory.
3. The file `rti_license.dat` in the directory specified by the environment variable `NDDSHOME`.

Key Terms

XML document The input XML contained within the `<dds>` root, which contains one or more configurations for an RTI Service.

Configuration name Unique identification of a service top-level configuration element. Provided with the `name` attribute.

Configuration variable An RTI-specific construct to be used in XML to define content that can be expanded at run time.

Shipped executable An RTI-provided command-line executable that runs an RTI Service.

Library API Public API that allows you to embed an RTI Service into your custom application.

1.12.2 Application Resource Model

RTI Services are described through a *hierarchical application resource model*. In this model, an application is composed of a set of *Resources*, each representing a particular component within the application. *Resources* have a parent-child relationship. Figure 1.25 shows a general view of this concept.

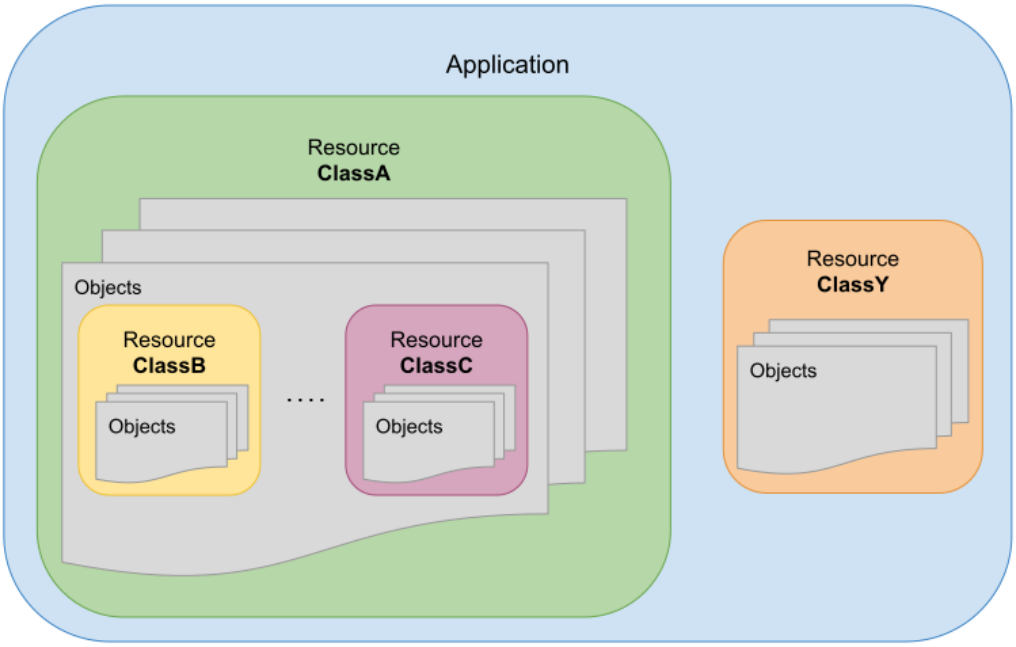

Figure 1.25: Application modeled as a set of related Resources

Each application specifies its resource model by indicating the available resources and their relationship. A *Resource* is determined by its class and a concrete object instance. It can belong to one of the following categories:

- **Simple**–Represents a single object.
- **Collection**–Represents a set of objects of the same class.

A Resource may be composed of one or more Resources. In this relationship, the *parent* Resource is composed of one ore more *child* Resources.

Example: Simple Resource Model of a Connext Application

Figure 1.26 depicts a UML class diagram to provide a generic resource model for *Connext* applications.

In this diagram, the composition relationship is used to denote the parents and children in the hierarchy. The direct relationship denotes a dependency between resources that is not parent-child.

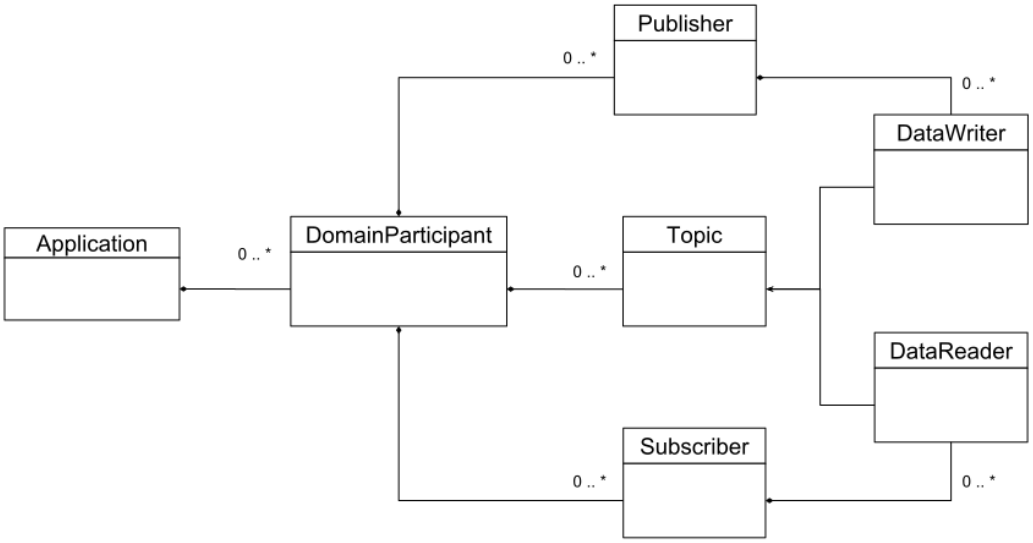

Figure 1.26: Connext DDS application resource model

Resource Identifiers

A resource identifier is a string of characters that uniquely address a concrete resource object within an application. It is expressed as a hierarchical sequence of identifiers separated by /, including all the parent resources and the target resource itself:

$$/resource\_id_1/resource\_id_2.../resource\_id_N$$

where each individual identifier references a concrete resource object *by its name*. The object name is either:

a) Fixed and specified by the resource model of the parent Resource class.

b) Given by the user of the application. This is the case where the parent resource is a collection in which the user can insert objects, providing a name for each of them.

The individual identifier can refer to one of the two kinds of resources, simple and collection resources. For example:

/collection_id$_1$/resource_id$_1$/resource_id$_2$

If the identifier refers to a collection resource, the following child identifier must refer to a simple resource. Both simple and collection resources can be parents (or children). In the previous example, resource_id$_1$ is a simple resource child of collection_id$_1$; it is also the parent of resource_id$_2$.

The hierarchy of identifiers is known as the *full resource identifier path*, where each resource on the left represents a parent resource. The *full resource identifier path* is composed of collection and simple resources. Each child resource identifier is known as the *relative resource* to the parent.

The resource identifier format follows these conventions:

- The first character is /, which represents the root resource and parent of all the available resources across the applications.

- A collection identifier is defined in lower `snake_case`, and it is always specified by the resource class.
- A simple resource identifier is defined in `camelCase` (lower and upper) and may be specified by both the resource class or the user.

Escaped Identifiers

An identifier can be escaped by enclosing it within double quotes (`"`). For example:

/"escaped_identifier"

An escaped identifier is interpreted as a whole and indivisible unit. Escaping a resource identifier is useful; it is also required when the identifier contains the resource separator `/` or the custom method separator `:`.

For example, the following full resource path:

```
/resource_1/"escaped/resource_2"
```

is composed of two relative resources, resource_id$_1$and escaped/resource$_2$. The use of the double quotes to escape the identifier indicates that the enclosing string shall be interpreted as a single identifier, and therefore *Routing Service* ignores the resource separator. If the identifier was not escaped, then *Routing Service* would interpret the resource path as two separate relative resources.

Any time an RTI Service sees a resource separator character (`/`) or the custom method separator `:` in an entity name (such as in the attribute `name`), it automatically escapes the name when it constructs the resource identifier. For example:

```
<service name="A/B">

<service name="A:B">
```

becomes

```
/routing_service/"A/B"

/routing_service/"A:B"
```

in the resource identifier.

Example: Resource Identifiers of a Generic Connext Application

Consider the *Connext* application resource model in *Example: Simple Resource Model of a Connext Application*. The following resource identifier addresses a concrete *DomainParticipant* named "MyParticipant" in a given application:

```
/domain_participants/MyParticipant
```

In this case, "domain_participants" is the identifier of a collection resource that represents a set of *DomainParticipants* in the application and its value is fixed and specified by the application. In contrast, "MyParticipant"

is the identifier of a simple resource that represents a particular *DomainParticipant* and its value is given by the user of the application at *DomainParticipant* creation time.

The following resource identifier addresses the implicit *Publisher* of a concrete *DomainParticipant* in a given application:

```
/domain_participants/MyParticipant/implicit_publisher
```

where "implicit_publisher" is the identifier of a simple resource that represents the always-present implicit *Publisher* and its value is fixed and specified by the *DomainParticipant* resource class.

Example: Resource Identifiers Generated from XML Entity Model

Consider the following XML configuration that models a generic RTI Service:

```
<service name="MyService">
    <entity_class1 name="MyEntity1"> ... </entity_class1>
    <entity_class1 name="Domain/MyEntity2"> ... </entity_class1>
</service>
```

The resulting generated resource identifiers will look as follows:

```
/service/MyService/entity_class1/MyEntity1
/service/MyService/entity_class1/"Domain/MyEntity2"
```

1.12.3 Remote Administration Platform

This section describes details of the *RTI Remote Administration Platform*, which represents the foundation of the remote access capabilities available in *RTI Routing Service*, *RTI Recording Service, RTI Queuing Service*, *RTI Cloud Discovery Service* and *RTI Observability Collector*. The *RTI Remote Administration Platform* provides a common infrastructure that unifies and consolidates the remote interface to all RTI Services.

Note: Remote administration of RTI Services requires an understanding of the *application resource model*. We recommend that you read *Application Resource Model* (*Application Resource Model*) before continuing with this section.

The *RTI Remote Administration Platform* addresses two areas:

- **Resource Interface:** How to perform operations on a set of resource objects that are available as part of the public interface of the remote service.
- **Communication:** How the remote service receives and sends information.

The combination of these two areas provides the general view of the *RTI Remote Administration Platform*, as shown in Figure 1.27. The *RTI Remote Administration Platform* is defined as a request/reply architecture. In this architecture, the service is modeled as a set of *resources* upon which the requester client can perform operations. Resources represent objects that have both *state* and *behavior*.

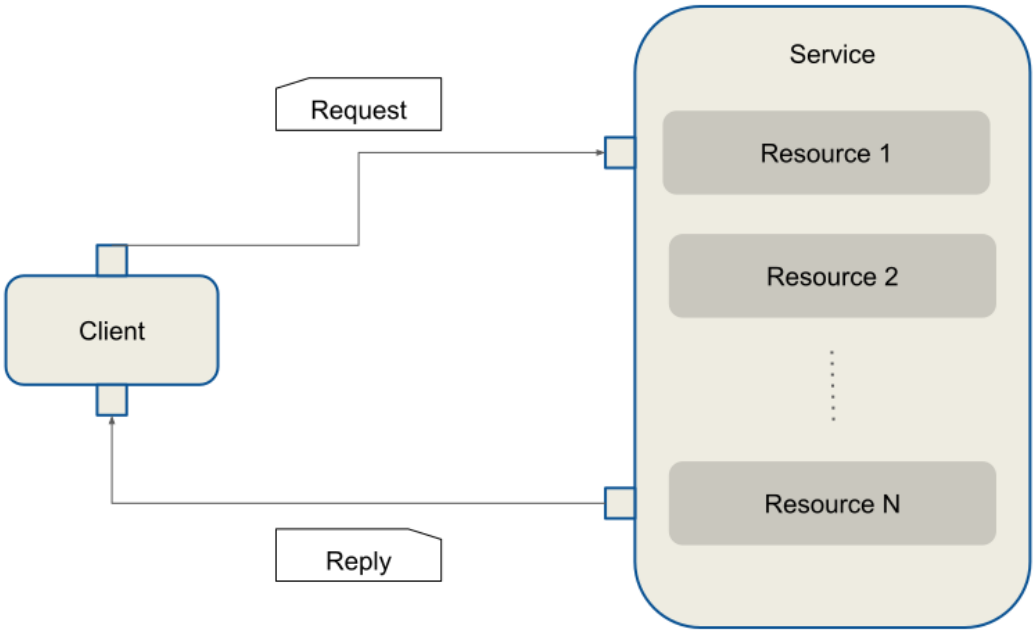

Figure 1.27: General View of the *RTI Remote Administration Platform* Architecture

Clients issue requests indicating the desired operation and receive replies from the service with the result of the requests. If multiple clients issue multiple requests to one or more services, the client will receive only replies to its own requests.

Remote Interface

Services offer their available functionality through their set of resources. The *RTI Remote Administration Platform* defines a Representational State Transfer (REST)-like interface to address service resources and perform operations on them. A resource operation is determined by a REST request and the associated result by a REST reply.

Table 1.40: REST Interface

Element	Description
REST Request	**[method] + [resource_identifier] + [body]** • `method`: Specifies the action to be performed on a service resource. There is only a small subset of methods, known as *standard methods* (see *Standard Methods*). • `resource_identifier`: Addresses a concrete service resource. Each concrete service has its own set of resources (see *Resource Identifiers*). • `body`: Optional request data that contains necessary information to complete the operation.
REST Reply	**[return code] + [body]** • `return code`: Integer indicating the result of the operation. • `body`: Optional reply data that contains information associated with the processing of the request.

Standard Methods

The *RTI Remote Administration Platform* defines the methods listed in Table 1.41.

Table 1.41: Standard Methods

Method	URI	Request Body	Reply Body
`CREATE`	Parent collection resource identifier	Resource representation	N/A
`GET`	Resource identifier	N/A	Resource representation
`UPDATE`	Resource identifier	Resource representation	N/A
`DELETE`	Resource identifier	Undefined	N/A

Custom Methods

There are certain cases in which an operation on a service resource cannot be mapped intuitively to a standard method and resource identifier. *Custom methods* address this limitation.

A custom method can be specified as part of the resource identifier, after the resource path, separated by a `:`.

`UPDATE` + [resource_identifier] **:** [custom_verb]

It is up to each service implementation to define which custom methods are available and on what resources they apply. Custom methods follow these conventions:

- They are invoked through the `UPDATE` standard method.

- They are named using lower snake_case.
- They may use the request body and reply body if necessary.

Example: Database Rollover

This example shows the REST request to perform a file rollover operation on a file-based database:

```
UPDATE /databases/MyDatabase:rollover
```

Communication

The information exchange between client and server is based on the DDS request-reply pattern, as shown in Figure 1.28. The client maps to a *Requester*, whereas the server maps to a *Replier*.

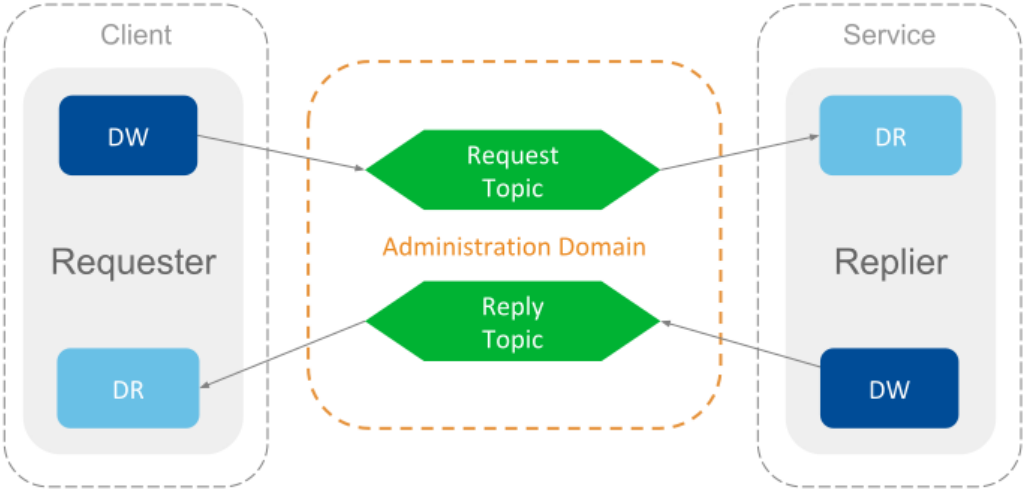

Figure 1.28: Communication in *RTI Remote Administration Platform* is Based on DDS Request-Reply

The communication is performed over a single request-reply channel, composed of two topics:

- **Command Request Topic:** Topic through which the client sends the requests to the server.
- **Command Reply Topic:** Topic through which the server sends the replies to the received requests.

The definition of these topics is shown in Table 1.42:

Table 1.42: Remote Administration *Topic*s

Topic	Name	Top-level Type Name
CommandRequestTopic	rti/service/admin/command_request	`rti::service::admin::CommandRequest`
CommandReplyTopic	rti/service/admin/command_reply	`rti::service::admin::CommandReply`

The definition for each *Topic* type is described below.

Listing 1.6: `CommandRequest` Type

```
@appendable
struct CommandRequest {
    @key int32 instance_id;
    @optional string<BOUNDED_STRING_LENGTH_MAX> application_name;
    CommandActionKind action;
    ResourceIdentifier resource_identifier;
    StringBody string_body;
    OctetBody octet_body;
};
```

Table 1.43: `CommandRequest`

Field Name	Description
`instance_id`	Associates a request with a given instance in the *CommandRequestTopic*. This can be used if your requester application model wants to leverage outstanding requests. In general, this member is always set to zero, so all requests belong to the same *CommandRequestTopic* instance.
`application_name`	Optional member that indicates the target service instance where the request is sent. If `NULL`, the request will be sent to all services.
`action`	Indicates the resource operation.
`resource_identifier`	Addresses a service resource.
`string_body`	Contains content represented as a string.
`octet_body`	Contains content represented as binary.

Listing 1.7: `CommandReply` Type

```
@appendable
struct CommandReply {
    CommandReplyRetcode retcode;
    int32 native_retcode;
    StringBody string_body;
    OctetBody octet_body;
};
```

Table 1.44: `CommandReply`

Field Name	Description
`retcode`	Indicates the result of the operation.
`native_retcode`	Provides extra information about the result of the operation.
`string_body`	Return value of the operation, represented as a string.
`octet_body`	Return value of the operation, represented as binary.

The type definitions for both the *CommandRequestTopic* and *CommandReplyTopic* are in the file `[NDDSHOME]/resource/idl/ServiceAdmin.idl`.

The definition of the request and reply topics is independent of any specific service implementation. In fact, the topic names are fixed, unique, and shared across all services that rely on the *RTI Remote Administration Platform*. Clients can target specific services through two mechanisms:

- Specifying a concrete service instance by providing its *application name*. The application name is a service attribute and can be set at service creation time.
- Specifying the configuration name loaded by the target services. The target service configuration shall be present in the service resource part of the `resource_identifier`.

Reply Sequence

Usually a request is expected to generate a single reply. Sometimes, however, a request may trigger the *generation of multiple replies*, all associated with the same request.

The *RTI Remote Administration Platform* communication architecture allows services to respond to certain requests with a *reply sequence*. All the samples in a reply sequence use the the metadata `SampleFlagBits` to indicate whether it belongs to a reply sequence and whether there are more replies pending.

The `SampleFlagBits` may contain different flags that indicate the status of the reply procedure. For a given reply sequence, the associated sample flags for each reply may contain:

- `SEQUENTIAL_REPLY`: If present, this indicates that the sample is the first reply of a reply sequence and there are more on the way.
- `FINAL_REPLY`: If present, this indicates that the sample is the last one belonging to a reply sequence. This flag is valid only if the `SEQUENTIAL_REPLY` is also set.

For more on SampleFlagBits, see documentation on the DDS_SampleInfo structure in the Connext DDS API Reference HTML documentation.

Example: Controlling services remotely from a Connext Application

The *Connext* GitHub examples repository includes an example that shows how to build and run a requester application that can send commands to a running *RTI Routing Service* instance.

Common Operations

The set of services that use the *RTI Remote Administration Platform* to implement remote administration also share a base remote interface that consolidates and unifies the semantics and behavior of certain common operations.

Services containing resources that implement the common operations conform to the base remote interface, making sure that signatures, semantics, behavior, and conditions are respected.

The following sections describe each of these common operations.

Create Resource

`CREATE [resource_identifier]`

Creates a resource object from its configuration in XML representation.

This operation creates a resource object and its contained entities. The created object becomes a child of the parent specified in the `resource_identifier`.

After successful creation, the resource object is fully addressable for additional remote access, and the associated object configuration is inserted into the currently loaded full XML configuration.

Request body

- `string_body`: XML representation of the resource object provided as `file://` or `str://`.
- Example `str://` request body:

```
str://"<my_resource name="NewResourceObject">
    ...
</my_resource>"
```

- Example `file://` request body:

```
file:///home/rti/config/service_my_resource.xml
```

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.
- The specified configuration is schematically invalid.
- There was an error creating the resource object.

Get Resource

`GET [resource_identifier]`

Returns an equivalent XML string that represents the current state of the resource object configuration, including any updates performed during its lifecycle.

Request body

- Empty.

Reply body

- `string_body`: XML representation of the resource object.
- Example reply body:

```
<my_resource name="MyObject">
    ...
</my_resource>
```

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.

Update Resource

`UPDATE [resource_identifier]`

Updates the specified resource object from its configuration in XML representation.

This operation modifies the properties of the resource object, including the associated configuration. Only the mutable properties of the resource class can be updated while the object is enabled. To update immutable properties, the resource object must be disabled first.

Note: Properties of a child resource cannot be updated as part of a parent resource. Instead, child resources must be addressed and updated independently.

Implementations may validate the received configuration against a scheme (DTD or XSD) that defines the valid set of accepted parameters (for example, only mutable elements).

The update content should only include only the properties to be updated or changed. You are not required to provide the full representation of the object being updated.

For example, consider the XML full representation of an object as follows:

```
<my_resource>
  <nested_resource_A>initial_A</nested_resource_A>
  <nested_resource_B>initial_B</nested_resource_B>
  <nested_resource_C>initial_C</nested_resource_C>
  ...
</my_resource>
```

The update should only contain the content for the properties you want to modify. For example, the following will only update `nested_resource_B` to a new value, leaving the other nested resources unchanged:

```
<my_resource>
  <nested_resource_B>updated_B</nested_resource_B>
  ...
</my_resource>
```

Request body

- `string_body`: XML representation of the resource object provided as `file://` or `str://`.
- Example `str://` request body:

```
str://"<my_resource name="MyResourceObject">
    ...
</my_resource>"
```

- Example `file://` request body:

```
file:///home/rti/config/service_update_my_resource.xml
```

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.
- The specified configuration is schematically invalid.
- The specified configuration contains changes in immutable properties.
- There was an error updating the resource object.

Set Resource State

`UPDATE [resource_identifier]/state`

Sends a state change request to the specified resource object.

This operation attempts to change the state of the specified resource object and propagates the request to the resource object's contained entities.

The target state must be one of the resource class's valid accepted states.

Request body

- `octet_body`: CDR representation of an entity state.

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.
- The target request is invalid.
- The resource object reported an error while performing the state transition.

Get Resource State

`GET [resource_identifier]/state`

Gets the current state of the specified resource object.

This operation attempts to fetch the state of the specified resource object.

The target's state is returned as a part of the reply.

Request body

- Empty

Reply body

- `octet_body`: CDR representation of an entity's current state.

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.
- The target request is invalid.
- The resource object reported an error while fetching its current state.

Delete Resource

`DELETE [resource_identifier]`

Deletes the specified resource object.

This operation deletes a resource object and its contained entities. The deleted object is removed from its parent resource object.

The associated object configuration is removed from the currently loaded full XML configuration.

After a successful deletion, the resource object is no longer addressable for additional remote access.

Request body

- Empty.

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.
- There was an error deleting the resource object.

1.12.4 Monitoring Distribution Platform

Monitoring refers to the distribution of health status information metrics from instrumented RTI Services. This section describes the architecture of the *monitoring* capability supported in *RTI Routing Service* and *RTI Recording Service*. You will learn what type of information these application can provide and how to access it.

RTI Services provide monitoring information through a *Distribution Topic*, which is a DDS *Topic* responsible for distributing information with certain characteristics about the service resources. An RTI Service provides monitoring information through the following **three distribution topics**:

- *ConfigDistributionTopic*: Distributes metrics related to the description and configuration of a Resource. This information may be immutable or change rarely.
- *EventDistributionTopic*: Distributes metrics related to Resource status notifications of asynchronous nature. This information is provided asynchronously when Resources change after the occurrence of an event.
- *PeriodicDistributionTopic*: Distribute metrics related to periodic, sampling-based updates of a Resource. Information is provided periodically at a configurable publication period.

These three *Topic*s are shared across all services for the distribution of the monitoring information. Table 1.45 provides a summary of these topics.

Table 1.45: Monitoring Distribution *Topic*s

Topic	Name	Top-level Type Name
ConfigDistributionTopic	rti/service/monitoring/config	rti::service::monitoring::Config
EventDistributionTopic	rti/service/monitoring/event	rti::service::monitoring::Event
PeriodicDistributionTopic	rti/service/monitoring/periodic	rti::service::monitoring::Periodic

Figure 1.29 shows the mapping of the monitoring information into the distribution *Topic*s. A distribution *Topic* **is keyed** on service resources categorized as *keyed Resources*. These are resources whose related monitoring information is provided as an instance on the distribution *Topic*.

Distribution Topic Definition

All distribution *Topic*s have a common type structure that is composed of two parts: a base type that identifies a resource object and a resource-specific type that contains actual status monitoring information.

The definition of a distribution *Topic* is shown in Figure 1.30.

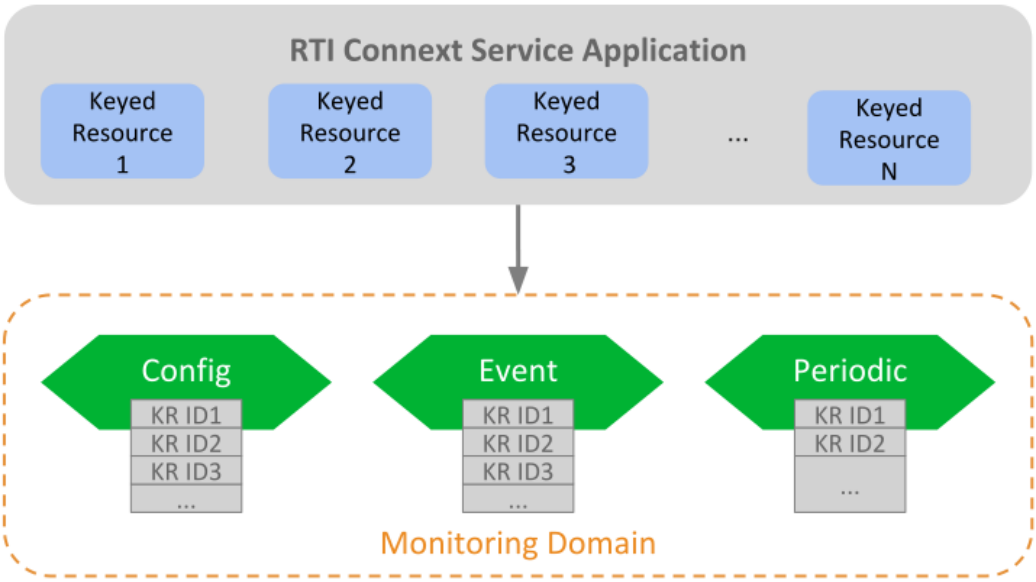

Figure 1.29: Monitoring Distribution *Topic*s of *RTI* Services

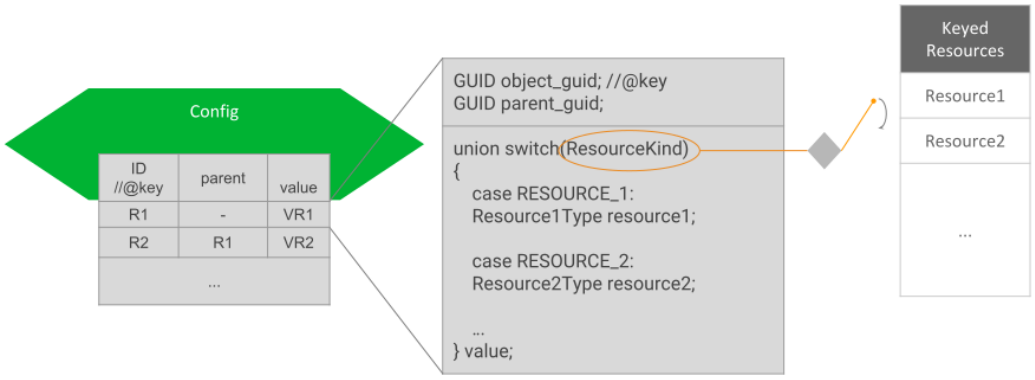

Figure 1.30: Monitoring Distribution *Topic* Definition

Keyed Resource Base Type Fields

This is the base type of all distribution *Topic*s and consists of two fields:

- `object_guid`: Key field. It represents a 16-byte sequence that uniquely identifies a *Keyed Resource* across all the available services in the monitoring domain. Hence, the associated instance handle key hash will be the same for all distribution *Topic*s, allowing easy correlation of a resource. It will also facilitate, as we will discuss later, easy instance data manipulation in a *DataReader*.
- `parent_guid`: It contains the object GUID of the parent resource. This field will be set to all zeros if the object is a top-level resource thus with no parent.

This base type, `KeyedResource`, is defined in `[NDDSHOME]/resource/idl/ServiceCommon.idl`.

Resource-Specific Type Fields

This is the type that conveys monitoring information for a concrete resource object. Since a distribution *Topic* is responsible for providing information about different resource classes, the resource-specific type consists of a single field that is a **Union of all the possible representations** for the keyed resources that provide that on the topic.

As expected, there must be consistency between the two parts of the distribution topic type. That is, a sample for a concrete resource object must contain the resource-specific union discriminator corresponding to the resource object's class.

Example: Monitoring of Generic Application

Assume a generic application that provides monitoring information about the modes of transports `Car`, `Boat` and `Plane`. Each mode is mapped to a keyed resource, each with a custom type that contains metrics specific to each class.

The monitoring distribution *Topic* top-level type, `TransportModeDistribution`, would be defined as follows, using IDL v4 notation:

```
#include "ServiceCommon.idl"

@nested
struct CarType {
    float speed;
    String color;
    String plate_number;
};

@nested
struct BoatType {
    float knots;
    float latitude;
    float longitude;
```

(continues on next page)

(continued from previous page)

```
};

@nested
struct PlaneType {
    float ground_speed;
    int32 air_track;
};

enum TransportModeKind {
    CAR_TRANSPORT_MODE,
    BOAT_TRANSPORT_MODE,
    PLANE_TRANSPORT_MODE
};

@nested
union TransportModeUnion switch (TransportModeKind) {
    case CAR_TRANSPORT_MODE:
    CarType car;

    case BOAT_TRANSPORT_MODE:
    BoatType boat;

    case PLANE_TRANSPORT_MODE:
    PlaneType plane;
}

struct TransportModeDistribution : KeyedResource {
    TransportModeUnion value;

};
```

Assume now that in the monitoring domain there are three resource objects, one for each resource class: a `Car` object 'CarA', a `Boat` object 'Boat1', and a `Plane` object 'PlaneX'. They all have unique resource GUIDs and each object represents an instance in the distribution *Topic*. The table shows the example of potential sample values:

Table 1.46: Samples in TransportModeDistribution *Topic*

	CarA	Boat1	PlaneX
`object_guid`	0x0C	0xAB	0xf2
`parent_guid`	0x00	0x00	0x00
`value` discriminator	CAR_TRANSPORT_MODE	BOAT_TRANSPORT_MODE	PLANE_TRANSPORT_MODE

DDS Entities

RTI Services allow you to distribute monitoring information in any domain. For that, they create the following DDS entities:

- A *DomainParticipant* on the monitoring domain.
- A single *Publisher* for all *DataWriter*s.
- A *DataWriter* for each distribution *Topic*.

A service will create these entities with default QoS or otherwise the corresponding service user's manual will specify the actual values. Services allow you to customize the QoS of the DDS entities, typically in the service monitoring configuration under the `<monitoring>` tag. You will need to refer to each service's user's manual.

Monitoring Metrics Publication

How services publish monitoring samples depends on the distribution *Topic*.

Configuration Distribution Topic

There are two events that cause the publication of samples in this topic:

- As soon as a *Resource* object is created. This event generates the first sample in the *Topic* for the resource object just created. Since these first samples are published as resources are created, it is guaranteed to be in hierarchical order; that is, the sample for a parent *Resource* is published before its children. When *Resources* are created depends on the service. Typically, *Resources* are created on service startup. Other cases include manual creation (e.g., through remote administration) or external event-driven creation (e.g., discovery of matching streams, in the case of *AutoRoute* in *Routing Service*).
- On *Resource* object update. This event occurs when the properties of the object change due to a set or update operation (e.g., through remote administration).

Event Distribution Topic

Services publish samples in this *Topic* in reaction to an internal event, such as a *Resource* state change. Which events and their associated information and when they occur is highly dependent on concrete service implementations.

Periodic Distribution Topic

Samples in this *Topic* are published periodically, according to a fixed configurable period. The metrics provided in this *Topic* are generated in two different ways:

- As a snapshot of the current value, taken at the publication time (e.g., current number of matching *DataReaders*). This represents a simple case and the metric is typically represented with an adequate primitive member.
- As a *statistic variable* generated from a set of discreet measurements, obtained periodically. This represents a *continous* flow of metrics, represented with the `StatisticVariable` type (see *Statistic Variable*).

There are two activities involved in the generation of the statistic variables: Calculation and Publication. All the configuration elements for these activities are available under the `<monitoring>` tag.

Calculation

The instrumented service periodically performs measurements on the metric. This activity is also known as *sampling* (don't confuse with data samples). The frequency of the measurements can be configured with the tag `<statistics_sampling_period>`. As a general recommendation, the sampling period should be a few times smaller than the publication period. A small sampling period provides more accurate statistics generation at the expense of increasing memory and CPU consumption.

Publication

The service periodically publishes a data sample containing a snapshot of the statistics generated during the calculation phase. The publication period can be configured with the tag `<status_publication_period>`.The value of a statistic variable corresponds to the time window of a publication period.

Monitoring Metrics Reference

This section describes the types used as common metrics across services. All the type definitions listed here are in `[NDDSHOME]/resource/idl/ServiceCommon.idl`.

Statistic Variable

Listing 1.8: Statistics

```
@appendable @nested
struct StatisticMetrics {
    uint64 period_ms;
    int64 count;
    float mean;
```

(continues on next page)

(continued from previous page)

```
        float minimum;
        float maximum;
        float std_dev;
    };

    @appendable @nested
    struct StatisticVariable {
        StatisticMetrics publication_period_metrics;
    };
```

Table 1.47: `StatisticMetrics`

Field Name	Description
period_ms	Period in milliseconds at which the metrics are published.
count	Sum of all the measurement values obtained during the publication period.
mean	Arithmetic mean of all the measurement values during publication period. For aggregated metrics, this value is the mean of all the aggregated metrics means.
min	Minimum of all the measurement values during publication period. For aggregated metrics, this value is the minimum of all the aggregated metrics minimums.
max	Maximum of all the measurement values during publication period. For aggregated metrics, this value is the maximum of all the aggregated metrics minimums.
std_dev	Standard deviation of all the measurement values during publication period. For aggregated metrics, this value is the standard deviation of all the aggregated metrics minimums.

Host Metrics

Listing 1.9: Host Types

```
    @appendable @nested
    struct HostPeriodic {
        @optional StatisticVariable cpu_usage_percentage;
        @optional StatisticVariable free_memory_kb;
        @optional StatisticVariable free_swap_memory_kb;
        int32 uptime_sec;
    };

    @appendable @nested
    struct HostConfig {
        BoundedString name;
        uint32 id;
        int64 total_memory_kb;
        int64 total_swap_memory_kb;
        BoundedString target;
```

(continues on next page)

(continued from previous page)

```
        };
```

Table 1.48: `HostConfig`

Field Name	Description
name	Name of the host where the service is running.
id	ID of the host where the service is running.
total_memory_kb	Total memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.
total_swap_memory_kb	Total swap memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.

Table 1.49: `HostPeriodic`

Field Name	Description
cpu_usage_percentage	Statistic variable that provides the global percentage of CPU usage on the host where the service is running. Availability of this value is platform dependent.
free_memory_kb	Statistic variable that provides the amount of free memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.
free_wap_memory_kb	Statistic variable that provides the amount of free swap memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.
uptime_sec	Time in seconds elapsed since the host on which the running service started. Availability of this value is platform dependent.

Process Metrics

Listing 1.10: Process Types

```
@appendable @nested
struct ProcessConfig {
    uint64 id;
};
@mutable @nested
struct ProcessPeriodic {
    @optional StatisticVariable cpu_usage_percentage;
    @optional StatisticVariable physical_memory_kb;
    @optional StatisticVariable total_memory_kb;
    int32 uptime_sec;
};
```

Table 1.50: `ProcessConfig`

Field Name	Description
id	Identifies the process where the service is running. The meaning of this value is platform dependent.

Table 1.51: `ProcessPeriodic`

Field Name	Description
cpu_usage_percentage	Statistic variable that provides the percentage of CPU usage of the process where the service is running. The field `count` of the variable contains the total CPU time in ms that the process spent during the publication period. Availability of this value is platform dependent.
physical_memory_kb	Statistic variable that provides the physical memory utilization in KiloBytes of the process where the service is running. Availability of this value is platform dependent.
total_memory_kb	Statistic variable that provides the virtual memory utilization in KiloBytes of the process where the service is running. Availability of this value is platform dependent.
uptime_sec	Time in seconds elapsed since the running service process started. Availability of this value is platform dependent.

Base Entity Resource Metrics

Listing 1.11: Base Entity Types

```
@mutable @nested
struct EntityConfig {
    ResourceId resource_id;
    XmlString configuration;
};
@mutable @nested
struct EntityEvent{
    EntityStateKind state;
```

(continues on next page)

(continued from previous page)

```
        };
```

Table 1.52: `EntityConfig`

Field Name	Description
resource_id	String representation of the resource identifier associated with the entity resource.
configuration	String representation of the XML configuration of the entity resource. The XML contains only children elements that are not entity resources.

Table 1.53: `EntityEvent`

Field Name	Description
state	State of the resource entity expressed as an enumeration of type `EntityStateKind`.

Network Performance Metrics

Listing 1.12: Network Performance Type

```
@appendable @nested
struct NetworkPerformance {
    @optional StatisticVariable samples_per_sec;
    @optional StatisticVariable bytes_per_sec;
    @optional StatisticVariable latency_millisec;
};
```

Table 1.54: `NetworkPerformance`

Field Name	Description
samples_per_sec	Statistic variable that provides information about the number of samples processed (received or sent) per second.
bytes_per_sec	Statistic variable that provides information about the number of bytes processed (received or sent) per second.
latency_millisec	Statistic variable that provides information about the latency in milliseconds for the data processed. The latency in a refers to the total time elapsed during the associated processing of the data, which depends on the type of application.

Thread Metrics

Listing 1.13: Thread Metrics Type

```
@mutable @nested
struct ThreadPeriodic {
    uint64 id;
    @optional StatisticVariable cpu_usage_percentage;
};

@mutable @nested
struct ThreadPoolPeriodic {
    @optional sequence<Service::Monitoring::ThreadPeriodic>␣
↪threads;
};
```

Table 1.55: `ThreadPeriodic`

Field Name	Description
id	OS-assigned thread identifier
cpu_usage_percentage	Statistic variable that provides the percentage of CPU usage of the thread belonging to the process where the service is running. The field `count` of the variable contains the total CPU time in ms that the thread spent during the publication period. Availability of this value is platform dependent.

1.12.5 Plugin Management

Some RTI Services allow for custom behavior through the use of *pluggable* components or *plugins* . The type of plugins is described in *Software Development Kit*. A plugin is represented as a top-level service-owned object whose main role is a factory of other pluggable components, which are responsible for providing the user-defined behavior.

Figure 1.31 shows that for each *class* of pluggable components there is a top-level object with the suffix `Plugin`. This is the object that the *Service* obtains at the moment of loading the plugin. Multiple `Plugin` objects can be registered from the same class, each uniquely identified by its *registered name*.

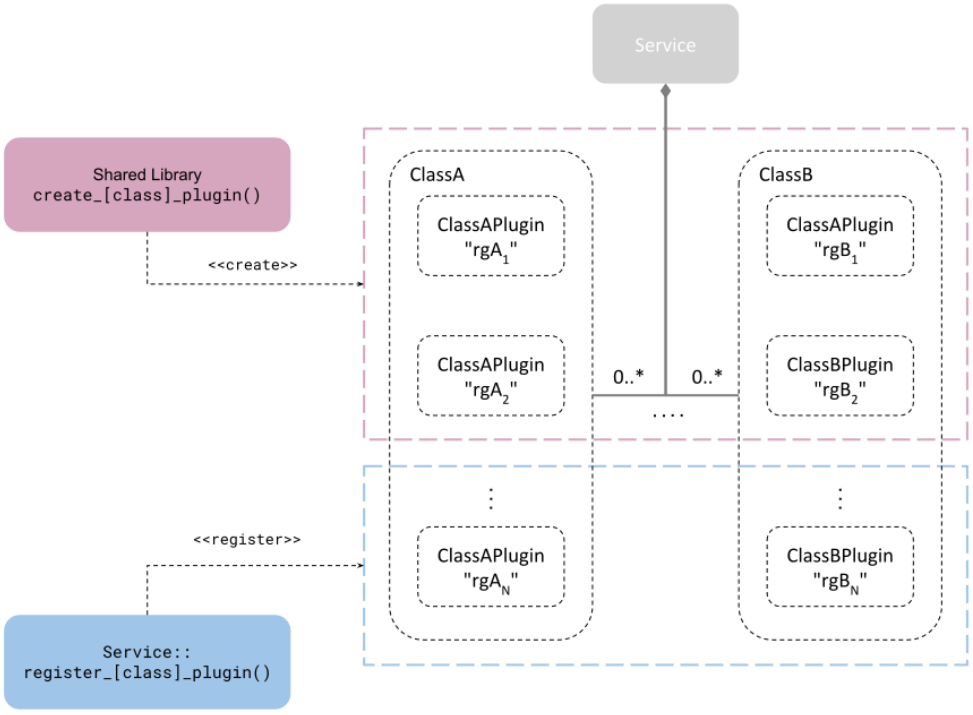

Figure 1.31: Plugin object management

Figure 1.31 also shows that there are two mechanisms through which a *Service* obtains a plugin object: a *shared library* or the Library API. Both mechanisms are complementary and are described with more detail in the next sections.

Shared Library

A plugin object is instantiated through a *create function*, which is included and addressable as part of a shared library. This function is also known as the *entry point* and each RTI Service defines the signature for each plugin class. This method requires specifying the path to the shared library and the name of the entry point (see *Configuration*). The *Service* loads the library the first time an instance of the plugin is needed (lazy initialization) and looks up the specified entry point symbol in the loaded library. The *Service* will always delete the plugin on *Service* stop.

This is the only method suitable when an RTI Service is deployed through an already linked executable, such as the shipped command-line executable (*Command-Line Executable*).

The plugin lifecycle is as follows:

1. After start, the *Service* creates a plugin object for each registered plugin in the configuration. The plugin object is instantiated through the shared library entry point, specified in the configuration.
2. The *Service* calls operations on the plugin objects as needed and keeps them alive while the *Service* remains started.
3. During stop, the *Service* deletes each plugin object by calling the class abstract deleter.

Configuration

An RTI Service configures the pluggable components within the `<plugin_library>` tag. RTI Services that support plugins will define a set of tags within in the form:

- `<[class]_plugin>` for C/C++ plugins
- `<java_[class]_plugin>` for Java plugins

where `[class]` refers to the name of the plugin class. For example, in *Routing Service* an available tag is `<adapter_plugin>`.

The definition of these tags is the same regardless of the plugin class and is described in the tables below.

Table 1.56 and Table 1.57 describe the configuration of each different plugin language.

Table 1.56: Configuration tags for C/C++ plugins.

Tags within <[class]_plugin>	Description	Multiplicity
<dll>	Shared library containing the implementation of the adapter plugin. This tag may specify the exact path (absolute or relative) of the file (for example, **lib/libmyplugin.so**) or a general name (no file extension). If no extension is provided, the path will be completed based on the running platform. For example, assuming a value for this tag of **dir/myplugin**: • Linux/macOS systems (or similar): **dir/libmyplugin.so** • Windows systems: **dir/myplugin.dll** If the library specified in this tag cannot be loaded (because the environment library path is not pointing to the path where the library is located), *Routing Service* will look for the library in the following locations, in this order: • **[plugin_search_path]**: Provided as part of the service parameters (see *Usage*) • **[executable_dir]**: Directory where the executable lives	1

continues on next page

Table 1.56 – continued from previous page

Tags within <[class]_plugin>	Description	Multiplicity
<create_function>	Entry point. This tag must contain the name of the function used to create the plugin instance. The function symbol must be present in the shared library specified in `<dll>`	1

<property>	A sequence of name-value string pairs that allow you to configure the plugin instance.
	Example:
	`<property>`
	`  <value>`
	`  <element>`
	`  <name>myplugin.user_name</name>`
	`  <value>myusername</value>`
	`  </element>`
	`  </value>`
	`</property>`
		0..1

Table 1.57: Configuration tags for Java plugins

Tags within <java_[class]_plugin>	Description	Multiplicity
<class_name>	Name of the class that implements the plugin. For example: com.myplugins.CustomPlugin The classpath required to run the Java plugin must be part of the RTI Service JVM configuration. See the `<jvm>` tag within the specific service configuration for additional information on JVM creation and configuration.	1

<property>	A sequence of name-value string pairs that allow you to configure the plugin instance.
	Example:
	`<property>`
	`  <value>`
	`  <element>`
	`  <name>myplugin.user_name</name>`
	`  <value>myusername</value>`
	`  </element>`
	`  </value>`
	`</property>`
		0..1

Library API

The user provides the plugin object via the Library API, through one of the available `attach_[class]_plugin()` operations. Upon successful return of the operation, the *Service* takes ownership of the plugin object and will delete it on *Service* stop.

The plugin lifecycle is as follows:

1. The user instantiates plugin objects and provides them to the *Service* through the `attach_[class]_plugin()` operation. This is allowed only before the *Service* starts.
2. After start, the *Service* becomes the owner of the registered plugin objects, calls operations on the plugin objects as needed, and keeps them alive while the *Service* remains started.
3. On stop, the *Service* deletes each registered plugin object by calling the class abstract deleter.

1.13 Troubleshooting

This section covers a few issues you may run into while deploying your system with *Cloud Discovery Service*.

Note: All the issues addressed in this section assume that your applications would communicate directly with each other without *Cloud Discovery Service*.

1.13.1 My Applications don't Communicate

There can be many reasons why your applications may not communicate with *Cloud Discovery Service*. Nevertheless, below are some of the common issues you may run into.

Make Sure Your Application can Accept Unknown Peers

For *DomainParticipants* to discover each other through *Cloud Discovery Service*, they require the ability to accept unknown peers. This is a Discovery QoS setting whose default value is true. Make sure your *DomainParticipants* are created with this value. For instance, you can add the following XML setting to the `<domain_participant_qos>` element used by your application:

```
<discovery>
    <accept_unknown_peers>true</accept_unknown_peers>
</discovery>
```

Check that Your Initial Peers List Points to Cloud Discovery Service

Your applications are required to provide discovery information to *Cloud Discovery Service*. For that, they must know where *Cloud Discovery Service* is running. You provide this information through the initial peers list using an *RTPS peer descriptor*.

You can specify the initial peers list through the Discovery QoS (programmatically, XML, or the environment). For instance, assuming a *Cloud Discovery Service* instance is running on UDP, port 7400 and host CDS_HOST:

Listing 1.14: XML

```
<discovery>
    <initial_peers>
        <element>rtps@udpv4://CDS_HOST:7400</element>
    </initial_peers>
</discovery>
```

Listing 1.15: Environment Variables

```
export NDDS_DISCOVERY_PEERS=rtps@udpv4://CDS_HOST:7400
```

See Where Your Cloud Discovery Service Instance is Listening

To verify that *Cloud Discovery Service* is actually listening where you specified, you can run with `-verbosity LOCAL` and look for the following message:

```
...
[.../receiver|ENABLE] listening for announcements on:
{
  "locator":[
    "udpv4://172.17.0.2:10000"
  ]
}
...
```

The message above indicates that *Cloud Discovery Service* is listening on the interface `172.17.0.2` over the `UDPv4` transport on port `10000`.

Additionally, make sure the *Cloud Discovery Service* location is reachable from where your applications run.

Identifying NAT traversal address resolutions

If you're running in a WAN environment along with *RTI Real-Time WAN Transport*, you can find out about how *Cloud Discovery Service* is resolve addresses and for which *DomainParticipants*. See *Debugging Cloud Discovery Service with the UDP WAN Transport* for details on how to obtain locator resolution information.

In general, you will need to verify that:

- *Cloud Discovery Service* receives announcements from the remote participants and is able to detect their public addresses.

- Remote participants sit behind Cone NATs.
- Firewalls/NAT configurations accept incoming traffic from the expected range of public addresses.

1.13.2 Cloud Discovery Service Log Errors

This section shows a few of the errors that *Cloud Discovery Service* may show on startup.

Invalid Port

If an invalid port is specified, you may see this error log output:

```
...
NDDS_Transport_UDPv4_Socket_bindWithIp:OS bind() failure, error 0: Success
NDDS_Transport_UDPv4_Socket_bindWithIp:invalid port 70000
NDDS_Transport_UDPv4_SocketFactory_create_receive_socket:invalid port 70000
NDDS_Transport_UDP_create_recvresource_rrEA:!create socket
COMMENDLocalReaderRW_init:!create unicast entryPort
...
```

The messages above indicate an error binding a UDP socket to port 70000, which is out of the valid port range for this transport.

Port Already in Use

If a port that is already in use is specified, you may see this error log output:

```
...
NDDS_Transport_UDPv4_Socket_bindWithIp:0X1CE8 in use
NDDS_Transport_UDPv4_SocketFactory_create_receive_socket:invalid port 7400
NDDS_Transport_UDP_create_recvresource_rrEA:!create socket
COMMENDLocalReaderRW_init:!create unicast entryPort
...
```

The messages above indicate an error binding a UDP socket to port 7400, which is being used by other process.

1.14 Release Notes

1.14.1 Supported Platforms

See the column for *Cloud Discovery Service* in the Table of Supported Platforms for Connext Applications, in the RTI Connext Core Libraries Release Notes.

1.14.2 Compatibility

For backward compatibility information between the current and previous releases of *Cloud Discovery Service*, please see the *Migration Guide* on the RTI Community portal.

Connext compatibility

Cloud Discovery Service can be used to provide discovery for applications built with *RTI Connext*, except as noted below.

- Starting in *RTI Connext DDS* 5.1.0, the default `message_size_max` for the UDPv4, UDPv6, TCP, and shared-memory transports changed to provide better out-of-the-box performance. *Cloud Discovery Service* also uses the new value for `message_size_max`. Consequently, *Cloud Discovery Service* is not out-of-the-box compatible with applications running older versions of *Connext*. Please see the *RTI Connext Core Libraries Release Notes* for instructions on how to resolve this compatibility issue with older *Connext* applications.

1.14.3 What's New in 7.2.0

Third-party software changes

The following third-party software used by *Cloud Discovery Service* have been upgraded:

Table 1.58: Third-Party Software Changes

Third-Party Software	Previous Version	Current Version
libxml2	2.9.4	2.11.4
libxslt	1.1.35	1.1.38

For information on third-party software used by *Connext* products, see the "3rdPartySoftware" documents in your installation: `<NDDSHOME>/doc/manuals/connext_dds_professional/release_notes_3rdparty`.

Simple Participant Discovery Protocol 2.0 Integration

Cloud Discovery Service now supports the new *Simple Participant Discovery Protocol 2.0*. The *Simple Participant Discovery Protocol 2.0* is an alternative to the original *Simple Participant Discovery Protocol*. *Simple Participant Discovery Protocol 2.0* is designed to decrease bandwidth usage and improve the reliability of the participant discovery and update process.

For information on how to configure *Cloud Discovery Service* to use *Simple Participant Discovery Protocol 2.0*, see the section on *Protocol Mode*.

RTI Lightweight Security Plugins Integration

Cloud Discovery Service now supports the new *RTI Lightweight Security Plugins* library for pre-shared key security. The *RTI Lightweight Security Plugins* is a separate library that defines an alternate set of *Security Plugins*. These plugins allow RTPS messages to be protected with a per-participant key derived from some publicly available data and a pre-shared key seed.

For information on how to configure and use the *RTI Lightweight Security Plugins* in *Cloud Discovery Service*, see the section on *Security*.

See the Migration Guide on the RTI Community Portal for migration issues related to this integration.

1.14.4 What's Fixed in 7.2.0

Fixes Related to Discovery

DomainParticipant discovery may have failed when using Real-Time WAN Transport

Cloud Discovery Service had an internal state race condition when using *Real-Time WAN Transport*. As a result, the discovery of some *DomainParticipants* failed if the race condition was hit. This issue only manifested when using *Real-Time WAN Transport*. The issue has been resolved.

[RTI Issue ID CDS-214]

Potential crash if dispose received when participant announcement forwarded

In previous releases, *Cloud Discovery Service* could crash if a dispose participant announcement was received while the same participant's stored announcement was being forwarded. This issue was due to an internal concurrency management problem. The problem has been resolved.

[RTI Issue ID CDS-228]

Fixes Related to Usability

Crash on startup

Cloud Discovery Service sometimes crashed on startup due to an internal timing issue. This issue did not affect running instances of *Cloud Discovery Service*. The problem has been resolved.

[RTI Issue ID CDS-201]

1.14.5 Previous Releases

What's New in 7.1.0

Cloud Discovery Service now shipped as host and target

Cloud Discovery Service is now shipped in two parts, as a host and a target package. This change brings Cloud Discovery Service in line with how other Connext packages are structured, as a host (base platform) and a target (specific architecture). The change also ensures that you have target-specific binaries for libraries/executables, which are useful when using the service- as-a-library functionality provided by the Library API for *Cloud Discovery Service*. When installing *Cloud Discovery Service*, first install the host followed by the target.

Note: This change does not apply to the version of *Cloud Discovery Service* shipped with the *Connext* LM package. The *Connext* LM package is shipped as a target-specific installation, which already ensures you have access to the correct binary library/executable for your platform.

What's Fixed in 7.1.0

Cloud Discovery Service internal state issues caused increase in network traffic in certain situations

Cloud Discovery Service uses internal buffer pools to store the incoming participant announcements. Due to an issue with the state management, there were certain instances where an incoming announcement could have been reclassified as an update, even when there was no change in its contents. This caused the resend mechanism (if configured) to trigger additional resends, resulting in increased network traffic. The issue has been resolved.

[RTI Issue ID CDS-199]

<output_capacity_allocation> XML tag not parsed from configuration

Cloud Discovery Service didn't parse the XML tag `<output_capacity_allocation>` when specified by the user in XML. As a result, users were unable to control the capacity allocation for the flow controller via XML. This issue has been resolved.

[RTI Issue ID CDS-198]

Discovery did not complete when using Real-Time WAN and multiple locators

When using a *DomainParticipant* with multiple *RTI Real-Time WAN Transport* locators, sometimes *Cloud Discovery Service* mixed up the locators and they were not resolved correctly. This caused the discovery process to never complete. This problem has been resolved.

[RTI Issue ID CDS-172]

What's New in 7.0.0

Improved steady state bandwidth utilization and discovery time in systems with many DomainParticipants communicating over lossy networks

In lossy networks, *DomainParticipant* discovery could take longer if *DomainParticipant* announcements forwarded from *Cloud Discovery Service* were dropped. In previous releases, the `<refresh_period>` configuration tag could be used to resend *DomainParticipant* announcements periodically. The tradeoff of using that parameter was that it could lead to unnecessary network traffic on systems that had completed *DomainParticipant* discovery (because the `<refresh_period>` mechanism continued resending indefinitely).

In this release, the `<refresh_period>` is replaced with a more efficient way to resend *DomainParticipant* announcements. This new mechanism resends the *DomainParticipant* announcements from *Cloud Discovery Service* a finite number of times, spaced randomly in an interval between a minimum and maximum value:

```
<forwarder>
    <event>
        <new_or_update_participant_announcements>
        </new_or_update_participant_announcements>

        <min_new_or_update_participant_announcements_period>
        </min_new_or_update_participant_announcements_period>

        <max_new_or_update_participant_announcements_period>
        </max_new_or_update_participant_announcements_period>
    </event>
</forwarder>
```

For additional information about these new parameters, see *Forwarder*.

Domain Participant Partitions support

Cloud Discovery Service supports systems that utilize *Domain Participant Partitions*. Partitioning at the *DomainParticipant* level can be particularly useful in large, WAN, distributed systems (with thousands of participants) in which not all participants need to know about each other at any given time.

Domain Participant Partitions functionality reduces network, CPU, and memory utilization, because *DomainParticipants* without matching partitions will not exchange information about themselves and their *DataWriters* and *DataReaders*.

Domain Participant Partitions support in *Cloud Discovery Service* ensures that *Cloud Discovery Service* doesn't forward a *DomainParticipant* announcement to *DomainParticipants* whose partition values don't match.

For details, see *Domain Participant Partitions*.

Removals

domain_id_info_seq field in DatabasePeriodic type used in monitoring removed

The `@optional sequence<DomainIdInfo> domain_id_info_seq` field in the DatabasePeriodic type used for *Monitoring* and defined in `CdsMonitoring.idl` has been removed. This change was done as a part of refactoring the internal state of *Cloud Discovery Service*. Refer to the *Migration Guide* on the RTI Community portal to see how this change could affect your upgrade to 7.0.0.

<refresh_period> tag removed

The `<refresh_period>` tag from previous releases has been replaced by new tags. (See *Improved steady state bandwidth utilization and discovery time in systems with many DomainParticipants communicating over lossy networks*.) The `<refresh_period>` is still present in the XSD, and *Cloud Discovery Service* logs a warning if that tag is parsed; however, the value of `<refresh_period>` no longer influences the operation of *Cloud Discovery Service*.

Third-Party Software Upgrades

The following third-party software used by *Cloud Discovery Service* has been upgraded:

Table 1.59: Third-Party Software Changes

Third-Party Software	Previous Version	Current Version
libxml2	2.9.12	2.9.14
libxslt	1.1.34	1.1.35

For information on third-party software used by *Connext* products, see the "3rdPartySoftware" documents in your installation: `<NDDSHOME>/doc/manuals/connext_dds_professional/release_notes_3rdparty`.

What's Fixed in 7.0.0

Problems with Discovery after disposal of a Remote Participant

Cloud Discovery Service may have assigned incorrect properties from one remote participant to another remote participant. This caused discovery problems. These incorrect properties typically belonged to another remote participant that was recently disposed. This problem has been resolved.

[RTI Issue ID CDS-155]

Cloud Discovery Service did not allow using other transports when Real-Time WAN Transport was enabled

Cloud Discovery Service had an issue that prevented the use of multiple transports at the same time, when the Real-Time WAN Transport (udpv4_wan transport alias) was enabled. This issue has been resolved.

[RTI Issue ID CDS-158]

Cloud Discovery Service did not allow use of UDPv6 transport alias

Cloud Discovery Service did not enable the UDPv6 transport (udpv6 transport alias), even when it was set in the <transport> tag. This problem has been resolved.

[RTI Issue ID CDS-161]

Cloud Discovery Service crashed when empty participant GUID provided to Remote Administration API

If the `/cloud_discovery_services/[service_name]/database/locators` Remote Administration endpoint was provided an empty Remote Participant GUID in its `string_body`, *Cloud Discovery Service* crashed due to a parsing error. This problem has been resolved.

[RTI Issue ID CDS-176]

1.14.6 Known Issues

Note: For an updated list of critical known issues, see the Critical Issues List on the RTI Customer Portal at https://support.rti.com/.

Cloud Discovery Service does not terminate when its internal DomainParticipant does not initialize properly

Cloud Discovery Service creates an internal *DomainParticipant* that reserves the *participant_id* 0, on domain ID 0. This *DomainParticipant* is required by *Cloud Discovery Service* to relay the participant discovery announcements. If the *participant_id* 0 for domain ID 0 is already being used by another *Connext* application, *Cloud Discovery Service* throws errors but doesn't terminate.

Although such a *Cloud Discovery Service* instance is running, it won't be able to relay any participant discovery announcements. A simple workaround is to ensure that *Cloud Discovery Service* is always the first *Connext* application started on a given machine.

[RTI Issue ID CDS-143]

Fourth digit of product version not logged by Cloud Discovery Service at startup

Cloud Discovery Service does not log the fourth digit (revision) of the product version at startup.

[RTI Issue ID CDS-195]

1.15 Copyrights and Notices

Trademarks

Copy and Use Restrictions

Third-Party Software

Notices

Deprecations and Removals

Any deprecations or removals noted in this document serve as notice under the Real-Time Innovations, Inc. Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance and support of RTI's software.

Deprecated means that the item is still supported in the release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported. By specifying that an item is deprecated in a release, RTI hereby provides customer notice that RTI reserves the right after one year from the date of such release and,

with or without further notice, to immediately terminate maintenance (including without limitation, providing updates and upgrades) for the item, and no longer support the item, in a future release.

Technical Support Real-Time Innovations, Inc. 232 E. Java Drive Sunnyvale, CA 94089 Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/

Index