RTI Routing Service

User's Manual

Version 6.0.1

Contents

Chapter 1

Introduction

RTI® Routing Service, is an out-of-the-box solution that allows developers to rapidly scale and integrate real-time systems that are disparate or geographically dispersed. It scales *RTI Connext® DDS* applications across domains, LANs and WANs, including firewall and NAT traversal.

It also supports DDS-to-DDS bridging by allowing you to make transformations in the data along the way. This allows unmodified DDS applications to communicate even if they were developed using incompatible interface definitions. This is often the case when integrating new and legacy applications or independently developed systems. Using *RTI Routing Service Adapter SDK*, you can extend *Routing Service* to interface with non-DDS systems using off-the-shelf or custom-developed adapters.

Traditionally, *Connext DDS* applications can only communicate with applications in the same domain. With *Routing Service*, you can send and receive data across domains. You can even transform and filter the data along the way! Not only can you change the actual data values, you can change the data's type. So the sending and receiving applications don't even need to use the same data structure. You can also control which data is sent by using allow and deny lists.

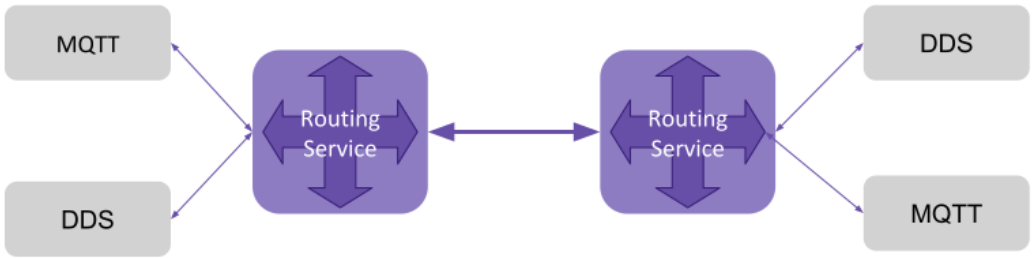


Figure 1.1: Routing Service Overview

Simply set up *Routing Service* to pass data from one domain to another and specify any desired data filtering and transformations. No change are required in the *Connext DDS* applications.

Key benefits of *Routing Service*:

- It can significantly reduce the time and effort spent integrating and scaling *Connext DDS* applications across Wide Area Networks and Systems-of-Systems.

- With *Routing Service*, you can build modular systems out of existing systems. Data can be contained in private domains within subsystems and you can designate that only certain "global topics" can be seen across domains. The same mechanism controls the scope of discovery. Both application-level and discovery traffic can be scoped, facilitating scalable designs.
- *Routing Service* provides secure deployment across multiple sites. You can partition networks and protect them with firewalls and NATS and precisely control the flow of data between the network segments.
- It allows you to manage the evolution of your data model at the subsystem level. You can use *Routing Service* to transform data on the fly, changing topic names, type definitions, QoS, etc., seamlessly bridging different generations of topic definitions.
- *Routing Service* provides features for development, integration and testing. Multiple sites can each locally test and integrate their core application, expose selected topics of data, and accept data from remote sites to test integration connectivity, topic compatibility and specific use-cases.
- It connects remotely to live, deployed systems so you can perform live data analytics, fault condition analysis, and data verification.
- *RTI Routing Service Adapter SDK* allows you to quickly build and deploy bridges to integrate DDS and non-DDS systems. This can be done in a fraction of the time required to develop completely custom solutions. Bridges automatically inherit advanced DDS capabilities, including automatic discovery of applications; data transformation and filtering; data lifecycle management and support across operating systems; programming languages and network transports.

1.1 How To Read This Manual

The content of this manual assumes you are familiar with *Connext DDS* concepts. While you can read any section independently, if you are new to *Routing Service* we recommend starting with the *Tutorials* to get an overview of what this application can do.

Then read the *Core Concepts* for deeper knowledge of *Routing Service* specific concepts. You can then refer to the *Configuration* to start defining and customizing your *Routing Service*.

You can read any of the other sections as you see fit based on what your application or system needs are.

1.2 Paths Mentioned in Documentation

This documentation refers to:

- `<NDDSHOME>` This refers to the installation directory for *Connext DDS*.

  The default installation paths are:

  – macOS® systems: `/Applications/rti_connext_dds-version`

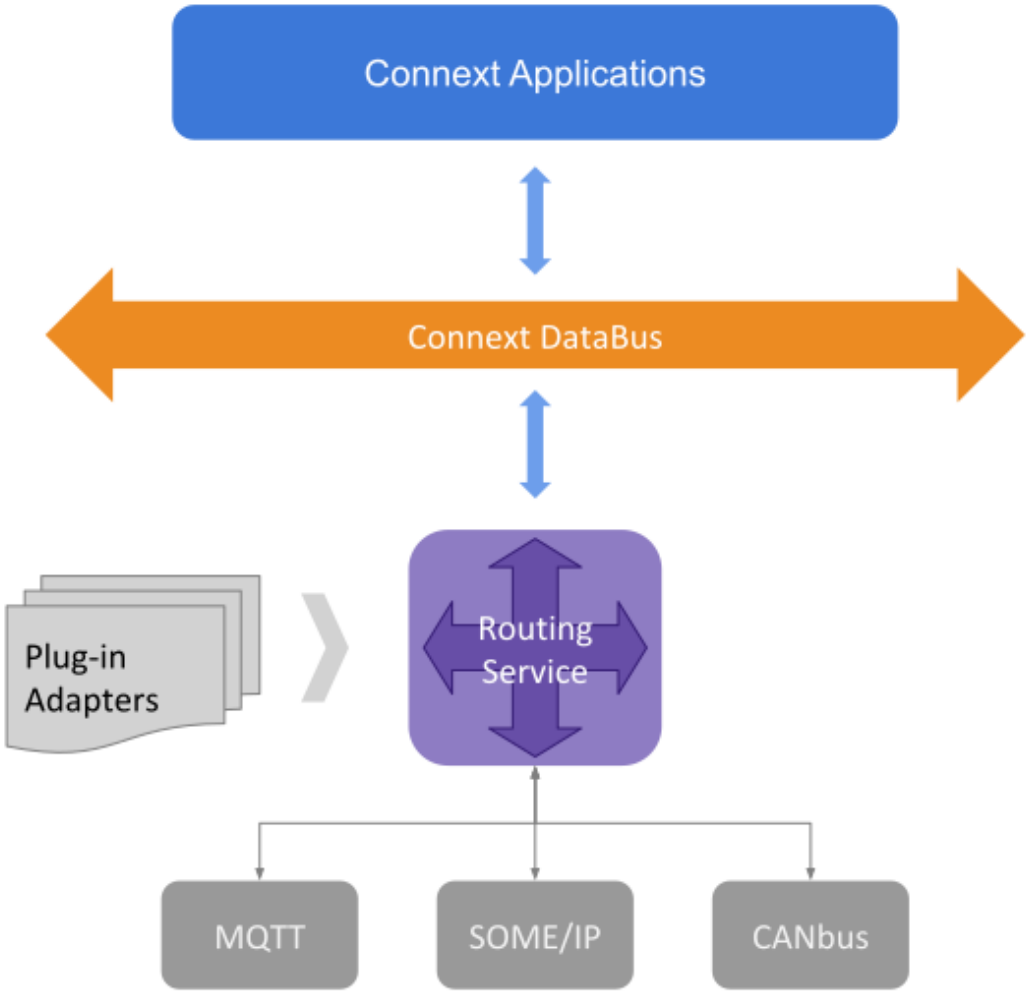


Figure 1.2: Quickly build and deploy bridges between natively incompatible protocols and technologies using *Connext DDS*

  - UNIX®-based systems, non-root user: `/home/your user name/rti_connext_dds-version`
  - UNIX-based 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`
  - UNIX-based 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 10 systems, the folder is `C:\Users\your user name\Documents`.

1.3 Files Mentioned in Documentation

Table 1.1: Files mentioned in the documentation

File	Description
`ServiceCommon.idl`	Definitions of infrastructure types.
`ServiceAdmin.idl`	Definition of remote administration types.
`RoutingServiceMonitoring.idl`	Definition of monitoring types specific to *Routing Service*.

Chapter 2

Core Concepts

This section aims to provide a deeper understanding of the *Routing Service* architecture and give you the required insight to configure and use it effectively.

You will learn about:

- *Application resource model*: Gives you a full picture of all the elements that compose *Routing Service*, including details about their relationships with the pluggable components and their lifecycle.
- *Builtin plugins*: Describes the builtin pluggable components that are part of the *Routing Service* module.

2.1 Resource Model

In this section you will learn the details of the *Routing Service* application resource model (see Section 12.1). It describes all the different *resource classes*, their functions and responsibilities, and their relationships with other resources.

Figure 2.1 shows a high-level view of the main classes that comprise the application resource model.

There are two main logical planes, each addressing orthogonal sets of capabilities:

- **Data Plane**: Set of resources associated with data flow, both user data and meta-data. A resource in this plane is also known as an *entity*. The data provision and processing is performed using *plugins* (see Section 7 for an overview of the list of available plugins).
- **Control Plane**: Set of resources associated with service monitoring and administration. These are the resources in charge of providing monitoring information and run-time administration of the resources from the data plane.

An alternative representation of the resource module is shown in Figure 2.2.

The next sections describe each entity with detail. The documentation for each entity will provide:

- A Description of the role and responsibility of the entity within *Routing Service*.
- The relationship, if any, with plugin components. This part will give you an understanding of how *Routing Service* achieves custom behavior.

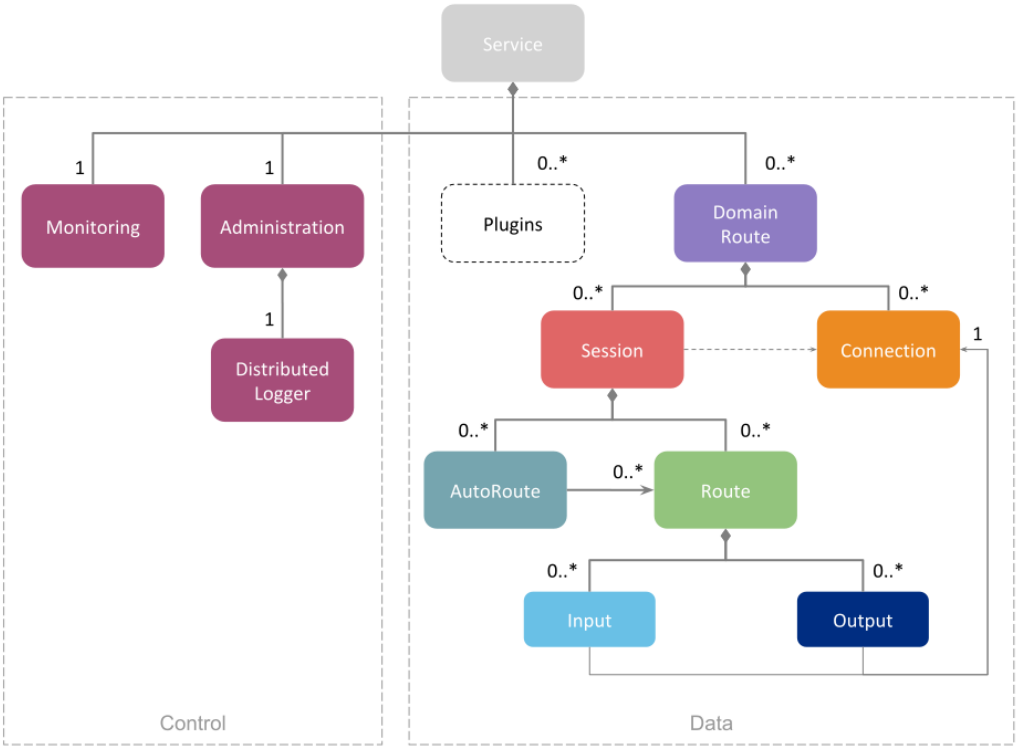


Figure 2.1: *Routing Service* Application Resource Model

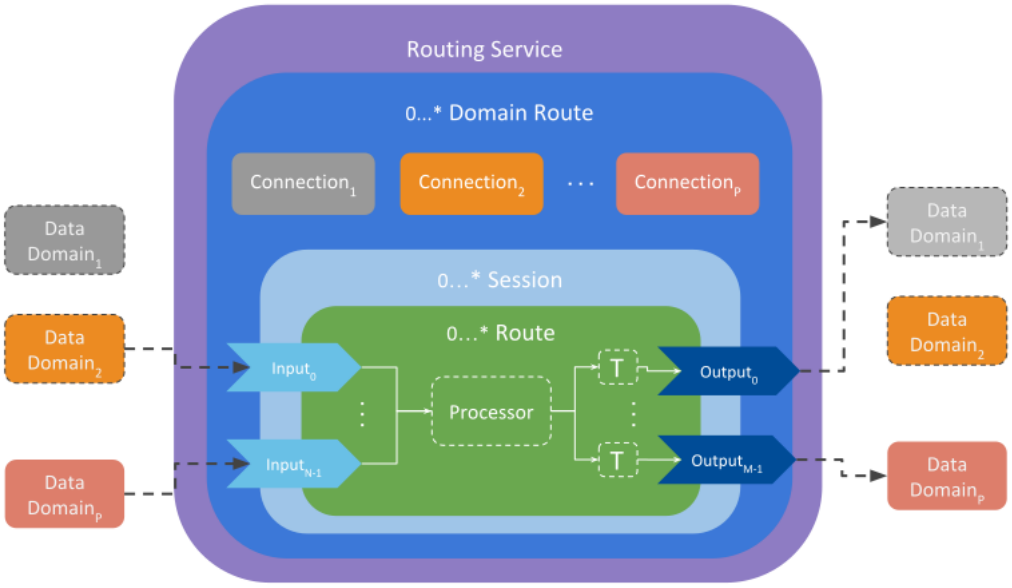


Figure 2.2: *Routing Service* Alternative representation of the Application Resource Model

- A Description of the states an entity can go through.

The next sections describe *Routing Service* from a generic point of view, independently of the *Adapter* (or any other type of plugin) that is used. To read more about how DDS is integrated with *Routing Service*, please see the (Section 2.2.1). It's recommended though that you still review the general model for a solid understanding of *Routing Service*.

2.1.1 Directory

Table 2.1 provides a resource directory with quick links to access different types of information for each resource or entity.

Table 2.1: Resource Reference

Resource	Configuration	Administration	Monitoring
Service	Section 4.5.1	Section 5.2.2	Section 6.2.1
DomainRoute	Section 4.5.4	Section 5.2.3	Section 6.2.2
Connection	Section 4.5.4	Section 5.2.4	Section 6.2.2
Session	Section 4.5.5	Section 5.2.5	Section 6.2.3
Route	Section 4.5.6	Section 5.2.7	Section 6.2.5
Input	Section 4.5.7	Section 5.2.8	Section 6.2.6
Output	Section 4.5.7	Section 5.2.8	Section 6.2.6

2.1.2 Service

The *Service* is the top-level resource. The *Service* is the entity that encapsulates all the resources needed for the operation of both the control and data planes. Typically, a *Service* refers to an execution of *Routing Service*.

In the control plane, the *Service* is composed of the *Monitoring* and *Administration* resources, which are optionally available sub-services. These components are described in Section 6 and Section 5, respectively.

In the data plane, the *Service* is composed of a collection of user plugins instances and a collection of *DomainRoutes*.

Plugin Interaction

The *Service* is responsible for loading and owning any of the *plugins* that you can provide through the Software Development Kit (see Section 7). Figure 2.3 shows the relationship between the *Service* and the plugin objects.

See Section 12.4 for more information about plugin management.

Service States

A *Service* can be in one of the states listed in Table 2.2.

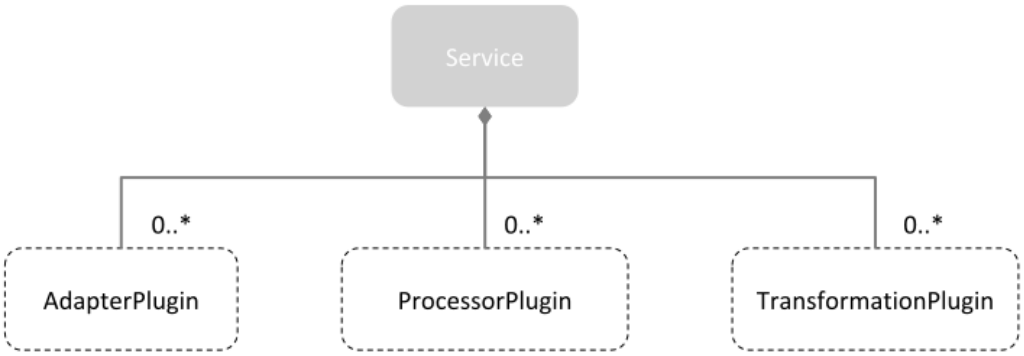


Figure 2.3: *Routing Service* composed of different plugins

Table 2.2: Service States

State	Description	Trigger	Plugin callback
`ENABLED`	A *Service* object has loaded the specified service configuration. Monitoring and Administration services are started if they are enabled in the configuration.	• User runs the *Routing Service* application either using the pre-built executable or through the Service API (see Section 3). • Remote command	N/A
`STARTED`	A *Service* object has created all the underlying resources, including creating and starting all the contained *DomainRoutes*, as specified in the configuration. Additionally, the service discovery thread (SDT) is also started. The SDT sets the context to read the data from the builtin input/output *stream* discovery *StreamReaders* Plugin configurations are validated but the libraries are loaded and instances created lazily when they are first needed.	• User spawns the entity • Remote command	N/A

Continued on next page

Table 2.2 – continued from previous page

State	Description	Trigger	Plugin callback
`STOPPED`	A *Service* object has deleted all the resources created during the start phase: the service discovery thread and *DomainRoutes* are deleted. Additionally, any plugin instances are deleted.	• User deletes the entity • Remote command	• `AdapterPlugin::delete` • `ProcessorPlugin::delete` • `TransformationPlugin::delete`
`DISABLED`	A *Service* object has deleted all the resources created during the enable phase. Entering this state occurs only temporarily while the *Service* object is being deleted.	• User shutdowns the entity • Remote command	N/A

2.1.3 DomainRoute

A *DomainRoute* defines a collection of independent data domains (such as DDS, MQTT, AMQP, etc.), each modeled as a *Connection*. It's also composed of a collection of *Sessions*.

DataReader States

A *DomainRoute* can be in one of the states listed in Table 2.3.

Table 2.3: *DataReader* states

State	Description	Trigger	Plugin callback
`ENABLED`	A *DomainRoute* object has created all the underlying *Connections* and *Sessions* as indicated in the configuration.	• Service starts (Section 2.1.2) • Remote command	N/A

Continued on next page

Table 2.3 – continued from previous page

State	Description	Trigger	Plugin callback
`STARTED`	A *DomainRoute* object has enabled all the contained *Connections* and started all the contained *Sessions*. The *DomainRoute* is attached to the service discovery thread and may start processing *stream* discovery data.	• Service starts (Section 2.1.2) • Remote command	N/A
`STOPPED`	A *DomainRoute* object has stopped all *Sessions* and disabled all the *Connections*. The *DomainRoute* is detached from the service discovery thread.	• Service stops (Section 2.1.2) • Remote command	N/A
`DISABLED`	A *DomainRoute* object has deleted all the underlying *Connections*. Entering this state occurs only temporarily while the *DomainRoute* object is being deleted.	• Stop *DataReader*	N/A

2.1.4 Connection

A *Connection* defines an access point to a specific data domain. The access to a data domain is provided through an instance of an *Adapter* plugin, which is specified in the configuration (See Table 4.7 and Table 4.8). For example, the associated *Adapter* plugin implementation could provide a connection to an HTTP Server through an HTTP Client, or a logical connection to a DDS Domain through a *DomainParticipant*.

The *Connection* is also responsible for tracking all the *stream* information that is provided by the underlying *input and output stream discovery StreamReaders*. The *Connection* gets notified about new or disposed *streams* and propagates this information downstream to the *Routes* and *AutoRoutes*, which will process and generate events accordingly.

Note: A *DomainParticipant* is a special type of *Connection* that represents an instance of a `DdsConnection`. For this case, special custom tags are available that facilitate configuring the `DdsConnection`.

Plugin Interaction

Figure 2.4 shows the relationship with the plugin objects. A *Connection* shall hold one, and only one, `adapter::Connection` object.

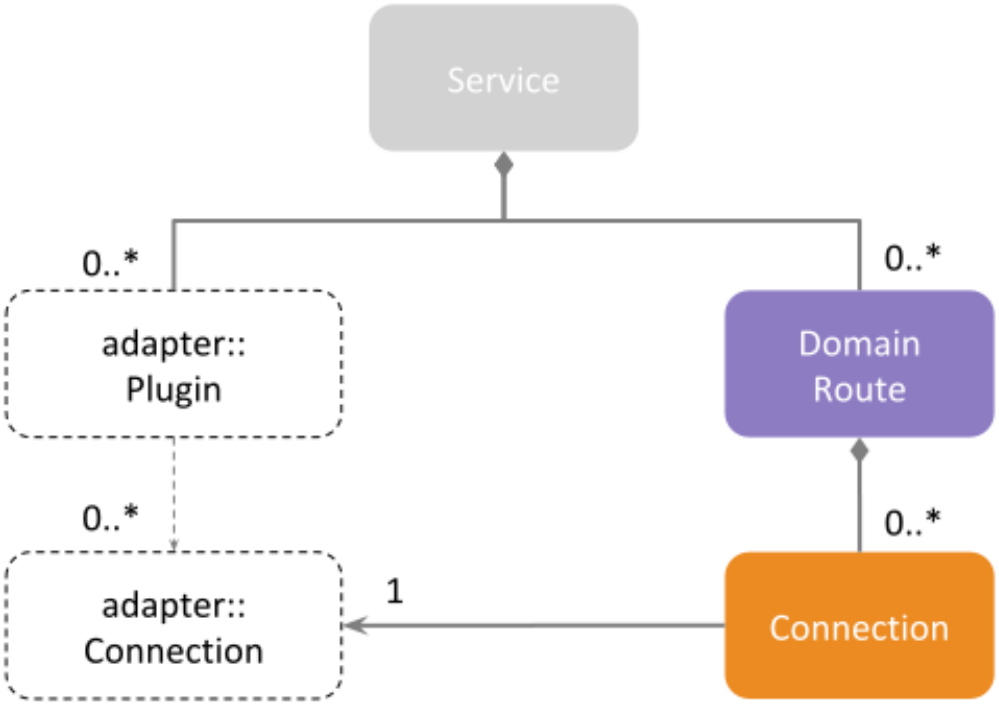


Figure 2.4: Relationship of plugins with a *Connection*

Connection States

A *Connection* can be in one of the states listed in Table 2.4.

Table 2.4: *Connection* states

State	Description	Trigger	Plugin callback
`ENABLED`	A *Connection* object has created the underlying *Adapter* connection object.	• *DomainRoute* starts (Section 2.1.3)	• `AdapterPlugin::new` (only once for each plugin class) • `AdapterPlugin::create_connection`

Continued on next page

Table 2.4 – continued from previous page

State	Description	Trigger	Plugin callback
DISABLED	A *Connection* object has deleted the *Adapter* connection object it holds.	• *Domain-Route* stops (Section 2.1.3)	AdapterPlugin::delete_connection

Type Registration

The *Connection* is the entity where *type registration* takes place. A *Connection* keeps a list of registered types, where each entry in the list contains:

- **registered type name**: Unique name used to identify and register a concrete type within the *Connection*.
- **type representation**: In-memory structure that describes the type itself. The type representation is adapter-dependent and *Routing Service* assumes `TypeCode` as default type representation for types.

A type is associated with a *stream* and its registration is required in order to create *StreamReaders* and *StreamWriters*. A type can be registered in two ways:

- Through *stream* discovery information, provided by the builtin stream discovery *StreamReaders*. On stream discovery, the associated information contains the registered name and the representation for a type.
- Through XML *Connection* configuration (see Section 4.5.7). A type definition is provided in XML and the *Routing Service* parser will generate a `TypeCode` from it. *Connection* configuration can then reference this XML type definition to register it.

2.1.5 Session

A *Session* defines a collection of *Routes* and *AutoRoutes*. It also defines a multi-threaded safe context for *Route* event processing.

Events from a *Route* are processed sequentially within the same *Session*. A *Route* event is processed by a single thread at a time. That is, the same route cannot be processed concurrently. However, within a *Session*, different *Routes* that can be processed concurrently, as many as the number of threads available within the *Session*.

Figure 2.5 shows the event processing mechanism. Consider a *Session* with a pool of `N` threads and composed of `P` *Routes*.

- *Session* threads are idle waiting for *Routes* to become active. An active *Route* is one that has events pending processing.
- Once an active *Route* is selected for processing, all the pending events at that time will be consumed sequentially one after the other (see Section 2.1.6 for information about route processing). To prevent starvation, new events arriving will be deferred for the next selection cycle.

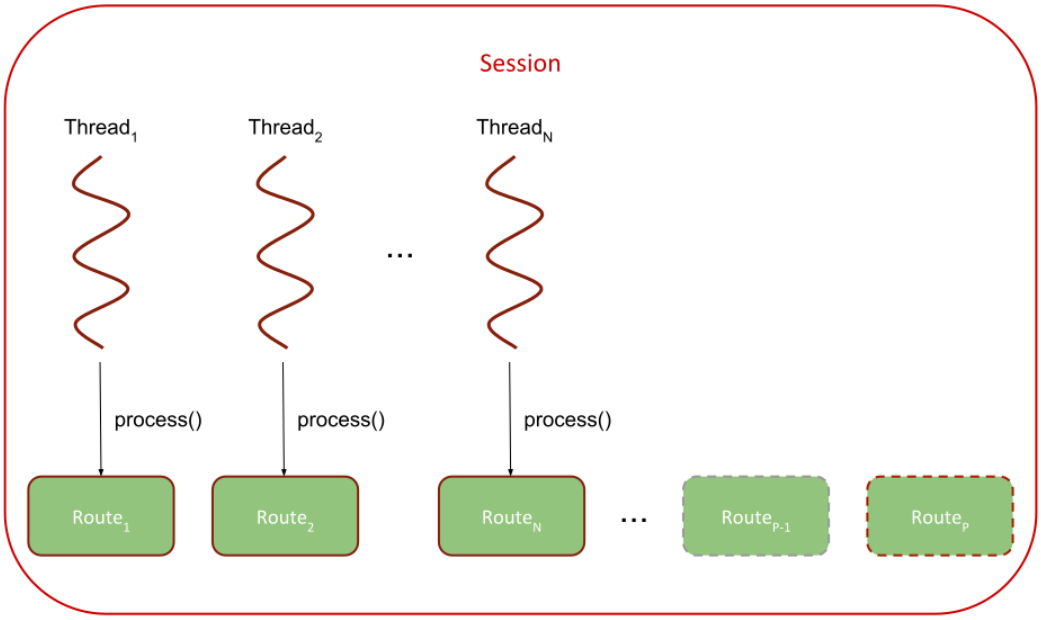


Figure 2.5: Processing mechanism of *Routes* within a *Session*

- A *Session* selects *Routes* for processing in a round-robin fashion, following the same order as they are defined in the *Session* configuration. At a maximum only `N` *Routes* can be processed concurrently. Remaining active *Routes* will wait until a thread becomes available.

Figure 2.5 shows a *Session* concurrently processing `N` active *Routes*. Other remaining `P-N` *Routes*, such as Route_{P}, are active and waiting for a thread to become available; $\text{Route}_{\text{P-1}}$ is not active (no pending events).

Plugin Interaction

Figure 2.6 shows the relationship with the plugin objects. A *Session* shall hold one `adapter::Session` object for each *Connection* in the parent *DomainRoute*.

Session States

A *Session* can be in one of the states listed in Table 2.5.

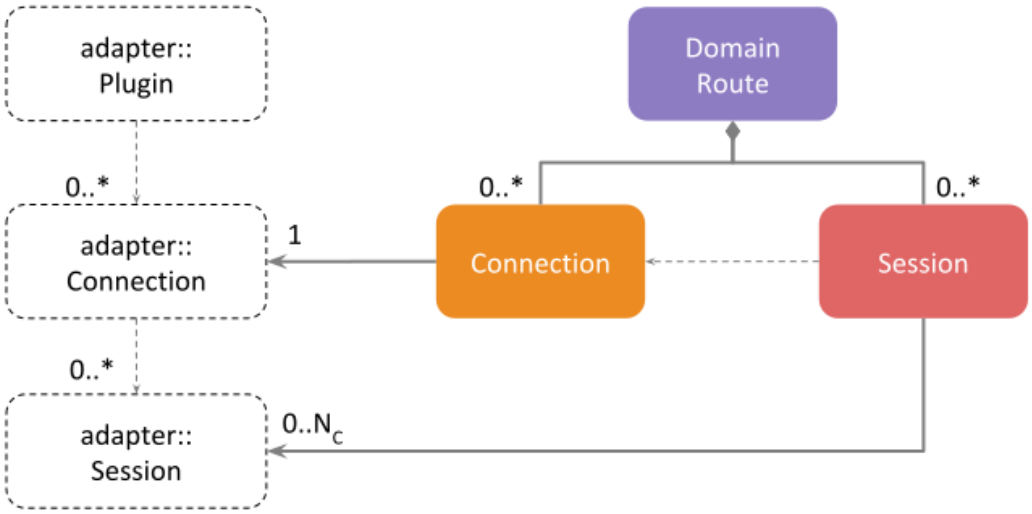


Figure 2.6: Relationship of plugins with a *Session*

Table 2.5: *Session* states

State	Description	Trigger	Plugin callback
`ENABLED`	A *Session* object has created all the underlying `adapter::Session` objects. It has also created all the *AutoRoutes* and *Routes* that are defined in the configuration.	• *DomainRoute* starts (Section 2.1.3) • Remote command	`Connection::create_session`
`STARTED`	A *Session* object has started the thread pool, and enabled all the underlying *AutoRoutes* and *Routes*. In this state, the *Session* is actively processing *Route* events.	• *DomainRoute* starts (Section 2.1.3) • Remote command	N/A

Continued on next page

Table 2.5 – continued from previous page

State	Description	Trigger	Plugin callback
`STOPPED`	A *Session* object has stopped the thread pool, and disabled all the underlying *AutoRoutes* and *Routes*.	• *DomainRoute* stops (Section 2.1.3) • Remote command	N/A
`DISABLED`	A *Session* object has deleted all the `adapter::Session` objects it holds.	• *DomainRoute* stops (Section 2.1.3) • Remote command	`Connection::delete_session`

2.1.6 Route

A *Route* defines a processing unit for data streams. A *Route* is composed of `N` *Inputs* and `M` *Outputs*, each referencing any of the *Connections* defined as part of the parent *DomainRoute*.

A *Route* generates certain events that are processed safely and serially within one of the threads from the parent *Session*. *Route* events are processed through a pluggable *Processor*.

Note: A *TopicRoute* is a special type of *Route*. All its *Inputs* and *Outputs* are tied to the builtin DDS *Adapter*. For this case, special and custom tags are available that facilitate configuring the *TopicRoute*.

Plugin Interaction

Figure 2.7 shows the relationship with the plugin objects. A *Route* shall hold one *Processor* object, which will receive the notifications of the events affecting the owner *Route*.

For more information about the *Processor* behavior and *Route* events, see the main page of API documentation (Section 7).

Route States

A *Route* state machine is shown in Figure 2.8.

Table 2.6 shows all the states a *Route* can enter.

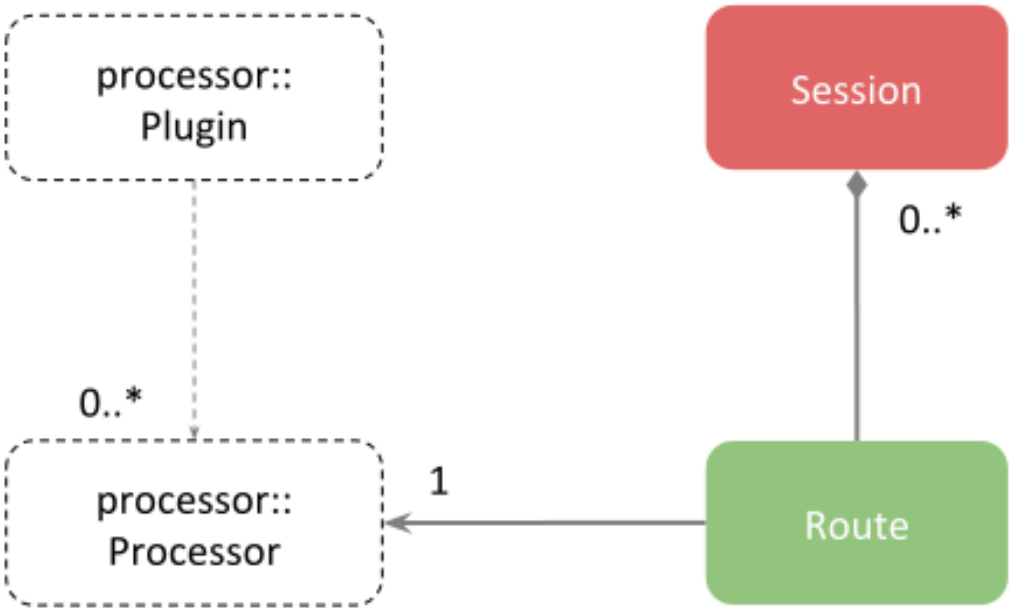


Figure 2.7: Relationship of plugins with a *Route*

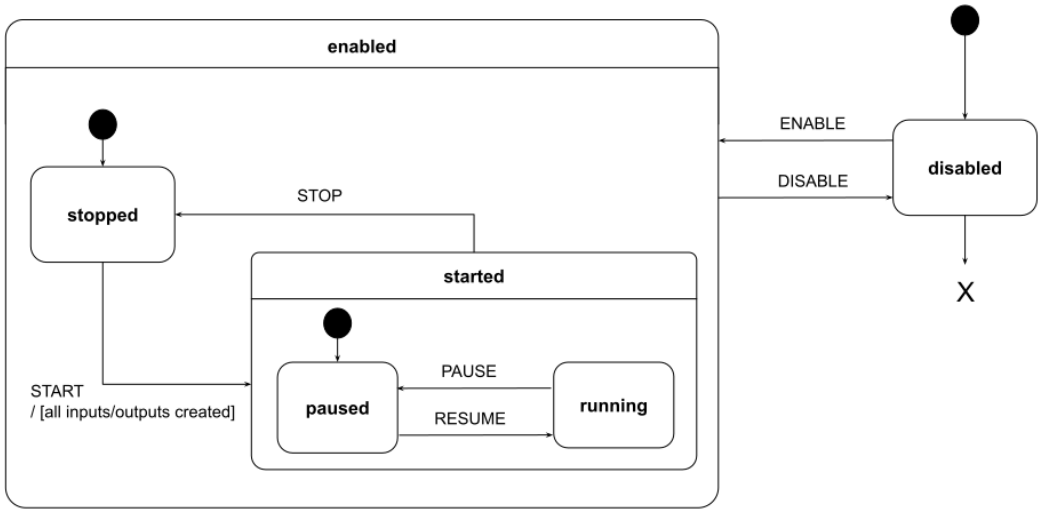


Figure 2.8: *Route* state machine

Table 2.6: *Route* states

State	Description	Trigger	Plugin callback
`ENABLED`	A *Route* has created the underlying *Processor*. The *Route* is attached to the parent *Session* and is receiving event notifications.	• *Session* starts (Section 2.1.5) • Remote command	• `ProcessorPlugin::new` (only once for each plugin class) • `ProcessorPlugin::create_processor`
`DISABLED`	A *Route* has deleted the underlying *Processor*. The *Route* is detached from the parent *Session* so no events are notified.	• *Session* stops (Section 2.1.5) • Remote command	`ProcessorPlugin::delete_processor`
`STARTED`	A *Route* has enabled all its *Inputs* and *Outputs*.	• *Session* starts (Section 2.1.5) • Enable *Input* (Section 2.1.8) or *Output* (Section 2.1.9) • Remote command	`Processor::on_route_event`

Continued on next page

Table 2.6 – continued from previous page

State	Description	Trigger	Plugin callback
`STOPPED`	A *Route* has disable at least one of its *Inputs* and *Outputs*.	• *Session* stops (Section 2.1.5) • Disable *Input* (Section 2.1.8) or *Output* (Section 2.1.9) • Remote command	`Processor::on_route_event`
`RUNNING`	A *Route* is ready to process data stream related events. These include: • `DATA_ON_INPUTS` • `PERIODIC_ACTION`	• *Session* starts (Section 2.1.5) • Enable *Input* (Section 2.1.8) or *Output* (Section 2.1.9) • Remote command	• `Processor::on_route_eve` • `StreamReader::read` • `StreamReader::return_lo` • `Transformation::transfo` • `Transformation::return_` • `StreamWriter::write`
`PAUSED`	A *Route* is temporarily suspending the processing of data stream related events.	• *Session* stops • Disable *Input* (Section 2.1.8) or *Output* (Section 2.1.9) • Remote command	`Processor::on_route_event`

2.1.7 AutoRoute

An *AutoRoute* represents a factory of single-input single-output *Routes*. An *AutoRoute* creates *Routes* based on a *name filter* criteria that matches the name or type of a *stream*.

An *AutoRoute* creates a *Route* for each pair:

$$[S_m, T_n]$$

where S_m and T_n are the name for the *stream* `m` and the name of the type `n`, respectively. The generation of a *Route* occurs only on the event of a newly discovered *stream*. The resulting *Route* has a single *Input* and a single *Output*, both for the same *stream* name and type.

The created *Route* executes within the context of the parent *Session* of the *AutoRoute*. Figure 2.9 illustrates this relationship.

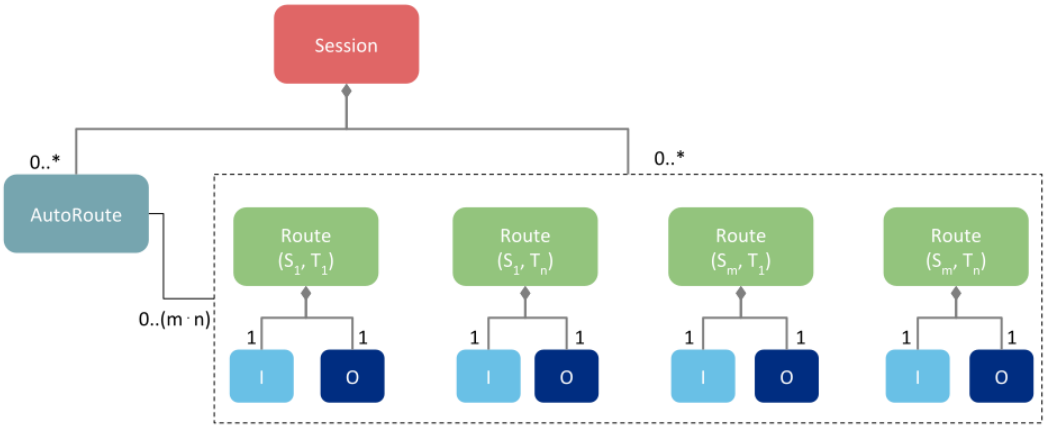


Figure 2.9: *AutoRoute* as a map of *Routes* keyed by stream and type names

The *AutoRoute* creates a *Route* only if it has not previously matched the S_m and T_n pair. *AutoRoutes* never delete the created *Route*, independently of whether the matching *streams* are disposed or not.

Note: An *AutoTopicRoute* is a special type of *AutoRoute* whose *Inputs* and *Outputs* are tied to the builtin DDS *Adapter*. For this case, special and custom tags are available that facilitate configuring the *AutoTopicRoute*.

AutoRoute States

An *AutoRoute* can be in one of the states listed in Table 2.7.

Table 2.7: *AutoRoute* states

State	Description	Trigger	Plugin callback
`ENABLED`	*AutoRoute* object is read to start matching *streams* and create *Routes*. Previously discovered streams are matched retroactively.	• *Session* starts (Section 2.1.5) • Remote command	N/A
`STARTED`	This state is equivalent to the `ENABLED` state and the transition is automatic upon enabling. This state is added for consistency with the other entities.	• Enable *AutoRoute*	N/A
`STOPPED`	This state is equivalent to the `DISABLED` state and the transition is automatic upon disabling. This state is added for consistency with the other entities.	• Disable *AutoRoute*	N/A
`DISABLED`	*AutoRoute* stops matching all newly discovered *streams*. All the *Routes* created from this *AutoRoute* are deleted.	• *Session* stops (Section 2.1.5)	N/A

2.1.8 Input

An *Input* is responsible for obtaining data associated with a specific *stream* uniquely identified by its name and type. An *Input* must reference an existing *Connection* within the parent *DomainRoute*. The referenced *Connection* determines the data domain where the *Input* will obtain data.

An *Input* has scope only within the parent *Route*. It cannot be shared in other *Routes*. If another *Route* requires accessing the same data stream, a new *Input* shall be defined within such *Route*.

Plugin Interaction

Figure 2.10 shows the relationship with the plugin objects. An *Input* shall hold one, and only one, `adapter::StreamReader` object.

Input States

An *Input* can be in one of the states listed in Table 2.8.

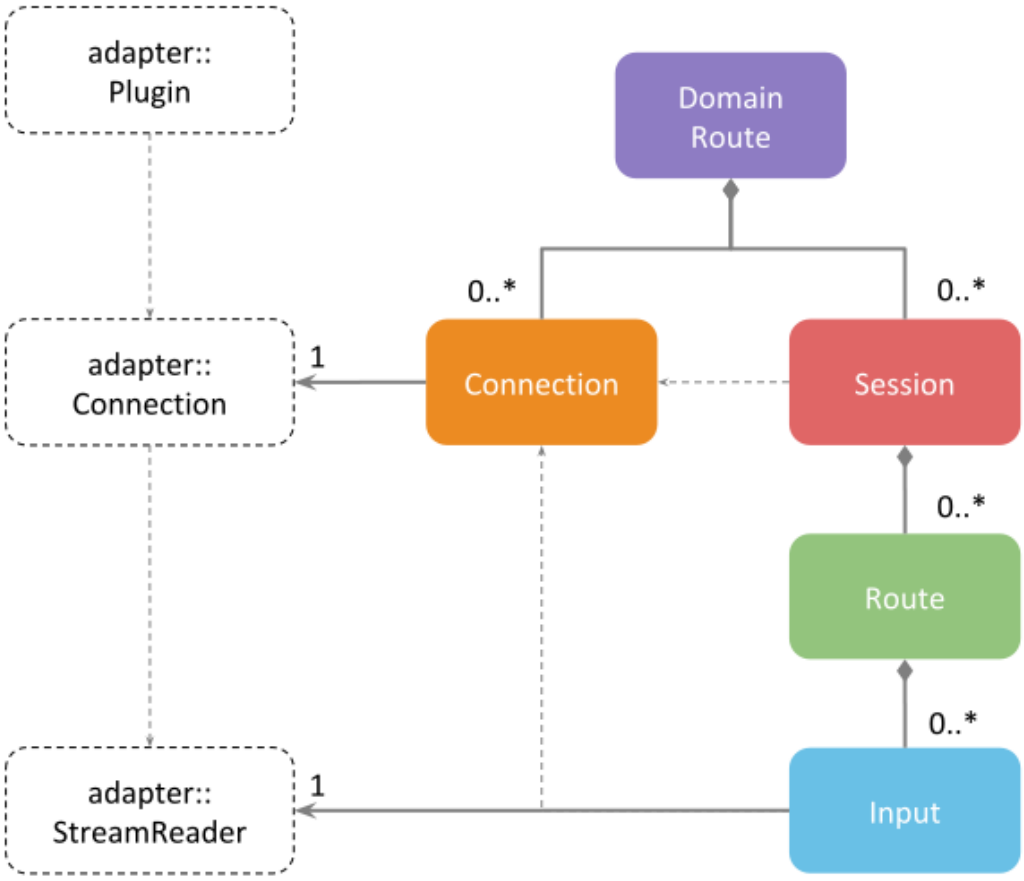


Figure 2.10: Relationship of plugins with an *Input*

Table 2.8: *Input* states

State	Description	Trigger	Plugin callback
`ENABLED`	*Input* has created its underlying *StreamReader* and it's ready to read data.	The following two conditions shall be met: • Matching type is available • Creation mode condition becomes true	• `Connection::create_stream_reade` • `Processor::on_route_event`
`STARTED`	This state is equivalent to the `ENABLED` state and the transition is automatic upon enabling. This state is added for consistency with the other entities.	• Enable *Input*	N/A
`STOPPED`	This state is equivalent to the `DISABLED` state and the transition is automatic upon disabling. This state is added for consistency with the other entities.	• Disable *Input*	N/A
`DISABLED`	*Input* has deleted its underlying *StreamReader* and can no longer read data.	Creation mode condition becomes false	• `Connection::delete_stream_reade` • `Processor::on_route_event`

2.1.9 Output

An *Output* is responsible for writing data associated with a specific *stream* uniquely identified by its name and type. An *Output* must reference an existing *Connection* within the parent *DomainRoute*. The referenced *Connection* determines the data domain where the *Output* will provide data.

An *Output* has scope only within the parent *Route*. It cannot be shared in other *Routes*. If another *Route* requires access to the same data stream, a new *Output* shall be defined within such *Route*.

Plugin Interaction

Figure 2.11 shows the relationship with the plugin objects. An *Output* shall hold one, and only one, `adapter::StreamWriter` object. Optionally, an *Output* can hold one, and only one, `transformation::Transformation` object.

The *Output* provides the data to a domain by calling the `StreamWriter::write` operation. If a *Transformation* is present, the `Transformation::transform` operation is called right before writing on the *StreamWriter*, followed by a `Transformation::return_loan` right after.

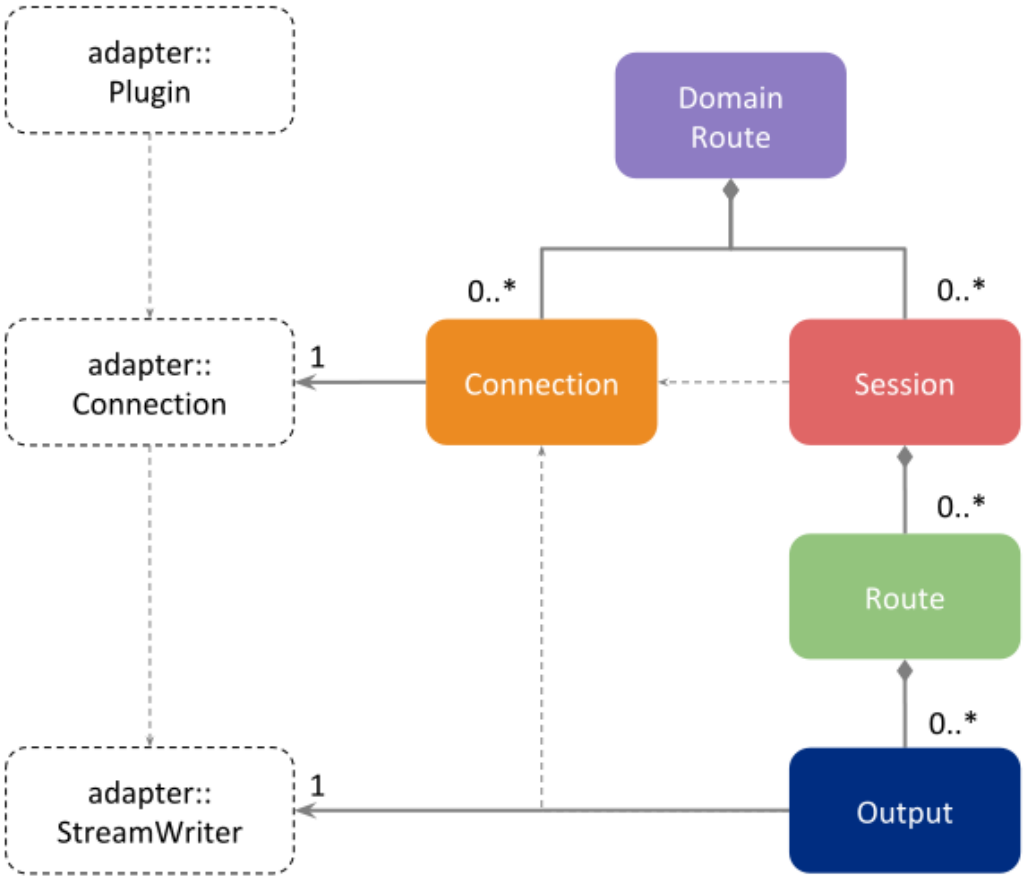


Figure 2.11: Relationship of plugins with an *Output*

Output States

An *Output* can be in one of the states listed in Table 2.9.

Table 2.9: *Output* states

State	Description	Trigger	Plugin callback
`ENABLED`	*Output* has created its underlying *StreamWriter* and it's ready to write data.	The following two conditions shall be met: • Matching type is available • Creation mode condition becomes true	• `Connection::create_stream_write` • `Processor::on_route_event` • `TransformationPlugi` (only once for each plugin class) • `TransformationPlugi`
`STARTED`	This state is equivalent to the `ENABLED` state and the transition is automatic upon enabling. This state is added for consistency with the other entities.	• Enable *Output*	N/A
`STOPPED`	This state is equivalent to the `DISABLED` state and the transition is automatic upon disabling. This state is added for consistency with the other entities.	• Disable *Output*	N/A
`DISABLED`	*Output* has deleted its underlying *StreamWriter* and can no longer write data.	Creation mode condition becomes false	• `Connection::delete_stream_write` • `TransformationPlugi` • `Processor::on_route_event`

2.2 Builtin plugins

Builtin plugins come pre-registered in memory within *Routing Service*. Any configurable aspects are available through dedicated special tags for enhanced usability.

2.2.1 DDS Adapter

This is an *Adapter* implementation that provides access to DDS domains. Figure 2.12 shows the architecture of the DDS *Adapter*.

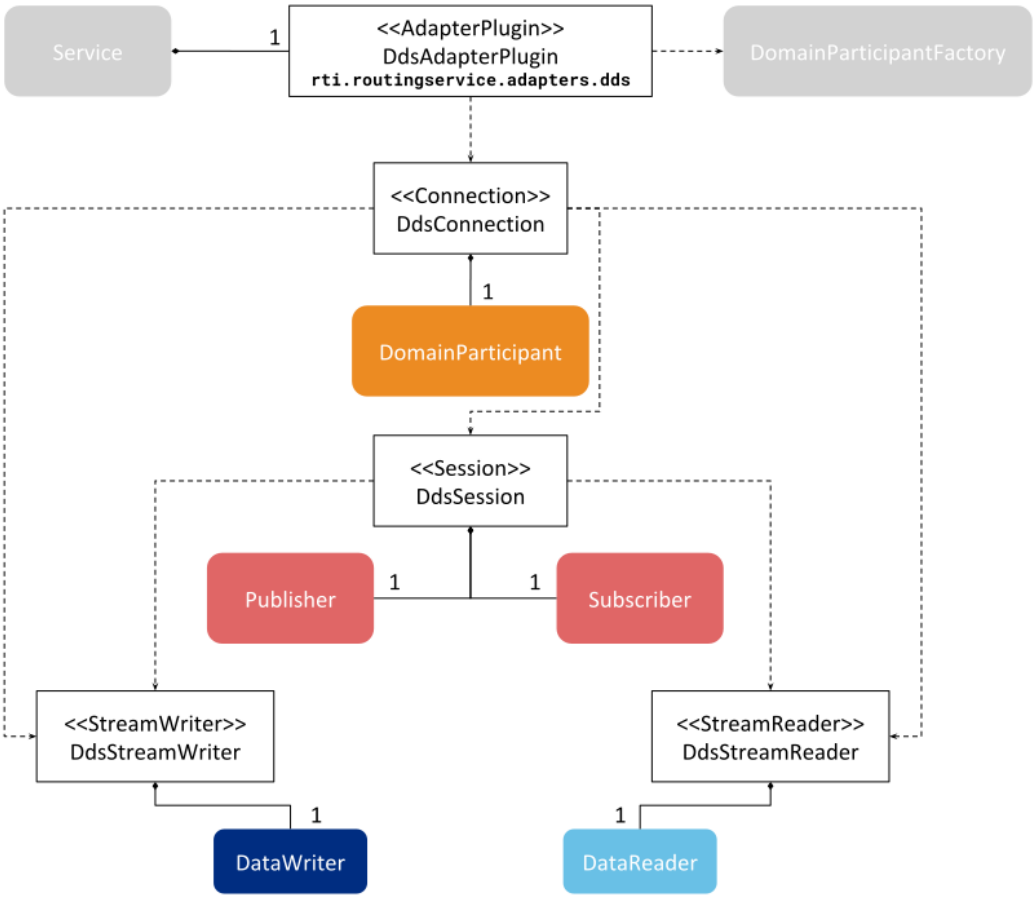


Figure 2.12: DDS *Adapter* architecture

Most of the use cases expect to have DDS as the main data domain in the user data plane. For this reason, you will find that *Routing Service* specializes some entities so that they are directly associated with DDS. These entities are:

- *Participant*
- *AutoTopicRoute*
- *TopicRoute*
- *DdsInput*

- *DdsOutput*

These entities are equivalent to the generic entities shown in Figure 2.1 except that the *Adapter* entity they enclose is created from the builtin DDS *Adapter* (Section 2.2.1). Figure 2.13 shows the DDS specialization of the generic resource model.

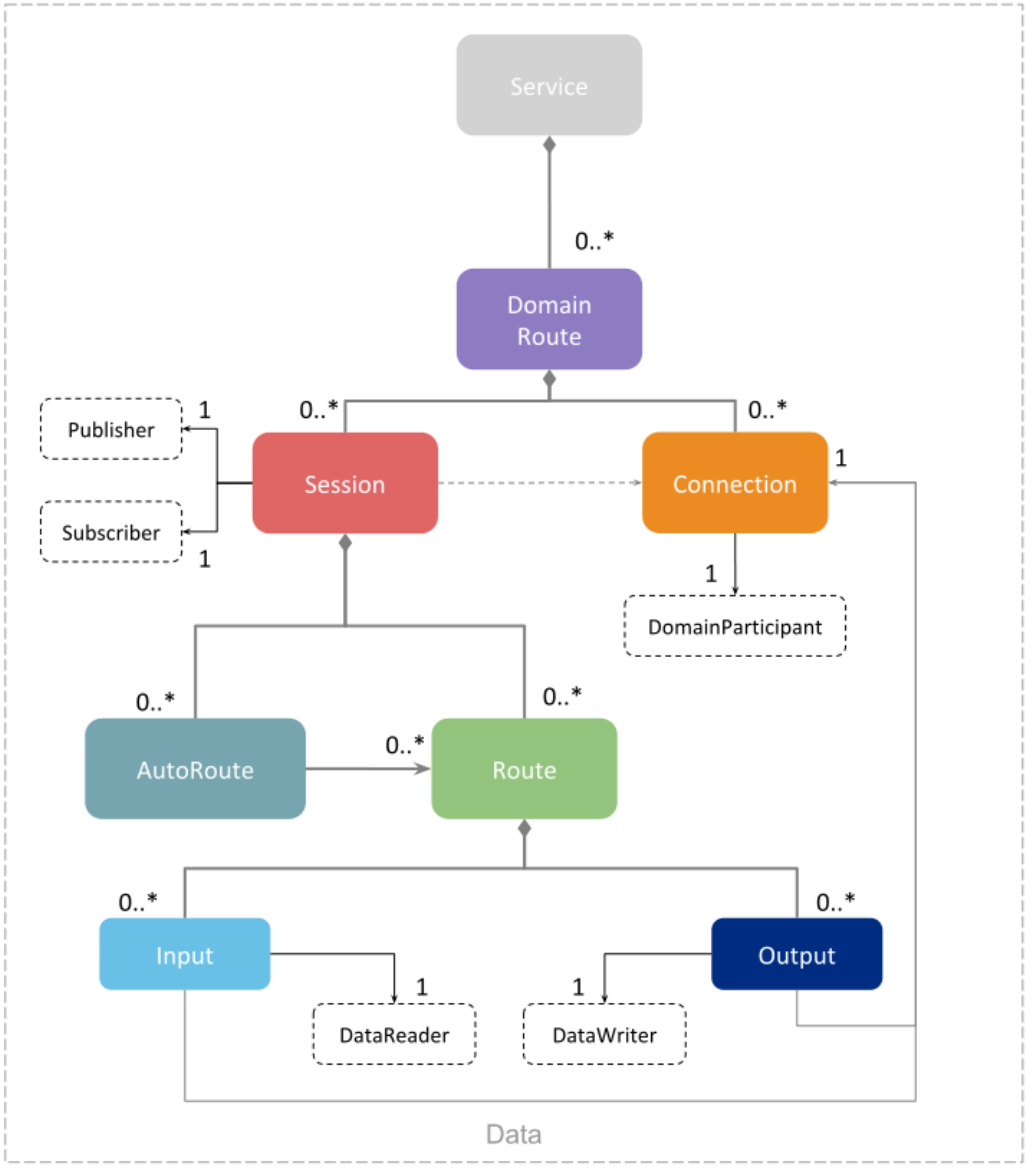


Figure 2.13: *Routing Service* DDS Application Resource Model

DDS AdapterPlugin

The `DdsAdapter` is an implementation of the *Adapter* interface. It's responsible for creating DDS *Connections*.

Table 2.10: DDS *Adapter*

Mapping	Configuration Tag
It uses the *DomainParticipantFactory* to create the participants needed by each DDS *Connection*	`<participant_factory_qos>` (only in `USER_QOS_PROFILES.xml`)

DDS Connection

The `DdsConnection` is an implementation of the *Connections* interface. It is responsible for joining to a specific DDS Domain. It's also the factory for creating DDS *Sessions*, *StreamReaders* and *StreamWriters*.

The `DdsConnection` relies on the `DdsAdapter` for creating *DomainParticipants*. This class creates the *Topics* associated with the *DataReaders* and *DataWriters* it also creates.

Table 2.11: DDS *Connection*

Mapping	Configuration Tag
Composed of only one *DomainParticipant*	`<domain_route>/<participant>` (see Table 4.8)

DDS Session

The `DdsSession` is an implementation of the *Session* interface. It's responsible for creating *Subscribers* and *Publishers*.

Table 2.12: DDS *Session*

Mapping	Configuration Tag
Composed of only one *Publisher* and one *Subscriber*	`<session>/<subscriber_qos>` and `<session>/<publisher_qos>` (see Table 4.9)

Note that, as explained in Section 2.1.5, a new `DdsSession` object is instantiated for each pair `<session>` and `<participant>` element within the parent *DomainRoute*.

DDS StreamReader

The `DdsStreamReader` is an implementation of the *StreamReader* interface. It's responsible for reading data from a *Topic* and providing it to the parent *Route*, which is in charge of processing it through the installed *Processor*.

Table 2.13: DDS *StreamReader*

Mapping	Configuration Tag
Composed of only one *DataReader*	`<route>/<dds_input>` and `<topic_route>/<input>` (see Table 4.13)

The referenced DDS *Connection* and parent `<session>` determines from which *DomainParticipant*

and *Subscriber* the *DataReader* is created.

The configuration of the *Input* owning the *StreamReader* indicates:

- The referenced DDS *Connection* that contains the *DomainParticipant*
- The parent `<session>`, which along with the referenced *Connection*, determines which `DdsSession` and hence *Subscriber* is used to create the *DataReader*.
- The name of the *Topic* in the domain of the *DomainParticipant*.

DDS StreamWriter

The `DdsStreamWriter` is an implementation of the *StreamWriter* interface. It's responsible for writing data to a *Topic*. The data is provided by the parent *Route* through the installed *Processor*.

Table 2.14: DDS *StreamWriter*

Mapping	Configuration Tag
Composed of only one *DataWriter*	`<route>/<dds_output>` and `<topic_route>/<output>` (see Table 4.13)

The referenced DDS *Connection* and parent `<session>` determines from which *DomainParticipant* and *Publisher* the *DataWriter* is created.

The configuration of the *Output* owning the *StreamWriter* indicates:

- The referenced DDS *Connection* that contains the *DomainParticipant*
- The parent `<session>`, which along with the referenced *Connection* determines which `DdsSession` and hence *Publisher* is used to create the *DataWriter*.
- The name of the *Topic* in the domain of the *DomainParticipant*.

2.2.2 Forwarding Processor

This is a *Processor* implementation that forwards samples within a *Route*. The plugin registered name is reserved and has the value `rti.routingservice.RoutingProcessor`.

The functions of the builtin forwarding *Processor* are:

- Forwarding all the *live* data samples received from each *Input* to each *Output*.
- Proxying the *TopicQueries* received by the `DdsStreamWriter`, making sure all the *TopicQuery* data samples received from each *Input* are sent to the corresponding *Outputs* and final destination *DataReaders*. (see Section 9.2).

The builtin forwarding *Processor* is set by default in all *AutoRoutes* and *Routes*.

Note that if you install your own *Processor* implementation, you will override the functionality described above. In this case, even if the dedicated configuration tags are specified (such as `<topic_query_proxy>`), they will not have any effect.

Chapter 3

Usage

This chapter explains how to run *Routing Service* either from the distributed command-line executable or from a library.

3.1 Command-Line Executable

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

```
rtiroutingservice [options]
```

In this section we will see:

- How to Start *Routing Service* (Section 3.1.1).
- How to Stop *Routing Service* (Section 3.1.2).
- *Routing Service* Command-line Parameters (Section 3.1.3).

3.1.1 Starting Routing Service

To start *Routing Service* with a default configuration, enter:

```
$NDDSHOME/bin/rtiroutingservice
```

This command will run *Routing Service* indefinitely until you stop it. See Section 3.1.2.

Table 3.1 describes the command-line parameters.

3.1.2 Stopping Routing Service

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

3.1.3 Routing Service Command-Line Parameters

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

Table 3.1: Routing Service Command-Line Parameters

Parameter	Description
-appName <string>	Assigns a name to the execution of the Routing Service. Remote commands and status information will refer to the instances using this name. In addition, the names of *DomainParticipants* created by the service will be based on this name. **Default:** empty string (uses configuration name).
-cfgFile <string>	Semicolon-separated list of configuration file paths. **Default:** unspecified
-cfgName <string>	Specifies the name of the *Routing Service* configuration to be loaded. It must match a `<routing_service>` tag in the configuration file. **Default:** rti.routingservice.builtin.config.default.
-convertLegacyXml <string>	Converts the legacy XML specified with `-cfgFile` and produces the result in the specified output path. If no output path is provided, the converted file will be in the same path than -cfgFile with the suffix `converted`.
-domainIdBase <int>	Sets the base domain ID. This value is added to the domain IDs for all the *DataReader*'s *DomainParticipants* in the configuration file. For example, if you set -domainIdBase to 50 and use domain IDs 0 and 1 in the configuration file, then the Routing Service will use domains 50 and 51. **Default:** 0
-D<name>=<value>	Defines a variable that can be used as an alternate replacement for XML environment variables, specified in the form $(VAR_NAME). Note that definitions in the environment take precedence over these definitions.
-heapSnapshotDir <dir>	Specifies the 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>	Specifies the period at which heap monitoring snapshots are dumped. For example, *Routing Service* will generate a heap snapshot every <sec>. Enables heap monitoring if > 0. **Default**: 0 (disabled)
-help	Prints this help and exits.
-identifyExecution	Appends the host name and process ID to the service name provided with the -appName option. This option helps ensure unique names for remote administration and monitoring. For example: MyRoutingService_myhost_20024 **Default:** false
-ignoreXsdValidation	Loads the configuration even if the XSD validation fails.

Continued on next page

Table 3.1 – continued from previous page

Parameter	Description
-licenseFile	Specifies the path to the license file, required for license-managed distributions.
-listConfig	Prints the available configurations and exits.
-maxObjectsPerThread <int>	Maximum number of thread-specific objects that can be created. **Default:** 2048
-noAutoEnable	Starts Routing Service in a disabled state. Use this option if you plan to enable the service remotely. **Overrides:** This option overrides the <routing_service> tag's "enabled" attribute in the configuration file. **Default:** false
-pluginSearchPath	<path> Specifies a directory where plug-in libraries are located. **Default:** current working directory
-remoteAdministrationDomainId <int>	Enables remote administration and sets the domain ID for remote communication. **Overrides:** This option overrides the <administration> tag's "enabled" attribute and <administration>/<domain_id> in the configuration file. **Default:** unspecified
-remoteMonitoringDomainId <int>	Enables remote monitoring and sets the domain ID for status publication. **Overrides:** This option overrides <monitoring>/<enabled> and <monitoring>/<domain_id> in the configuration file. **Default:** unspecified
-skipDefaultFiles	Skips attempting to load the default configuration files **Default:** false
-stopAfter <int>	Number of seconds the *Routing Service* runs before it stops. **Default:** (infinite).
-verbosity <int>	Controls what type of messages are logged: 0. Silent 1. Exceptions (*Connext DDS* and *Routing Service*) 2. Warnings (*Routing Service*) 3. Warnings (*Connext DDS*) 4. Local (*Routing Service*) 5. Remote (*Routing Service*) 6. Activity (*Routing Service*) and Local (*Connext DDS*) Each verbosity level, `n`, includes all the verbosity levels smaller than `n`. **Default:** 1 (Exceptions)
-version	Prints the *Routing Service* version number and exits.

All the command-line parameters are optional; if specified, they override the values of their corresponding settings in the loaded XML configuration. See Section 4 for the set of XML elements that can be overridden with command-line parameters.

3.2 Routing Service Library

Routing Service can be deployed as a library linked into your application on selected architectures (see Section 13). This allows you to create, configure, and start *Routing Service* instances from your application.

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

If you are using the C API, see the example in `<path to examples>/routing_service/routing_service_lib`. Example makefiles and project files for several architectures are provided. Also see the README.txt file in the `routing_service_lib/src` directory.

3.2.1 Example

```
struct RTI_RoutingServiceProperty property =
        RTI_RoutingServiceProperty_INITIALIZER;
struct RTI_RoutingService * service = NULL;

/* initialize property */
property.cfg_file     = "my_routing_service_cfg.xml";
property.service_name = "my_routing_service";
...

service = RTI_RoutingService_new(&property);
if(service == NULL) {
    /* log error */
    ...
}

if(!RTI_RoutingService_start(service)) {
    /* log error */
    ...
}

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

RTI_RoutingService_delete(service);
```

Chapter 4

Configuration

4.1 Configuring Routing Service

This document describes how to configure *Routing Service*. For installation instructions or to walk through some simple examples, please see the appropriate section.

When you start *Routing Service*, you can specify a configuration file in XML format. In that file, you can set properties that control the behavior of the service. This chapter describes how to write a configuration file.

4.2 Terms to Know

Before learning how to configure *Routing Service*, you should become familiar with a few key terms and concepts:

- An *AutoRoute* defines a set of potential *Routes* that can be instantiated based on deny/allow filters on the stream name and registered type name.
- A *Transformation* is a pluggable component that changes data from an input stream to an output stream.
- An *Adapter* is a pluggable component that allows *Routing Service* to consume and produce data for different data domains. By default, *Routing Service* is distributed with a builtin DDS adapter.

4.3 How to Load the XML Configuration

Routing Service loads its XML configuration from multiple locations. This section presents the various sources of configuration files, listed in load order.

- `[working directory]/USER_ROUTING_SERVICE.xml` This file is loaded automatically if it exists.
- `[NDDSHOME]/resource/xml/RTI_ROUTING_SERVICE.xml` This file is loaded automatically if it exists.

- One or more files (semicolon-separated) specified using the command-line parameter `-cfgFile`.

Note: `[working directory]` indicates the path to the current working directory from which you run *Routing Service*.

The tag `[NDDSHOME]` indicates the path to your *Connext DDS* installation.

You may use a combination of the above sources and load multiple configuration files.

Here is an example configuration file. You will learn the meaning of each line as you read the rest of this section.

```
<?xml version="1.0"?>
<dds>
    <routing_service name="TopicBridgeExample" group_name="MyGroup">
        <domain_route name="DomainRoute">
            <participant name="domain0">
               <domain_id>0</domain_id>
            </participant>

            <participant name="domain1">
               <domain_id>1</domain_id>
            </participant>

            <session name="Session">
                <topic_route name="SquaresToCircles">

                    <input participant="domain0">
                        <registered_type_name>
                            ShapeType
                        </registered_type_name>
                        <topic_name>Square</topic_name>
                    </input>

                    <output participant="domain1">
                        <registered_type_name>
                            ShapeType
                        </registered_type_name>
                        <topic_name>Circle</topic_name>
                    </output>

                </topic_route>
            </session>
        </domain_route>
    </routing_service>
</dds>
```

4.4 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, *Routing Service* relies on an XSD file that describes the format of the XML content. 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 definitions of the XML elements are in `[NDDSHOME]/resource/schema/rti_routing_service.xsd`.

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

Note: The tag `[NDDSHOME]` indicates the path to your *Connext DDS* installation.

4.5 XML Tags for Configuring RTI Routing Service

This section describes the XML tags you can use in a *Routing Service* configuration file. The following diagram and Table 4.1 describe the top-level tags allowed within the root `<dds>` tag.

Warning: The tables in this section may not necessarily reflect the order the *Routing Service* XSD requires. Use these tables as a documentation reference only.

Table 4.1: Top-Level Tags in the Configuration File

Tags within <dds>	Description	Multiplicity
<qos_library>	Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for a *Connext DDS* application. See Configuring QoS with XML, in the Connext DDS Core Libraries User's Manual.	0..*
<types>	Defines types that can be used by *Routing Service*. See Section 4.5.7.	0..1

Continued on next page

Table 4.1 – continued from previous page

Tags within <dds>	Description	Multiplicity
<plugin_library>	Specifies a library of *Routing Service* plugins. Available plug-ins are *Adapters*, *Transformations* and *Processors*. See Section 4.5.9.	0..*

<routing_service>	Specifies a *Routing Service* configuration. See Section 4.5.1.
	Attributes
	• `name`: Uniquely identifies a *Routing Service* configuration. Optional.
	• `enabled`: A boolean that indicates whether this entity is auto-enabled when the service starts. If set to false, the entity can be enabled after the service starts through remote administration. Optional. Default: true.
	• `group_name`: A name that can be used to implement a specific policy when the communication happens between *Routing Service* of the same group. For example, in the builtin DDS adapter, a *DomainParticipant* will ignore other *DomainParticipants* in the same group, as a way to avoid circular communication. Optional. Default: RTI_RoutingService_`[Host Name]`_`[Process ID]`
	Example
	`<routing_service name="ExampleService">`
	`  <!-- your service settings ... -->`
	`</routing_service>`
		1..*

4.5.1 Routing Service Tag

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

When you start *Routing Service*, you can specify which `<routing_service>` tag to use to configure the service using the `-cfgName` command-line parameter.

Because a configuration file may contain multiple `<routing_service>` tags, one file can be used to configure multiple *Routing Service* executions.

Figure 4.2 and Table 4.2 describes the tags allowed within a `<routing_service>` tag.

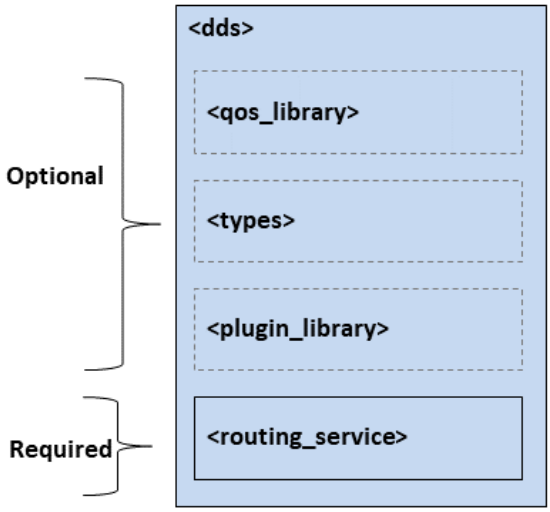


Figure 4.1: Top-level Tags in the Configuration File

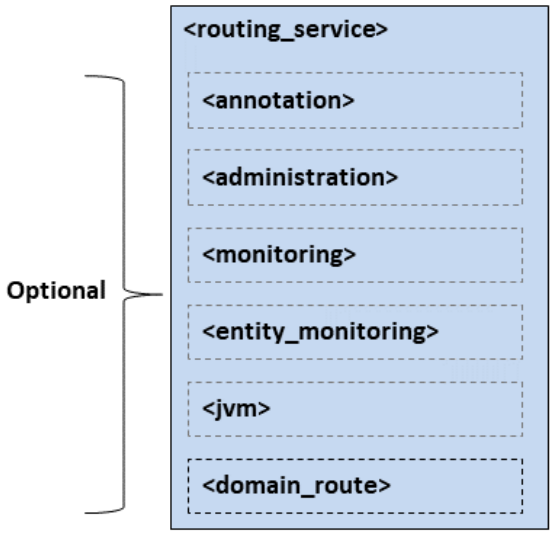


Figure 4.2: *Routing Service* Tag Structure

Table 4.2: *Routing Service* Tag

Tags within <routing_service>	Description	Multiplicity
<annotation>	Contains a <documentation> tag that can be used to provide a configuration description.	0..1
<administration>	Enables and configures remote administration. See Section 4.5.2 and Section 5.	0..1
<monitoring>	Enables and configures general remote monitoring. General monitoring settings are applicable to all the *Routing Service* entities unless they are explicitly overridden. See Section 4.5.3 and Section 6.	0..1
<entity_monitoring>	Enables and configures remote monitoring for the service entity. See Section 4.5.3 and Section 6.	0..1
<jvm>	Configures the Java JVM used to load and run Java adapters. For example: **Example** <jvm> <class_path> <element>SocketAdapter.jar</element> </class_path> <options> <element>-Xms32m</element> <element>-Xmx128m</element> </options> </jvm> You can use the `<options>` tag to specify options for the JVM, such as the initial and maximum Java heap sizes.	0..1
<domain_route>	Defines a mapping between two or more data domains. See Section 4.5.4. Attributes • `name`: uniquely identifies a domain_route configuration. Optional. • `enabled`: A boolean that indicates whether this entity is auto-enabled when the service starts. If set to false, the entity can be enabled after the service starts through remote administration. Optional. Default: true.	0..1

Example: Specifying a configuration in XML

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

Starting *Routing Service* with the following command will use the `<routing_service>` tag with the name `EmptyConfiguration`.

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

4.5.2 Administration

You can create a *Connext DDS* application that can remotely control *Routing Service*. The `<administration>` tag is used to enable remote administration and configure its behavior. By default, remote administration is turned off in *Routing Service* for security reasons. A remote administration section is not required in the configuration file.

When remote administration is enabled, *Routing Service* will create a *DomainParticipant*, *Publisher*, *Subscriber*, *DataWriter*, and *DataReader*. These entities are used to receive commands and send responses. You can configure these entities with QoS tags within the `<administration>` tag. The following table lists the tags allowed within `<administration>` tag. Notice that the `<domain_id>` tag is required.

For more details, please see Section 5.

Note: The command-line options used to configure remote administration take precedence over the XML configuration (see Section 3).

Table 4.3: Administration Tag

Tags within <administration>	Description	Multiplicity
<enabled>	Enables/disables administration. **Default:** true	0..1
<domain_id>	Specifies which domain ID *Routing Service* will use to enable remote administration.	0..1
<participant_qos>	Configures the *DomainParticipant* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..1
<publisher_qos>	Configures the *Publisher* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..1
<subscriber_qos>	Configures the *Subscriber* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..1
<datawriter_qos>	Configures the *DataWriter* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults with the following changes: • history.kind = DDS_KEEP_ALL_HISTORY_QOS • resource_limits.max_samples = 32	0..1

Continued on next page

Table 4.3 – continued from previous page

Tags within <administration>	Description	Multiplicity
<datareader_qos>	Configures the *DataReader* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults 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>	Configures *RTI Distributed Logger*. When you enable it, *Routing Service* will publish its log messages to *Connext DDS*.
	Example:
	`<administration>`
	`...`
	`<distributed_logger>`
	`<enabled>true</enabled>`
	`</distributed_logger>`
	`</administration>`
		0..1

<autosave_on_update>	A boolean that, if true, automatically triggers a save command when configuration updates are received. This value is sent as part of the monitoring configuration data for the *Routing Service*. **Default:** false.	0..1
<save_path>	Specifies the file that will contain the saved configuration. A `<save_path>` must be specified if you want to use the remote save command ((Section 5.2). If the specified file already exists, the file will be overwritten when save is executed. **Default:** [CURRENT DIRECTORY].	0..1

4.5.3 Monitoring

You can create a *Connext DDS* application that can remotely monitor the status of *Routing Service*. To enable remote monitoring and configure its behavior, use the `<monitoring>` and `<entity_monitoring>` tags.

By default, remote monitoring is turned off in *Routing Service* for security and performance reasons. A remote monitoring section is not required in the configuration file.

When remote monitoring is enabled, *Routing Service* will create one *DomainParticipant*, one *Publisher*, five *DataWriters* for data publication (one for each kind of entity), and five *DataWriters* for status publication (one for each kind of entity). You can configure the QoS of these entities with the `<monitoring>` tag defined under `<routing_service>`. The general remote monitoring parameters specified using the `<monitoring>` tag in `<routing_service>` can be overwritten on a per entity basis using the `<entity_monitoring>` tag.

For more details, please see Section 6.

Note: The command-line options used to configure remote monitoring take precedence over the XML configuration (See Section 3).

Table 4.4: Monitoring Tag

Tags within <monitoring>	Description	Multiplicity
<enabled>	Enables/disables general remote monitoring. Setting this value to true enables monitoring in all the entities unless they explicitly disable it by setting this tag to false in their local `<entity_monitoring>` tags. Setting this tag to false disables monitoring in all the entities. In this case, any monitoring configuration settings in the entities are ignored. **Default:** true	0..1
<domain_id>	Specifies which domain ID *Routing Service* will use to enable remote monitoring.	0..1
<ignore_initialization_failure>	Indicates whether a failure initializing the monitoring engine for the service or any of the underlying entities is ignored.|br| If false, a failure initializing monitoring will result in a failure creating the service or the affected entities. **Default:** false	0..1
<participant_qos>	Configures the *DomainParticipant* QoS for remote monitoring. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults, with the following change: • resource_limits.type_code_max_serialized_length = 4096	0..1
<publisher_qos>	Configures the *Publisher* QoS for remote monitoring. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..1
<datawriter_qos>	Configures the *DataWriter* QoS for remote monitoring. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults with the following change: • durability.kind = DDS_TRANSIENT_LOCAL_DURABILITY_QOS	0..1

Continued on next page

Table 4.4 – continued from previous page

Tags within <monitoring>	Description	Multiplicity

<statistics_sampling_period>	Specifies the frequency, in seconds, at which status statistics are gathered. Statistical variables such as latency are part of the entity status.
	Example:
	`<statistics_sampling_period>`
	`<sec>1</sec>`
	`<nanosec>0</nanosec>`
	`</statistics_sampling_period>`
	The statistics period for a given entity should be smaller than the publication period. The statistics sampling period defined in `<routing_service>` is inherited by all the entities. An entity can overwrite the period. **Default:** 1
		0..1

<statistics_publication_period>	Specifies the frequency, in seconds, at which the status of an entity is published.
	Example:
	`<statistics_publication_period>`
	`<sec>5</sec>`
	`<nanosec>0</nanosec>`
	`</statistics_publication_period>`
	The statistics sampling period defined in `<routing_service>` is inherited by all the entities. An entity can overwrite the period. **Default:** 5
		0..1

Monitoring Configuration Inheritance

The monitoring configuration defined in `<routing_service>` is inherited by all the entities defined inside the tag.

An entity can overwrite three elements of the monitoring configuration:

- The status publication period
- The statistics sampling period
- The historical statistics windows

Each one of these three elements is inherited and can be overwritten independently using the `<entity_monitoring>` tag.

Table 4.5: Entity Monitoring Tag

Tags within <entity_monitoring>	Description	Multiplicity
<enabled>	Enables/disables remote monitoring for a given entity. If general monitoring is disabled, this value is ignored. **Default:** true	0..1

Continued on next page

Table 4.5 – continued from previous page

Tags within <entity_monitoring>	Description	Multiplicity

<statistics_sampling_period>	Specifies the frequency at which status statistics are gathered. Statistical variables such as latency are part of the entity status.
	Example:
	`<statistics_sampling_period>`
	`  <sec>1</sec>`
	`  <nanosec>0</nanosec>`
	`</statistics_sampling_period>`
	The statistics period for a given entity should be smaller than the publication period.
	If this tag is not defined, historical statistics are inherited from the general monitoring settings.
	Default: 1 second.
		0..1

<statistics_publication_period>	Specifies the frequency at which the status of an entity is published.
	Example:
	`<statistics_publication_period>`
	`  <sec>5</sec>`
	`  <nanosec>0</nanosec>`
	`</statistics_publication_period>`
	If this tag is not defined, historical statistics are inherited from the general monitoring settings.
	Default: 5 seconds.
		0..1

Example: Overriding Publication Period

```
<routing_service name="MonitoringExample">
    <monitoring>
        <domain_id>55</domain_id>
        <status_publication_period>
            <sec>1</sec>
        </status_publication_period>
        <statistics_sampling_period>
            <sec>1</sec>
            <nanosec>0</nanosec>
        </statistics_sampling_period>
    </monitoring>
    ...
    <domain_route>
        <entity_monitoring>
            <status_publication_period>
                <sec>4</sec>
            </status_publication_period>
        </entity_monitoring>
        ...
```

(continues on next page)

(continued from previous page)

```
    </domain_route>
</routing_service>
```

4.5.4 Domain Route

A `<domain_route>` defines a mapping between different data domains. Data available in any of these data domains can be routed to other data domains. For example, a *DomainRoute* could define a mapping among multiple DDS domains, or between a DDS domain and a MQTT provider's network. How this data is actually read and written is defined in specific *Routes*.

A `<domain_route>` creates one or more *Connections*. Each *Connection* typically belongs to a different data domain. The `<connection>` tag requires the specification of the attribute `name`, which will be used by the *Route* to select input and output domains, and the `plugin_name`, which will be used to associate a *Connection* with an adapter plugin defined within `<adapter_library>`.

Routing Service comes with a builtin implementation of a DDS adapter, which can be used by specifying the `<participant>` tag. Each tag corresponds to exactly one *DomainParticipant*. A *DomainRoute* can include both `<connection>` and `<participant>` tags to provide communication between DDS domains and other data domains.

Table 4.6: Domain Route Tag

Tags within <domain_route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *DomainRoute*. See Section 6.	0..1
<connection>	Applicable to non-DDS domains. Configures a custom, adapter-based connection. Attributes • `name`: Uniquely identifies a service configuration. Required. • `plugin_name`: Name of the plug-in that creates an adapter object. This name shall refer to an adapter plug-in registered either in a `<plugin_library>` or with the service's attach_adapter_plugin() operation. Required. See Table 4.7.	0..*
<participant>	Applicable to DDS domains. Configures a DDS adapter *DomainParticipant*. See Table 4.8.	0..*

Continued on next page

Table 4.6 – continued from previous page

Tags within <domain_route>	Description	Multiplicity
<session>	Defines a multi-threaded context in which data is routed according to specified routes. See Section 4.5.5. Attributes • `name`: uniquely identifies the *Session* configuration. Optional. • `enabled`: A boolean that indicates whether this entity is auto-enabled when the service starts. If set to false, the entity can be enabled after the service starts through remote administration. Optional. Default: true.	0..*

Table 4.7: Connection Tag

Tags within <connection>	Description	Multiplicity

<property>	A sequence of name-value string pairs that allows you to configure the *Connection* instance.
	Example:
	`<property>`
	`  <value>`
	`  <element>`
	`  <name>jms.connection.username</name>`
	`  <value>myusername</value>`
	`  </element>`
	`  </value>`
	`</property>`
		0..1

<registered_type>	Registers a type name and associates it with a type representation. When you define a type in the configuration file, you have to register the type in order to use it in *Routes*. See Section 4.5.6.	0..*

Table 4.8: Participant Tag

Tags within <participant>	Description	Multiplicity
<domain_id>	Sets the domain ID associated with the *DomainParticipant*. **Default:** 0	0..1

Continued on next page

Table 4.8 – continued from previous page

Tags within <participant>	Description	Multiplicity

<participant_qos>	Sets the participant QoS. The contents of this tag are specified in the same manner as a *Connext DDS* QoS profile. If not specified, the DDS defaults are used, except for the participant name which takes the following value: "RTI Routing Service: <service name>.<domain route name>#[1|2]" (for example "RTI Routing Service: MyService.MyDomainRoute#1").
	
	Note: Changing the default participant name may prevent *Routing Service* from being detected by Admin Console.
	
	You can use a `<participant_qos>` tag inside a `<qos_library>/<qos_profile>` previously defined in your configuration file by referring to it, and also override any value:
	Example:
	`<participant_qos base_name="MyLibrary::MyProfile">`
	`  <discovery>`
	`  <initial_peers>`
	`  <element>udpv4://192.168.1..12</element>`
	`  <element>shmem://</element>`
	`  </initial_peers>`
	`  </discovery>`
	`</participant_qos>`
	See Configuring QoS with XML, in the Connext DDS Core Libraries User's Manual.
		0..1

Continued on next page

Table 4.8 – continued from previous page

Tags within <participant>	Description	Multiplicity

<memory_management>	Configures certain aspects of how *Connext DDS* allocates internal memory. The configuration is per *DomainParticipant* and therefore affects all the contained DDS entities.
	Example:
	`<memory_management>`
	`  <sample_buffer_min_size>`
	`  1024`
	`  </sample_buffer_min_size>`
	`  <sample_buffer_trim_to_size>`
	`  true`
	`  </sample_buffer_trim_to_size>`
	`</memory_management>`
	This tag includes the following tags:
	• **sample_buffer_min_size**: For all *DataReaders* and *DataWriters*, the way *Connext DDS* allocates memory for samples is as follows: *Connext DDS* pre-allocates space for samples up to size X in the *DataReader* and *DataWriter* queues. If a sample has an actual size greater than X, the memory is allocated dynamically for that sample. The default size is 64KB. This is the maximum amount of pre-allocated memory. Dynamic memory allocation may occur when necessary if samples require a bigger size.
	• **sample_buffer_trim_to_size**: If set to true, after allocating dynamic memory for very large samples, that memory will be released when possible. If false, that memory will not be released but kept for future samples if needed. The default is false.
	This feature is useful when a data type has a very high maximum size (e.g., megabytes) but most of the samples sent are much smaller than the maximum possible size (e.g., kilobytes). In this case, the memory footprint is reduced dramatically, while still correctly handling the rare cases in which very large samples are published.
		0..1

<registered_type>	Registers a type name and associates it with a type representation. When you define a type in the configuration file, you have to register the type in order to use it in *Routes*. See Section 4.5.6.	0..*

Example: Mapping between Two DDS Domains

```
<domain_route name="DdsDomainRoute">
    <participant name="domain54">
        <domain_id>54</domain_id>
        ...
    </participant>

    <articipant name="domain55">
        <domain_id>55</domain_id>
        ...
    </participant>

    ...
</domain_route>
```

Example: Mapping between a DDS Domain and raw Sockets

```
<domain_route name="DomainRoute">
    <connection name="SocketAdapter">
        ...
    </connection>

    <participant name="domain55">
        <domain_id>55</domain_id>
        ...
    </participant>

    ...
</domain_route>
```

4.5.5 Session

A `<session>` tag defines a multi-threaded context for route processing, including data forwarding. The data is routed according to specified *Routes* and *AutoRoutes*.

Each *Session* will have an associated thread pool to process *Routes* concurrently, preserving *Route* safety. Multiple *Routes* can be processed concurrently, but a single *Route* can be processed only by one thread at time. By default, the session thread pool has a single thread, which serializes the processing of all the *Routes*.

Sessions that bridge domains will create a *Publisher* and a *Subscriber* from the *DomainParticipants* associated with the domains. Table 4.9 lists the tags allowed within a `<session>` tag.

Table 4.9: Session Tag

Tags within <session>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *Session*. See Section 6.	0..1

Continued on next page

Table 4.9 – continued from previous page

Tags within <session>	Description	Multiplicity

<thread_pool>	Defines the number of threads to process *Routes* and sets the mask, priority, and stack size of each thread.
	Example:
	`<thread_pool>`
	`  <mask>MASK_DEFAULT</mask>`
	`  <priority>THREAD_PRIORITY_DEFAULT</priority>`
	`  <stack_size>`
	`  THREAD_STACK_SIZE_DEFAULT`
	`  </stack_size>`
	`</thread_pool>`
	Default values:
	• **size**: 1
	• **mask**: MASK_DEFAULT
	• **priority**: THREAD_PRIORITY_DEFAULT
	• **stack_size**: THREAD_STACK_SIZE_DEFAULT
		0..1

<periodic_action>	Specifies a period at which Processors will receive notifications of the periodic event. The *Session* will wake up and notify the installed Processor every specified period.
	Default: INFINITE (no periodic notification)
	Example:
	`<periodic_action>`
	`  <sec>1</sec>`
	`  <nanosec>0</nanosec>`
	`</periodic_action>`
	The example above indicates the installed Processor should be notified every one second.
		0..1

<property>	A sequence of name-value string pairs that allows you to configure the *Session* instance.
	Example:
	`<property>`
	`  <value>`
	`  <element>`
	`  <name>com.rti.socket.timeout</name>`
	`  <value>1</value>`
	`  </element>`
	`  </value>`
	`</property>`
	These properties are only used in non-DDS domains.
		0..1

Continued on next page

Table 4.9 – continued from previous page

Tags within <session>	Description	Multiplicity
<subscriber_qos>	Only applicable to *Routes* that are *Connext DDS Routes*. Sets the QoS associated with the session *Subscribers*. There is one *Subscriber* per *DomainParticipant*. The contents of this tag are specified in the same manner as a *Connext DDS* QoS profile. See Configuring QoS with XML, in the Connext DDS Core Libraries User's Manual. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..1
<publisher_qos>	Only applicable to *Routes* that are *Connext DDS Routes*. Sets the QoS associated with the session *Publishers*. There is one *Publisher* per *DomainParticipant*. The contents of this tag are specified in the same manner as a *Connext DDS* QoS profile. See Configuring QoS with XML, in the Connext DDS Core Libraries User's Manual. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..*
<topic_route> or <route>	Defines a mapping between multiple input and output streams. Attributes • `name`: uniquely identifies a *TopicRoute* or *Route* configuration. Optional. • `enabled`: A boolean that indicates whether this entity is auto-enabled when the service starts. If set to false, the entity can be enabled after the service starts through remote administration. Optional. Default: true. See Section 4.5.6.	0..*
<auto_topic_route> or <auto_route>	Defines a factory for *Route* based on type and stream filters. See Section 4.5.8. Attributes • `name`: uniquely identifies an *AutoTopicRoute* or *AutoRoute* configuration. Optional. • `enabled`: A boolean that indicates whether this entity is auto-enabled when the service starts. If set to false, the entity can be enabled after the service starts through remote administration. Optional. Default: true.	0..*

4.5.6 Route

A *Route* explicitly defines a mapping between one or more input data streams and one or more output data streams. The input and output streams may belong to different data domains.

Route events are processed in the context of the thread belonging to the parent *Session*. *Route* event processing includes, among others, calls to the *StreamReader* read and *StreamWriter* write operations.

Table 4.10 lists the tags allowed within a `<route>`. Table 4.11 lists the tags allowed within a `<topic_route>`.

Table 4.10: Route Tag

Tags within <route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *Route*. See Section 6.	0..1
<route_types>	Defines if the input connection will use types discovered in the output connection and vice versa for the creation of *StreamWriters* and *StreamReaders* in the *Route*. See Section 4.5.7. **Default:** false	0..1
<publish_with_original_timestamp>	When this tag is true, the data samples read from the input stream are written into the output stream with the same timestamp that was associated with them when they were made available in the input domain. This option may not be applicable in some adapter implementations in which the concept of timestamp is unsupported. **Default:** false	0..1
<processor>	Sets a custom Processor for handling the data forwarding process. See Section 7. **Attributes** • `plugin_name`: Name of the plug-in that creates a *Processor* object. This name shall refer to a processor plug-in registered either in a `<plugin_library>` or with the service attach_processor() operation.	0..1
<dds_input>	Only applicable to DDS inputs. Defines an input topic. See Section 4.5.7. **Attributes** • `name`: uniquely identifies an input configuration. Optional.	0..*
<dds_output>	Only applicable to DDS outputs. Defines an output topic. See Section 4.5.7. **Attributes** • `name`: uniquely identifies an output configuration. Optional.	0..*

Continued on next page

Table 4.10 – continued from previous page

Tags within <route>	Description	Multiplicity
<input>	Only applicable to non-DDS inputs. Defines an input stream. See Section 4.5.7. **Attributes** • `name`: uniquely identifies an input configuration. Optional.	0..*
<output>	Only applicable to non-DDS outputs. Defines an output stream. See Section 4.5.7. **Attributes** • `name`: uniquely identifies an output configuration. Optional.	0..*

Table 4.11: Topic Route Tag

Tags within <topic_route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *TopicRoute*. See Section 6.	0..1
<route_types>	Defines if the input connection will use types discovered in the output connection and vice versa for the creation of *DataReaders* and *DataWriters* in the *Route*. See Section 4.5.7. **Default:** false	0..1
<publish_with_original_info>	Writes the data sample as if they came from its original writer. Setting this option to true allows having redundant routing services and prevents the applications from receiving duplicate samples. **Default**: false	0..1
<publish_with_original_timestamp>	Indicates if the data samples are written with their original source timestamp. **Default**: false	0..1
<propagate_dispose>	Indicates whether or not disposed samples (NOT_ALIVE_DISPOSE) must be propagated by the *TopicRoute*. This action may be overwritten by the execution of a transformation. **Default**: true	0..1
<propagate_unregister>	Indicates whether or not disposed samples (NOT_ALIVE_NO_WRITERS) must be propagated by the *TopicRoute*. This action may be overwritten by the execution of a transformation. **Default**: true	0..1

Continued on next page

Table 4.11 – continued from previous page

Tags within <topic_route>	Description	Multiplicity

<topic_query_proxy>	Configures the forwarding of *TopicQueries*. See Section 9 for detailed information on how *Routing Service* processes *TopicQueries*.
	The snippet below shows that topic query proxy is enabled in propagation mode, which causes the creation of a *TopicQuery* on the route's input for each *TopicQuery* that an output's matching *DataReader* creates.
	Example:
	`<topic_query_proxy>`
	`  <enabled>true</enabled>`
	`  <mode>PROPAGATION</mode>`
	`</topic_query_proxy>`
		0..1

<filter_propagation>	Configures the propagation of content filters. Specifies whether the feature is enabled and when events are processed (Section 8).
	Filter propagation events can be batched to reduce the traffic in detriment of increasing the delay in propagating the composed filter. Event batching can be configured with the following tags:
	• <max_event_count>: Indicates the minimum number of filter indication events required before propagating the composed filter.
	• <max_event_delay>: Indicates the minimum amount of time to wait before propagating the composed filter.
	The previous two tags can be set in combination. In this case, the composed filter is propagated whenever one of these conditions is met first.
	The snippet below shows that filter propagation is enabled, and a filter update is propagated on the *StreamReader* only after the occurrence of every three filter events (see Section 8).
	Example:
	`<filter_propagation>`
	`  <enabled>true</enabled>`
	`  <max_event_count>3</max_event_count>`
	`  <max_event_delay>`
	`  <sec>DURATION_INFINITE_SEC</sec>`
	`  <nanosec>DURATION_INFINITE_NSEC</`
	`↪nanosec>`
	`  </max_event_delay>`
	`</filter_propagation>`
		0..1

Continued on next page

Table 4.11 – continued from previous page

Tags within <topic_route>	Description	Multiplicity
<processor>	Sets a custom Processor for handling the data forwarding process. See Section 7. **Attributes** • `plugin_name`: Name of the plug-in that creates a *Processor* object. This name shall refer to a processor plug-in registered either in a `<plugin_library>` or with the service attach_processor() operation.	0..1
<input>	Defines an input topic. See Section 4.5.7. **Attributes** • `name`: uniquely identifies an input configuration. Optional.	0..*
<output>	Defines an output topic. See Section 4.5.7. **Attributes** • `name`: uniquely identifies an output configuration. Optional.	0..*

4.5.7 Input/Output

Inputs and outputs in a *Route* or *TopicRoute* have an associated *StreamReader* and *StreamWriter*, respectively. For DDS domains, the *StreamReader* will contain a *DataReader* and the *StreamWriter* will contain a *DataWriter*. The *DataReaders* and *DataWriters* belong to the corresponding *Session Subscriber* and *Publisher*.

DDS inputs and outputs within a *Route* are defined using the `<dds_input>` and `<dds_output>` tags. Inputs and outputs from other data domains are defined using the `<input>` and `<output>` tags. A *TopicRoute* is a special kind of *Route* that allows defining mapping between DDS topics only.

Table 4.12: Route Input/Output Tags

Tags within <input> and <output> of <route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *Input/Output*. See Section 6.	0..1
<stream_name>	Specifies the stream name.	1
<registered_type_name>	Specifies the registered type name of the stream.	1
<creation_mode>	Specifies when to create the StreamReader/StreamWriter. **Default:** IMMEDIATE See Section 4.5.7.	0..1

Continued on next page

Table 4.12 – continued from previous page

Tags within <input> and <output> of <route>	Description	Multiplicity

<property>	A sequence of name-value string pairs that allows you to configure the *StreamReader*/*StreamWriter*.
	Example:
	`<property>`
	`<value>`
	`<element>`
	`<name>com.rti.socket.port</name>`
	`<value>16556</value>`
	`</element>`
	`</value>`
	`</property>`
		0..1

<transformation> (within <output> only)	Sets a data transformation to be applied for every data sample. See Section 4.5.7. Attributes • `plugin_name`: Name of the plug-in that creates a *Transformation* object. This name shall refer to a transformation plug-in registered either in a `<plugin_library>` or with the service attach_transformation() operation.	0..1

Table 4.13: TopicRoute Input/Output Tags

Tags within <input> and <output> (in <topic_route>) and <dds_input> and <dds_output> (in <route>)	Description	Multiplicity
<topic_name>	Specifies the topic name.	1
<registered_type_name>	Specifies the registered type name of the topic.	1
<creation_mode>	Specifies when to create the StreamReader/StreamWriter. **Default:** IMMEDIATE See Section 4.5.7.	0..1
<datareader_qos> or <datawriter_qos>	Sets the DataReader or DataWriter QoS. The contents of this tag are specified in the same manner as a *Connext DDS* QoS profile. See Configuring QoS with XML, in the Connext DDS Core Libraries User's Manual. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..1

Continued on next page

Table 4.13 – continued from previous page

Tags within <input> and <output> (in <topic_route>) and <dds_input> and <dds_output> (in <route>)	Description	Multiplicity

<content_filter>	Defines a SQL content filter for the *DataReader*.
	Example:
	`<content_filter>`
	`<expression>`
	`x > 100`
	`</expression>`
	`</content_filter>`
		0..1

<transformation>‘ (within <output> only)	Sets a data transformation to be applied for every data sample. See Section 4.5.7. Attributes • `plugin_name`: Name of the plug-in that creates a *Transformation* object. This name shall refer to a transformation plug-in registered either in a `<plugin_library>` or with the service attach_transformation() operation.	0..1

Creation Modes

The way a *Route* creates its *StreamReaders* and *StreamWriters* and starts reading and writing data can be configured.

The `<creation_mode>` tag in a *Route*'s `<input>` and `<output>` tags controls when *StreamReaders*/*StreamWriters* are created.

Table 4.14: Route Creation Mode

<creation_mode> values	Description
IMMEDIATE	The *StreamReader*/*StreamWriter* is created as soon as possible; that is, as soon as the types are available. Note that if the type is defined in the configuration file, the creation will occur when the service starts.

Continued on next page

Table 4.14 – continued from previous page

<creation_mode> values	Description
ON_DOMAIN_MATCH	The *StreamReader* is not created until the associated connection discovers a data Producer on the same stream. If the adapter supports partition, the discovered Producer must also belong to the same partition for a match to occur. For example, a DDS input will not create a *DataReader* until a *DataWriter* for the same topic and partition is discovered on the same domain. The *StreamWriter* is not created until the associated connection discovers a data Consumer on the same stream. If the adapter supports partition, the discovered Producer must also belong to the same partition for a match to occur. For example, a DDS output will not create a *DataWriter* until a *DataReader* for the same topic and partition is discovered on the same domain.
ON_ROUTE_MATCH	The *StreamReader*/*StreamWriter* is not created until all its counterparts in the *Route* are created.
ON_DOMAIN_AND_ROUTE_MATCH	Both conditions must be true.
ON_DOMAIN_OR_ROUTE_MATCH	At least one of the conditions must be true.

The same rules also apply to the *StreamReader*/*StreamWriter* destruction. When the condition that triggered the creation of that entity becomes false, the entity is destroyed. Note that IMMEDIATE will never become false.

For example, if the creation mode of an `<input>` tag is ON_DOMAIN_MATCH, when all the matching user *DataWriters* in the input domain are deleted, the input *DataReader* is deleted.

Example: Route Starts as Soon as a User DataWriter is Publishing on 1st Domain

```
<topic_route>
    <input participant="domain1">
        <creation_mode>ON_DOMAIN_MATCH</creation_mode>
        ...
    </input>
    <output participant="domain2">
        <creation_mode>ON_ROUTE_MATCH</creation_mode>
    ...
    </output>
</topic_route>
```

Example: Route Starts when Both User DataWriter Appears in 1st Domain and User DataReader Appears in 2nd Domain

```
<topic_route>
    <input participant="domain1">
        <creation_mode>ON_DOMAIN_AND_ROUTE_MATCH</creation_mode>
        ...
    </input>
    <output participant="domain2">
        <creation_mode>ON_DOMAIN_AND_ROUTE_MATCH</creation_mode>
    ...
    </output>
</topic_route>
```

Specifying Types

The tag `<registered_type_name>` within the `<input>` and `<output>` tags contains the registered type name of the stream. The actual definition of that type can be set in the configuration file or it can be discovered by the *Connections*.

See Section 2.1.4 for more details about type registration.

Defining Types in the Configuration File

To define and use a type in your XML configuration file:

- Define your type within the `<types>` tag. The type description is done using the *Connext DDS* XML format for type definitions. See Creating User Data Types with Extensible Markup Language (XML), in the RTI Connext DDS Core Libraries User's Manual.
- Register it in the `<connection>`/`<participant>` where you will use it.
- Refer to it in the domain route(s) that will use it.

Example: Type Registration in XML

```
<dds>
    ...
    <types>
        <struct name="PointType">
        ...
        </struct>
    </types>
    ...
    <routing_service name="MyRoutingService">
        ...
        <domain_route>
            <connection name="MyConnection">
                ...
                <registered_type name="Position" type_name="PointType"/>
            </connection>
```

(continues on next page)

(continued from previous page)

```
            <participant name="MyParticipant">
                ...
                <registered_type name="Position" type_name="PointType"/>
            </participant>
            ...
            <session>
                <topic_route>
                    <input participant="2">
                        <registered_type_name>Position</registered_type_name>
                    </input>
                    ...
                </topic_route>
            </session>
            ...
        </domain_route>
        ...
    </routing_service>
    ...
<dds>
```

Discovering Types

If the registered type name is not defined in the configuration file, *Routing Service* has to discover its type representation (e.g. typecode). An *Input* or an *Output* cannot be enabled if the type has not been registered yet within the referenced *Connection*.

By default, the *StreamReader* creation will be tied to the discovery of types in the input domain and the *StreamWriter* creation will be tied to the discovery of types in the output domain. If you want to use types discovered in either one of the domains for the creation of both the *StreamReader* and *StreamWriter*, you must set the `<route_types>` tag to true.

Example: Route Creation with Type Obtained from Discovery

```
<dds>
    ...
    <routing_service name="MyRoutingService">
        ...
        <domain_route>
            <participant name="MyParticipant"/>
            ...
            <session>
                <topic_route>
                    <input participant="domain1">
                        <registered_type_name>Position</registered_type_name>
                    </input>
                    ...
                </topic_route>
            </session>
            ...
        </domain_route>
```

(continues on next page)

(continued from previous page)

```
        ...
    </routing_service>
    ...
<dds>
```

Data Transformation

An *Output* can transform the incoming data using a *Transformation*. To instantiate a *Transformation*:

1. Implement the transformation plugin API and register in a plug-in library, or attach it to a service instance if you are using the Service API. See Section 7.
2. Instantiate a *Transformation* object by specifying a `<transformation>` tag inside a `<output>` or `<dds_output>`.

Table 4.15 lists the tags allowed within a `<transformation>` tag.

Table 4.15: Transformation Tag

Tags within <transformation>	Description	Multiplicity

<property>	A sequence of name-value string pairs that allows you to configure the custom *Transformation* plug-in object.
	Example:
	`<property>`
	`  <value>`
	`  <element>`
	`  <name>X</name>`
	`  <value>Y</value>`
	`  </element>`
	`  <element>`
	`  <name>Y</name>`
	`  <value>X</value>`
	`  </element>`
	`  </value>`
	`</property>`
		0..1

4.5.8 Auto Route

The tag `<auto_route>` defines a set of potential *Routes*, with single input and output, both with the same registered type and stream name. A *Route* can eventually be instantiated when a new stream is discovered with a type name and a stream name that match the filters in the *AutoRoute*. When this happens, a *Route* is created with the configuration defined by the *AutoRoute*.

The generated *Route* has a name constructed as follows:

```
[auto_route_name]@[stream_name]
```

where `[auto_route_name]` represents the name of the *AutoRoute* and `[stream_name]` the name of the matching *stream*.

DDS inputs and outputs within an *AutoRoute* are defined using the XML tags `<dds_input>` and `<dds_output>`. Input and outputs from other data domains are defined using the tags `<input>` and `<output>`.

An *AutoTopicRoute* is a special kind of *AutoRoute* that defines a mapping between two DDS domains.

See the following tables for more information on allowable tags:

- Table 4.16 lists the tags allowed within a `<auto_route>`.
- Table 4.17 lists the tags allowed within a `<auto_topic_route>`.

Table 4.16: AutoRoute Tag

Tags within <auto_route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *AutoRoute*. See Section 6.	0..1
<publish_with_original_timestamp>	When this tag is true, the data samples read from the input stream are written into the output stream with the same timestamp that was associated with them when they were made available in the input domain. This option may not be applicable in some adapter implementations in which the concept of timestamp is unsupported. **Default:** false	0..1
<processor>	Sets a custom Processor for handling the data forwarding process. See Section 7. Attributes • `plugin_name`: Name of the plug-in that creates a *Processor* object. This name shall refer to a processor plug-in registered either in a `<plugin_library>` or with the service attach_processor() operation.	0..1
<dds_input>	Only applicable to DDS inputs. Defines an input topic.	0..1
<dds_output>	Only applicable to DDS outputs. Defines an output topic.	0..1
<input>	Only applicable to non-DDS inputs. Defines an input stream.	0..1
<output>	Only applicable to non-DDS outputs. Defines an output stream.	0..1

Table 4.17: AutoTopicRoute Tag

Tags within <auto_topic_route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *AutoTopicRoute*. See Section 6.	0..1
<publish_with_original_info>	Writes the data sample as if they came from its original writer. Setting this option to true allows having redundant routing services and prevents the applications from receiving duplicate samples. **Default**: false	0..1
<publish_with_original_timestamp>	Indicates if the data samples are written with their original source timestamp. **Default**: false	0..1
<propagate_dispose>	Indicates whether or not disposed samples (NOT_ALIVE_DISPOSE) must be propagated by the *TopicRoute*. This action may be overwritten by the execution of a transformation. **Default**: true	0..1
<propagate_unregister>	Indicates whether or not disposed samples (NOT_ALIVE_NO_WRITERS) must be propagated by the *TopicRoute*. This action may be overwritten by the execution of a transformation. **Default**: true	0..1

<topic_query_proxy>	Configures the forwarding of *TopicQueries*. See Section 9 for detailed information on how *Routing Service* processes *TopicQueries*.
	The snippet below shows that topic query proxy is enabled in propagation mode, which causes the creation of a *TopicQuery* on the route's input for each *TopicQuery* that an output's matching *DataReader* creates.
	Example:
	`<topic_query_proxy>`
	`  <enabled>true</enabled>`
	`  <mode>PROPAGATION</mode>`
	`</topic_query_proxy>`
		0..1

Continued on next page

Table 4.17 – continued from previous page

Tags within <auto_topic_route>	Description	Multiplicity

<filter_propagation>	Configures the propagation of content filters. Specifies whether the feature is enabled and when events are processed.
	The snippet below shows that filter propagation is enabled, and a filter update is propagated on the *StreamReader* only after the occurrence of every three filter events (see Section 8).
	Example:
	`<filter_propagation>`
	`<enabled>true</enabled>`
	`<max_event_count>3</max_event_count>`
	`<max_event_delay>`
	`<sec>DDS_DURATION_INFINITE_SEC</sec>`
	`<nanosec>DDS_DURATION_INFINITE_NSEC</nanosec>`
	`</max_event_delay>`
	`</filter_propagation>`
		0..1

<processor>	Sets a custom Processor for handling the data forwarding process. See Section 7. Attributes • `plugin_name`: Name of the plug-in that creates a *Processor* object. This name shall refer to a processor plug-in registered either in a `<plugin_library>` or with the service attach_processor() operation.	0..1
<input>	Defines an input topic.	0..1
<output>	Defines an output topic.	0..1

Table 4.18: AutoRoute Input/Output Tags

Tags within <input> and <output> of <auto_route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *Input*/*Output*. See Section 6.	0..1
<allow_stream_name_filter>	A stream name filter. You may use a comma-separated list to specify more than one filter. **Default:** * (allow all)	0..1
<allow_registered_type_name_filter>	A registered type name filter. You may use a comma-separated list to specify more than one filter. **Default:** * (allow all)	0..1

Continued on next page

Table 4.18 – continued from previous page

Tags within <input> and <output> of <auto_route>	Description	Multiplicity
<deny_stream_name_filter>	A stream name filter that should be denied (excluded). This is applied after the `<allow_stream_name_filter>`. **Default:** empty (not applied)	1
<deny_registered_type_filter>	A registered type name filter that should be denied (excluded). This is applied after the `<allow_registered_type_name_filter>`. **Default:** empty (not applied)	0..1
<creation_mode>	Specifies when to create the StreamReader/StreamWriter. **Default:** IMMEDIATE See Section 4.5.7.	0..1
<property>	A sequence of name-value string pairs that allows you to configure the *StreamReader/StreamWriter*. **Example:** <property> <value> <element> <name>com.rti.socket.port</name> <value>16556</value> </element> </value> </property>	0..1

Table 4.19: AutoTopicRoute Input/Output Tags

Tags within <input> and <output> (in <auto_topic_route>) <dds_input> and <dds_output> (in <auto_route>)	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *Input/Output*. See Section 6.	0..1
<allow_topic_name_filter>	A *Topic* name filter. You may use a comma-separated list to specify more than one filter. **Default:** * (allow all)	0..1
<allow_registered_type_name_filter>	A registered type name filter. You may use a comma-separated list to specify more than one filter. **Default:** * (allow all)	0..1
<deny_topic_name_filter>	A *Topic* name filter that should be denied (excluded). This is applied after the `<allow_stream_name_filter>`. **Default:** empty (not applied)	1

Continued on next page

Table 4.19 – continued from previous page

Tags within <input> and <output> (in <auto_topic_route>) <dds_input> and <dds_output> (in <auto_route>)	Description	Multiplicity
<deny_registered_type_filter>	A registered type name filter that should be denied (excluded). This is applied after the `<allow_registered_type_name_filter>`. **Default:** empty (not applied)	0..1
<creation_mode>	Specifies when to create the StreamReader/StreamWriter. **Default:** IMMEDIATE See Section 4.5.7.	0..1
<datareader_qos> or <datawriter_qos>	Sets the *DataReader* or *DataWriter* QoS. The contents of this tag are specified in the same manner as a *Connext DDS* QoS profile. See Configuring QoS with XML, in the Connext DDS Core Libraries User's Manual. If the tag is not defined, *Routing Service* will use the *Connext DDS* defaults.	0..1
<content_filter>	Defines a SQL content filter for the *DataReader*. **Example:** `<content_filter> <expression> x &gt; 100 </expression> </content_filter>`	0..1

4.5.9 Plugins

All the pluggable components specific to *Routing Service* are configured within the `<plugin_library>` tag. Table 4.20 describes the available tags.

Plug-ins are categorized and configured based on the source language. *Routing Service* supports C/C++ and Java plug-ins. See Section 7 for further information on developing *Routing Service* plug-ins.

Table 4.20: Configuration tags for plug-in libraries

Tags within <plugin_library>	Description	Multiplicity
<adapter_plugin>	Specifies a C/C++ *Adapter* plug-in. See Table 12.16. Attributes • `name`: uniquely identifies an *Adapter* plug-in within a library. This name qualified with the library name represents the plug-in registered name that is referred by `<connection>` tags. See Table 4.6.	0..*
<java_adapter_plugin>	Specifies a Java *Adapter* plug-in. See Table 12.17. Attributes (See `<adapter_plugin>`).	0..*
<transformation_plugin>	Specifies a C/C++ *Transformation* plug-in. See Table 12.16. Attributes • `name`: uniquely identifies an *Transformation* plug-in within a library. This name qualified with the library name represents the plug-in registered name that is referred by `<transformation>` tags. See Section 4.5.6.	0..*
<processor_plugin>	Specifies a C/C++ *Processor* plug-in. See Table 12.16. Attributes • `name`: uniquely identifies an *Processor* plug-in within a library. This name qualified with the library name represents the plug-in registered name that is referred by `<processor>` tags. See Section 4.5.6.	0..*

4.6 Enabling Distributed Logger

Routing Service provides integrated support for *RTI Distributed Logger*.

Distributed Logger is included in *Connext DDS* but it is not supported on all platforms; see the Connext DDS Core Libraries Platform Notes to see which platforms support *Distributed Logger*.

When you enable *Distributed Logger*, *Routing Service* will publish its log messages to *Connext DDS*. 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:

```
<routing_service name="default">
    <administration>
        ...
```

(continues on next page)

(continued from previous page)

```
        <distributed_logger>
            <enabled>true</enabled>
        </distributed_logger>
    </administration>
    ...
</routing_service>
```

For the list of elements that configure *Distributed Logger* see Section 4.5.2. For more details about *Distributed Logger*, see Enabling Distributed Logger in RTI Services.

4.7 Support for Extensible Types

Routing Service includes partial support for the '"Extensible and Dynamic Topic Types for DDS" specification <http://www.omg.org/spec/DDS-XTypes>' from the Object Management Group (OMG). This section assumes that you are familiar with Extensible Types and you have read the *Connext DDS Core Libraries Getting Started Guide Addendum for Extensible Types*.

- *Inputs* and *Outputs* can subscribe to and publish topics associated with final and extensible types.
- You can select the type version associated with a topic route by providing the type description in the XML configuration file. The XML description supports structure inheritance. You can learn more about structure inheritance in the *Connext DDS Getting Started Guide Addendum for Extensible Types*.
- The TypeConsistencyEnforcementQosPolicy can be specified on a per-topic-route basis, in the same way as other QoS policies.
- Within a *DomainParticipant*, a topic cannot be associated with more than one type version. This prevents the same *DomainParticipant* from having two *Route DataReader* or *DataWriter* with different versions of a type for the same *Topic*. To achieve this behavior, create two different *DomainParticipant*, each associating the topic with a different type version.

The type declared in an *Input* is the version return edin the read operations within the installed *Processor* of the parent *Route*, which then can be provided directly to the *Outputs*, as long as they have a compatible type (or a *Transformation* that makes it compatible). An *Input* can subscribe to different-but-compatible types, but those samples are translated to the actual type of the *Input*.

4.7.1 Example: Samples Published by Two Writers of Type A and B, Respectively

```
struct A {
  long x;
};

struct B {
    long x;
    long y;
};
```

Table 4.21: Forwarded data when type in *TopicRoute* is not extended

Samples published by two *DataWriters* of types A and B, respectively	Samples forwarded by a *TopicRoute* for type A in both input and output	Samples received by a B reader
A [x=1]	A [x=1]	B [x=1, y=0]
B [x=10, y=11]	A [x=10]	B [x=10, y=0]

Table 4.22: Forwarded data when type in *TopicRoute* is extended

Samples published by two *DataWriters* of types A and B, respectively	Samples forwarded by a *TopicRoute* for type B in both input and output	Samples received by a B reader
A [x=1]	B [x=1, y=0]	B [x=1, y=0]
B [x=10, y=11]	B [x=10, y=11]	B [x=10, y=11]

4.8 Support for RTI FlatData and Zero Copy Transfer Over Shared Memory

Routing Service supports communication with applications that use RTI FlatData™ and Zero-Copy transfer over shared memory, *only on the subscription side*.

Warning: On the publication side, *Routing Service* will ignore the type annotations for these capabilities and will communicate through the regular serialization and deserialization paths.

To enable *Routing Service* to work with RTI FlatData and Zero-copy transfer over shared memory, you will need to manually define the type in the XML configuration with the proper annotations, and then register this type manually in each *DomainParticipant*. You can use each of these capabilities separately or together.

For further information about these capabilities, see the Sending Large Data section in the *RTI Connext DDS User's Manual*.

4.8.1 Example: Configuration to enable both FlatData and zero-copy transfer over shared memory

```
<dds>
    <types>
        <struct name="Point"
                transferMode="shmem_ref"
                languageBinding="flat_data"
```

(continues on next page)

(continued from previous page)

```
                extensibility= "final">
            <member name="x" type="long"/>
            <member name="y" type="long"/>
        </struct>
    </types>

    <qos_library name="MyQosLib">
        <qos_profile name="ShmemOnly">
            <participant_qos>
                <discovery>
                    <initial_peers>
                        <element>shmem://</element>
                    </initial_peers>
                </discovery>
                <transport_builtin>
                    <mask>SHMEM</mask>
                </transport_builtin>
            </participant_qos>
        </qos_profile>
    </qos_library>

    <routing_service name="FlatDataWithZeroCopy">

        <domain_route>
            <participant name="InputDomain">
                <domain_id>0</domain_id>
                <participant_qos base_name="MyQosLib::ShmemOnly"/>
                <registered_type name="Point" type_name="Point"/>
            </participant>
            <participant name="OutputDomain">
                <domain_id>1</domain_id>
                <registered_type name="Point" type_name="Point"/>
            </participant>

            <session>
                <topic_route>
                    <input participant="InputDomain">
                        <topic_name>PointTopic</topic_name>
                        <registered_type_name>Point</registered_type_name>
                    </input>
                    <output participant="OutputDomain">
                        <topic_name>PointTopic</topic_name>
                        <!-- The output will ignore the FlataData and Zero Copy␣
↪capabilities -->
                        <registered_type_name>Point</registered_type_name>
                    </output>
                </topic_route>
            </session>
        </domain_route>
    </routing_service>
</dds>
```

Chapter 5

Remote Administration

This section provides documentation on *Routing Service* remote administration.

Note: *Routing Service* remote administration is based on the *RTI Remote Administration Platform* described in Section 12.2. We recommend that you read that section before using *Routing Service* remote administration.

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

5.1 Overview

5.1.1 Enabling Remote Administration

By default, remote administration is disabled in *Routing Service*. To enable remote administration, you can use the `<administration>` tag (see Section 4.5.1) or the `-remoteAdministrationDomainId` command-line parameter, which enables remote administration and sets the domain ID for remote communication (see Section 3.1).

5.1.2 Available Service Resources

Table 5.1 lists the public resources specific to *Routing Service*. Each resource identifier is expressed as a hierarchical sequence of identifiers, including parent and target resources. (See Section 12.1.2 for details.)

In the table below, the elements `(rs)`, `(dr)`, `(c)`, `(s)`, `(ar)`, `(r)`, `(i)`, and `(o)` refer 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:

```
<routing_service name="MyRouter">...</routing_service>
```

The resource identifier is:

```
/routing_services/MyRouter
```

In the table, the resource identifier is written as /routing_services/(rs), where (rs) is the routing service name, (dr) is the domain route name, and so on. 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 5.1: Resources and Their Identifiers in *Routing Service*

Resource	Resource Identifier
Service	/routing_services/(rs)
DomainRoute	/routing_services/(rs)/domain_routes/(dr)
Connection or *Participant*	/routing_services/(rs)/domain_routes/(dr)/connections/(c)
Session	/routing_services/(rs)/domain_routes/(dr)/sessions/(s)
AutoRoute or *AutoTopicRoute*	/routing_services/(rs)/domain_routes/(dr)/sessions/(s)/auto_routes/(ar)
Route or *TopicRoute*	/routing_services/(rs)/domain_routes/(dr)/sessions/(s)/routes/(r)
Route Input or DDS *Input*	/routing_services/(rs)/domain_routes/(dr)/sessions/(s)/routes/(r)/inputs/(i)
Route Output or DDS *Output*	/routing_services/(rs)/domain_routes/(dr)/sssions/(s)/routes/(r)/outputs/(i)

Example

This example shows you how to address a resource of each possible resource class in *Routing Service*, using the example configuration in Section 5.3 as a reference. (For a complete reference of the available configuration tags used in *Routing Service*, see Section 4.5.)

Service

Entity with name "MyRouter":

```
<routing_service name="MyRouter">...</routing_service>
```

Resource identifier:

```
/routing_services/MyRouter
```

DomainRoute

Entity with name "MyDomainRoute" in parent "MyRouter":

```
<routing_service name="MyRouter">
    <domain_route name="MyDomainRoute">...</domain_route>
</routing_service>
```

Resource identifier:

```
/routing_services/MyRouter/domain_routes/MyDomainRoute
```

Participant

Entity with name "MyParticipant" in parent "MyDomainRoute":

```
<routing_service name="MyRouter">
    <domain_route name="MyDomainRoute">
        <participant name="Session">...</participant>
    </domain_route>
</routing_service>
```

Resource identifier:

```
/routing_services/MyRouter/domain_routes/MyDomainRoute/connections/
↪MyParticipant
```

Session

Entity with name "MySession" in parent "MyDomainRoute":

```
<routing_service name="MyRouter">
    <domain_route name="MyDomainRoute">
        <session name="MySession">...</session>
    </domain_route>
</routing_service>
```

Resource identifier:

```
/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession
```

AutoTopicRoute (or *AutoRoute*)

Entity with name "MyAutoTopicRoute" in parent "MySession":

```
<routing_service name="MyRouter">
    <domain_route name="MyDomainRoute">
        <session name="MySession">
            <auto_topic_route name="MyAutoTopicRoute">...</auto_topic_route>
        </session>
    </domain_route>
</routing_service>
```

Resource identifier (all on one line):

```
/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/
routes/MyTopicRoute
```

TopicRoute (or *Route*)

Entity with name "MyTopicRoute" in parent "MySession":

```
<routing_service name="MyRouter">
    <domain_route name="MyDomainRoute">
        <session name="MySession">
            <topic_route name="MyTopicRoute">...</topic_route>
        </session>
    </domain_route>
</routing_service>
```

Resource identifier (all on one line):

```
/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/
routes/MyTopicRoute
```

Input

Entity with name "MyInput" in parent "MyTopicRoute":

```
<routing_service name="MyRouter">
    <domain_route name="MyDomainRoute">
        <session name="MySession">
            <topic_route name="MyTopicRoute">
                <input name="MyInput">...</input>
            </topic_route>
        </session>
    </domain_route>
</routing_service>
```

Resource identifier (all on one line):

```
/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/
routes/MyRoute/inputs/MyInput
```

Output

Entity with name "MyOutput" in parent "MyTopicRoute":

```
<routing_service name="MyRouter">
    <domain_route name="MyDomainRoute">
        <session name="MySession">
            <topic_route name="MyTopicRoute">
                <output name="MyOutput">...</output>
            </topic_route>
        </session>
    </domain_route>
</routing_service>
```

Resource identifier (all on one line):

```
/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/
routes/MyRoute/outputs/MyOutput
```

5.1.3 Resource Object Representations

Table 5.2: Resource Representations in *Routing Service*

Resource Representation	Format (all element type definitions are from the file rti_routing_service.xsd)
ddsObjectRepresentation	`<xs:element name="dds" type="ddsRouter"/>`
routerObjectRepresentation	`<xs:element name="routing_service" type="routingService"/>`
domainRouteObjectRepresentation	`<xs:element name="domain_route" type="domainRoute"/>`
connectionObjectRepresentation	`<xs:element name="connection" type="domainRouteConnection"/>`
participantObjectRepresentation	`<xs:element name="participant" type="domainRouteParticipant"/>`
sessionObjectRepresentation	`<xs:element name="session" type="routerSession"/>`
autoRouteObjectRepresentation	`<xs:element name="auto_route" type="autoRoute"/>`
autoTopicRouteObjectRepresentation	`<xs:element name="auto_topic_route" type="autoTopicRoute"/>`
routeObjectRepresentation	`<xs:element name="route" type="route"/>`
topicRouteObjectRepresentation	`<xs:element name="topic_route" type="topicRoute"/>`

Continued on next page

Table 5.2 – continued from previous page

Resource Representation	Format (all element type definitions are from the file rti_routing_service.xsd)
inputObjectRepresentation	`<xs:element name="input" type="routeStreamPort"/>`
outputObjectRepresentation	`<xs:element name="output" type="routeStreamPort"/>`
ddsInputObjectRepresentation	`<xs:element name="input" type="topicRouteInput"/>` `<xs:element name="dds_input" type="topicRouteInput"/>`
ddsOutputObjectRepresentation	`<xs:element name="output" type="topicRouteOutput"/>` `<xs:element name="dds_output" type="topicRouteOutput"/>`

5.2 API Reference

This section documents each remote operation, organized by service resource class.

5.2.1 Remote API Overview

Note: To improve readability, `<SERVICE>` is sometimes used in place of the service resource portion of the resource identifier (e.g., /routing_services/(rs) or/routing_services/MyService). It does not represent valid syntax.

Table 5.3: Remote Interface Overview

Resource	Operation	Description
Service	*CREATE /routing_services/(rs)/domain_route*	Creates a new *DomainRoute*.
	CREATE /routing_services/(rs)/config	Loads a full service configuration.
	GET /routing_services/(rs)	Returns the *Service* configuration.
	UPDATE /routing_services/(rs)	Updates a *Service* object.
	UPDATE /routing_services/(rs)/state	Sets a *Service* state.

Continued on next page

Table 5.3 – continued from previous page

Resource	Operation	Description
	UPDATE /routing_services/(rs):save	Saves the *Service* loaded configuration.
	DELETE /routing_services/(rs)/domain_routes/(dr)	Deletes a *DomainRoute* object.
	DELETE /routing_services/(rs)/config	Returns the *Service* configuration.
	DELETE /routing_services/(rs)	Shuts down the running *Service*.
DomainRoute	*CREATE /routing_services/(rs)/domain_route/(dr)/sessions*	Creates a new *Session*.
	UPDATE /routing_services/(rs)/domain_route/(dr)	Updates a *DomainRoute*.
	UPDATE /routing_services/(rs)/domain_route/(dr)/state	Sets a *DomainRoute* state.
	DELETE /routing_services/(rs)/domain_route/(dr)/sessions/(s)	Deletes a *Session*.
Connection	*UPDATE <SERVICE>/domain_route/connections(c):add_peer*	Adds a list of peers in a *Connection* (a *Participant* in DDS adapter).
	UPDATE <SERVICE>/domain_route/(dr)/connections(c)	Updates a *Connection*.
	DELETE <SERVICE>/domain_route/(dr)/connections(c):remove_peer	Removes a list of peers in a *Connection* (a *Participant* in DDS adapter).
Session	*CREATE <SERVICE>/domain_route/(dr)/sessions/(s)/auto_routes*	Creates a new *AutoRoute*.
	CREATE <SERVICE>/domain_route/(dr)/sessions/(s)/routes	Creates a new *Route*.
	UPDATE <SERVICE>/domain_route/(dr)/sessions(s)	Updates a *Session*.
	UPDATE <SERVICE>/domain_route/(dr)/sessions(s)/state	Sets a *Session* state.
	DELETE <SERVICE>/domain_route/(dr)/sessions/(s)/auto_routes/(ar)	Deletes an *AutoRoute*.
	DELETE <SERVICE>/domain_route/(dr)/sessions/(s)/routes/(r)	Deletes a *Route*.
AutoRoute or *AutoTopicRoute*	*UPDATE <SERVICE>/domain_route/(dr)/sessions/(s)/auto_routes(ar)*	Updates an *AutoRoute*.
	UPDATE <SERVICE>/domain_route/(dr)/sessions/(s)/auto_routes(ar)/state	Sets an *AutoRoute* state.

Continued on next page

Table 5.3 – continued from previous page

Resource	Operation	Description
Route or *TopicRoute*	*UPDATE <SERVICE>/domain_route/(dr)/sessions/(s)/routes(r)*	Updates a *Route*.
	UPDATE <SERVICE>/domain_route/(dr)/sessions/(s)/routes(r)/state	Sets a *Route* state.
Input	*UPDATE <SERVICE>/domain_route/(dr)/sessions/(s)/routes(r)/inputs/(i)*	Updates an *Input* (Connext DDS and non-Connext DDS).
Output	*UPDATE <SERVICE>/domain_route/(dr)/sessions/(s)/routes(r)/outputs/(o)*	Updates an *Output* (Connext DDS and non-Connext DDS).

5.2.2 Service

`CREATE /routing_services/(rs)/domain_routes`

Operation `create_domain_route`

Creates a *DomainRoute* object from its *domainRouteObjectRepresentation* (see Table 5.2).

See *Create Resource* (Section 12.2.3).

- Example

  Create a *DomainRoute* with name "NewDomainRoute" under *Service* "MyRouter", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes

`string_body`	`str\://\"<domain_route name=\"NewDomainRoute\">`
	`...`
	`</domain_route>"`

The newly created object has the resource identifier:

/routing_services/MyRouter/domain_routes/NewDomainRoute

`CREATE /routing_services/(rs)/config`

Operation `load`

Loads a new configuration for the service from its *ddsObjectRepresentation* (see Table 5.2).

If the *Service* is already loaded, this operation will unload it first.

The provided configuration must contain a valid *Service* configuration with the same name that the initial configuration used when the service was first instantiated.

If the operation fails, the service will remain in an unloaded state.

Request body

- `string_body`: a valid *Service* XML configuration document provided as `file://` or `str://`.

Reply body

- Empty.

- Example

  Load a new configuration in *Service* "MyRouter".

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/config

`string_body`	`str://"<dds>`
	`...`
	`<qos_library name="QosLibrary">`
	`...`
	`</qos_library>`
	
	`...`
	`<routing_service name="MyRouter">`
	`...`
	`</routing_service>`
	`</dds>"`

`GET /routing_services/(rs)`

Operation: `get`

Returns a snapshot of the currently loaded full XML configuration as *ddsObjectRepresentation* (see Table 5.2).

See *Get Resource* (Section 12.2.3).

- Example reply body:

```
<routing_service name="MyRouter>
    <administration>...</administration>
    ...
</routing_service>
```

`UPDATE /routing_services/(rs)`

Operation: `update`

Updates the specified *Service* object.

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *routerObjectRepresentation* and contains only parameters valid for the update.

- Example

  Update a *Service* with the name "MyRouter".

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter

`string_body`	`str://\"<routing_service>`
	`...`
	`</routing_service>"`

`UPDATE /routing_services/(rs)/state`

Operation: `set_state`

Sets the state of a *Service* object.

See *Set Resource State* (Section 12.2.3).

Valid requested states:

- `STARTED`
- `STOPPED`
- `PAUSED`
- `RUNNING`

- Example

  Enable a *Service* with the name "MyRouter".

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/state
`octet_body`	`to_cdr_buffer(RTI::Service::EntityStateKind::ENABLED)`

`UPDATE /routing_services/(rs):save`

Operation: `save`

Dumps the currently loaded XML configuration into a file.

The output file is specified by the `save_path` configuration tag. The `save` operation will fail if the `save_path` has not been configured.

Request body

- Empty.

Reply body

- Empty.

`DELETE /routing_services/(rs)/domain_routes/(dr)`

Operation `delete_domain_route`

Deletes the specified *DomainRoute*.

See *Delete Resource* (Section 12.2.3).

`DELETE /routing_services/(rs)/config`

Operation `unload`

Unloads the current configuration of the service. If the *Service* is enabled, this operation will disable it first. Upon a successful request, the service will remain in an unloaded state and no other operations can be made until a configuration is loaded.

Request body

- Empty.

Reply body

- Empty.

`DELETE /routing_services/(rs)`

Operation `shutdown`

Initiates the shutdown sequence on the process where the *Service* object runs.

- If *Service* runs as a process executed by the shipped executable in the *RTI Connext DDS* installation, the process will exit upon receipt of the command.
- If *Service* is instantiated as a library in your application, the service instance will notify the installed remote shutdown hook.

In both cases, right before executing the shutdown sequence, *Service* will send a reply indicating the result of the operation. Note that if the operation returns successfully, the reply may be lost and never received by remote clients, since all the contained entities are deleted, including the *RTI Remote Administration Platform* entities.

This operation can be invoked at any time during the lifecycle of the service.

Request body

- Empty.

Reply body

- Empty.

5.2.3 DomainRoute

`CREATE /routing_services/(rs)/domain_routes/(dr)/sessions`

Operation: `create_session`

Creates a *Session* object from its *sessionObjectRepresentation* (see Table 5.2).

See *Create Resource* (Section 12.2.3).

- Example

  Create a *Session* with the name "NewSession" under the *DomainRoute* "MyDomainRoute", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions

`string_body`	`str://"<session name="NewSession">`
	`...`
	`</session>"`

  The newly created object has the resource identifier:

<SERVICE>/domain_routes/NewDomainRoute/sessions/NewSession

`UPDATE /routing_services/(rs)/domain_routes/(dr)`

Operation: `update`

Updates the specified *DomainRoute* object.

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *domainRouteObjectRepresentation* and contains only parameters valid for the update.

- Example

  Update a *DomainRoute* with the name "MyDomainRoute" under the *Service* "MyRouter", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute

`string_body`	`str://"<domain_route>`
	`...`
	`</domain_route>"`

`UPDATE /routing_services/(rs)/domain_routes/(dr)/state`

Operation: `set_state`

Sets the state of a *DomainRoute* object.

See *Set Resource State* (Section 12.2.3).

Valid requested states:

- `ENABLED`
- `DISABLED`

- Example

  Enable a *DomainRoute* with the name "MyDomainRoute" under the *Service* "MyRouter".

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routers/MyDomainRoute/state
`octet_body`	`to_cdr_buffer(RTI::Service::EntityStateKind::ENABLED)`

`DELETE /routing_services/(rs)/domain_routes/(dr)/sessions/(s)`

Operation `delete_session`

Deletes the specified *Session*.

See *Delete Resource* (Section 12.2.3).

Request body

- Empty.

Reply body

- Empty.

5.2.4 Connection

`UPDATE <SERVICE>/domain_routes/(dr)/connections/(c):add_peer`

Operation `add_peer`

Adds a list of peers to the specified *Connection*.

The *Connection* implementation shall refer to a `<participant>` object.

Request body

- `string_body`: A comma-separated list of peer descriptors, as described in peer descriptor format.
- Example peer descriptor list:

```
updv4://10.2.0.1,udpv4://239.255.0.1
```

Reply body

- Empty.

`UPDATE <SERVICE>/domain_routes/(dr)/connections/(c)`

Operation: `update`

Updates the specified *Connection* object.

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *connectionObjectRepresentation* or *participantObjectRepresentation*, and contains only parameters valid for the update.

- Example

  Update a *Connection* with the name "MyConnection" under the *DomainRoute* "MyDomainRoute", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/ connections/MyConnectiom

`string_body`	`str://"<connection>`
	`...`
	`</connection>"`

`UPDATE <SERVICE>/domain_routes/(dr)/connections/(c):remove_peer`

Operation `remove_peer`

Removes a list of peers from the specified *Connection*.

The *Connection* implementation shall refer to a `<participant>` object.

Request body

- `string_body`: A comma-separated list of peer descriptors, as described in peer descriptor format.
- Example peer descriptor list:

```
updv4://10.2.0.1,udpv4://239.255.0.1
```

Reply body

- Empty.

5.2.5 Session

`CREATE <SERVICE>/domain_routes/(dr)/sessions/(s)/auto_routes`

Operation: `create_auto_route`

Creates an *AutoRoute* or *AutoTopicRoute* object from its *autoRouteObjectRepresentation* or *autoTopicRouteObjectRepresentation* (see Table 5.2).

See *Create Resource* (Section 12.2.3).

- Example

  Create an *AutoRoute* with the name "NewAutoRoute" under the *Session* "MySession", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession/auto_routes

`string_body`	`str://"<auto_route name="NewAutoRoute">`
	`...`
	`</auto_route>"`

The newly created object has the resource identifier:

/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession/auto_routes/NewAutoRoute

---

`CREATE <SERVICE>/domain_routes/(dr)/sessions/(s)/routes`

Operation: `create_route`

Creates a *Route* or *TopicRoute* object from its *routeObjectRepresentation* or *topicRouteObjectRepresentation* (see Table 5.2).

See *Create Resource* (Section 12.2.3).

- Example

Create a *Route* with the name "NewRoute" under the *Session* "MySession", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession/routes

`string_body`	`str://"<route name="NewRoute">`
	`...`
	`</route>"`

The newly created object has the resource identifier:

/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession/routes/NewRoute

---

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)`

Operation: `update`

Updates the specified *Session* object.

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *sessionObjectRepresentation* and contains only parameters valid for update.

- Example

  Update a *Session* with the name "MySession" under the *DomainRoute* "MyDomainRoute", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession

`string_body`	`str://"<session>`
	`...`
	`</session>"`

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)/state`

Operation: `set_state`

Sets the state of a *Session* object.

See *Set Resource State* (Section 12.2.3).

Valid requested states:

- `ENABLED`
- `DISABLED`

- Example

  Enable a *Session* with the name "MySession" under the *DomainRoute* "MyDomainRoute".

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routers/MyDomainRoute/ sessions/MySession/state
`octet_body`	`to_cdr_buffer(RTI::Service::EntityStateKind::ENABLED)`

`DELETE <SERVICE>/domain_routes/(dr)/sessions/(s)/auto_routes/(ar)`

Operation `delete_auto_route`

Deletes the specified *AutoRoute*.

See *Delete Resource* (Section 12.2.3).

`DELETE <SERVICE>/domain_routes/(dr)/sessions/(s)/routes/(r)`

Operation `delete_route`

Deletes the specified *Route*.

See *Delete Resource* (Section 12.2.3).

5.2.6 AutoRoute

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)/auto_routes/(ar)`

Operation: `update`

Updates the specified *AutoRoute* or *AutoTopicRoute* object.

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *autoRouteObjectRepresentation* or *autoTopicRouteObjectRepresentation*, and contains only parameters valid for the update.

- Example

Update an *AutoRoute* with the name "MyAutoRoute" under the *Session* "MySession", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/auto_routes/MyAutoRoute

`string_body`	`str://"<auto_route>`
	`...`
	`</auto_route>"`

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)/auto_routes/(ar)/state`

Operation: `set_state`

Sets the state of an *AutoRoute* object.

See *Set Resource State* (Section 12.2.3).

Valid requested states:

- `ENABLED`
- `DISABLED`
- `RUNNING`
- `PAUSED`

- Example

  Pause an *AutoRoute* with the name "MyAutoRoute" under the *Session* "MySession".

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routers/MyDomainRoute/sessions/MySession/auto_routes/MyAutoRoutestate
`octet_body`	`to_cdr_buffer(RTI::Service::EntityStateKind::PAUSED)`

5.2.7 Route

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)/routes/(r)`

Operation: `update`

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *routeObjectRepresentation* or *topicRouteObjectRepresentation*, and contains only parameters valid for the update.

- Example

  Update a *Route* with the name "MyRoute" under the *Session* "MySession", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession/routes/MyRoute

`string_body`	`str://"<route>`
	`...`
	`</route>"`

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)/routes/(r)/state`

Operation: `set_state`

Sets the state of a *Route* object.

See *Set Resource State* (Section 12.2.3).

Valid requested states:

- `ENABLED`
- `DISABLED`
- `RUNNING`
- `PAUSED`

- Example

  Pause a *Route* with the name "MyRoute" under the *Session* "MySession".

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routers/MyDomainRoute/ sessions/MySession/routes/MyRoutestate
`octet_body`	`to_cdr_buffer(RTI::Service::EntityStateKind::PAUSED)`

5.2.8 Input/Output

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)/routes/(r)/inputs(i)`

Operation: `update`

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *routeInputObjectRepresentation* or *topicRouteInputObjectRepresentation*, and contains only parameters valid for the update.

- Example

  Update *Input* with the name "MyInput" under the *Route* "MyRoute", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession/routes/MyRoute/inputs/MyInput

`string_body`	`str://"<input>`
	`...`
	`</input>"`

`UPDATE <SERVICE>/domain_routes/(dr)/sessions/(s)/routes/(r)/outputs(i)`

Operation: `update`

See *Update Resource* (Section 12.2.3).

The expected XML configuration is a subset of *routeOutputObjectRepresentation* or *topicRouteOutputObjectRepresentation*, and contains only parameters valid for the update.

- Example

  Update *Output* with the name "MyOutput" under the *Route* "MyRoute", with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	CREATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/ sessions/MySession/routes/MyRoute/outputs/MyOutput

`string_body`	`str://"<output>`
	`...`
	`</output>"`

5.3 Example: Configuration Reference

This configuration example shows how individual commands would apply to a valid *Routing Service* configuration.

```
<?xml version="1.0"?>
<dds>
    <routing_service name="MyRouter">
        <domain_route name="MyDomainRoute">
            <participant name="MyParticipant">
               <domain_id>0</domain_id>
            </participant>
            <connection name="MyConnection">
            </connection>
            ... <!-- other connections/participants -->

            <session name="MySession">
                <auto_route name="MyAutoRoute">
                <publish_with_original_timestamp>true</publish_with_original_timestamp>
                    ...
                    <input name="MyInput">
                        ...
                        <property>
                            ...
                        </property>
                    </input>
                    <output name="MyOutput">
                        ...
                        <property>
                            ...
                        </property>
                    </output>
                </auto_route>
                <auto_topic_route name="MyAutoTopicRoute">
                    <publish_with_original_info>true</publish_with_original_info>
                    ...
                    <input name="MyInput">
                        ...
                        <datareader_qos>
                            ...
                        </datareader_qos>
                    </input>
                    <output name="MyOutput">
                        ...
                        <datawriter_qos>
                            ...
                        </datawriter_qos>
                    </output>
                </auto_topic_route>
                ... <!-- other auto (Topic) routes -->
                <route name="MyRoute">
                    <route_types>true</route_types>
                    <input name="MyInput">
```

(continues on next page)

(continued from previous page)

```
                    ...
                    <property>
                        ...
                    </property>
                </input>
                ... <!-- other inputs -->
                <output name="MyOutput">
                    ...
                    <property>
                        ...
                    </property>
                </output>
                ... <!-- other outputs -->
            </route>
            ... <!-- other (Topic) routes -->
            <topic_route name="MyTopicRoute">
                <route_types>true</route_types>
                ...
                <input name="MyInput">
                    ...
                    <datareader_qos>
                        ...
                    </datareader_qos>
                </input>
                ... <!-- other inputs -->
                <output name="MyOutput">
                    ...
                    <datawriter_qos>
                        ...
                    </datawriter_qos>
                </output>
                ... <!-- other outputs -->
            </topic_route>
        </session>
        ... <!-- other sessions -->
    </domain_route>
    ... <!-- other domain routes -->
</routing_service>
</dds>
```

Chapter 6

Monitoring

This section provides documentation on *Routing Service* remote monitoring.

Note: *Routing Service* monitoring is based on the *Monitoring Distribution Platform* described in Section 12.3. We recommend that you read Section 12.3 before using *Routing Service* monitoring.

6.1 Overview

6.1.1 Enabling Service Monitoring

By default, monitoring is disabled in *Routing Service*. To enable monitoring you can use the `<monitoring>` tag (see Section 4.5.1) or the `-remoteMonitoringDomainId` command-line parameter, which enables remote monitoring and sets the domain ID for data publication (see Section 3.1).

6.1.2 Monitoring Types

The available *Keyed Resource* classes and their types that can be present in the distribution monitoring topics are listed in Table 6.1. The complete type relationship is shown in Figure 6.1.

Table 6.1: *Routing Service Keyed Resources*

Keyed Resource Class	Config	Event	Periodic
Service	`ServiceConfig`	`ServiceEvent`	`ServicePeriodic`
DomainRoute	`DomainRouteConfig`	`DomainRouteEvent`	`DomainRoutePeriodic`
Session	`SessionConfig`	`SessionEvent`	`SessionPeriodic`
AutoRoute/AutoTopicRoute	`AutoRouteConfig`	`AutoRouteEvent`	`AutoRoutePeriodic`
Route/TopicRoute	`RouteConfig`	`RouteEvent`	`RoutePeriodic`
Input	`InputConfig`	`InputEvent`	`InputPeriodic`
Output	`OutputConfig`	`OutputEvent`	`OutputPeriodic`

All the type definitions for *Routing Service* monitoring information are in `[NDDSHOME]/resource/`

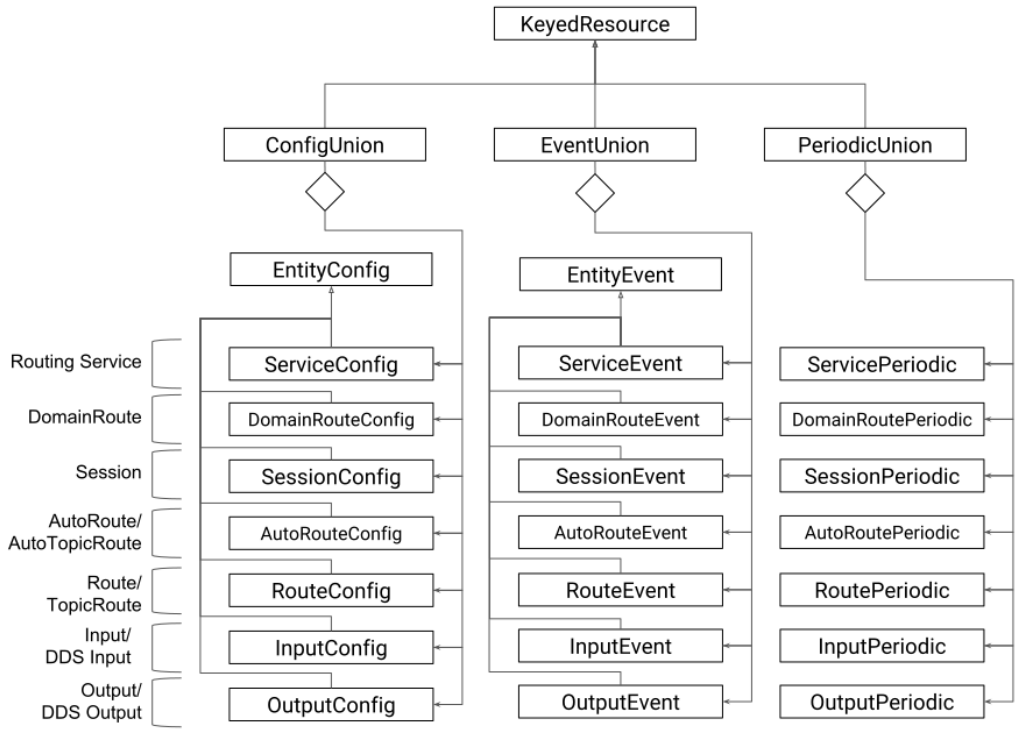


Figure 6.1: Keyed Resource Types for *Routing Service* monitoring

`idl/ServiceCommon.idl` and `[NDDSHOME]/resource/idl/RoutingServiceMonitoring.idl`.

Routing Service creates a *DataWriter* for each distribution *Topic*. All *DataWriters* are created from a single *Publisher*, which is created from a dedicated *DomainParticipant*. See Section 4.5.1 for details on configuring the QoS for these entities.

6.2 Monitoring Metrics Reference

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

6.2.1 Service

Listing 6.1: *Routing Service* Types

```
@mutable @nested
struct ServiceConfig : Service::Monitoring::EntityConfig {
    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 {
```

(continues on next page)

(continued from previous page)

```
};

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

Table 6.2: `ServiceConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 12.13.
application_name	Name of the *Routing Service* instance. The application name is provided through: • `appName` command-line option when run as executable. • `ServiceProperty::application_name` field when run as a library.
application_guid	GUID of the *Routing Service* instance. Unique across all service instances.
host	See Table 12.9.
process	See Table 12.11.

Table 6.3: `ServiceEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.14.

Table 6.4: `ServicePeriodic`

Field Name	Description
host	See Table 12.10.
process	See Table 12.12.

6.2.2 DomainRoute

Listing 6.2: *DataReader* Types

```
@mutable @nested
struct ConnectionConfigInfo {
    BoundedString name;
    AdapterClassKind class;
```

(continues on next page)

(continued from previous page)

```
        BoundedString plugin_name;
        XmlString configuration;
    };
    @mutable @nested
    struct ConnectionEventInfo {
        BoundedString name;
        @optional Service::BuiltinTopicKey participant_key;
    };

    @mutable @nested
    struct DomainRouteConfig : Service::Monitoring::EntityConfig {
        @optional sequence<ConnectionConfigInfo> connections;
    };

    @mutable @nested
    struct DomainRouteEvent : Service::Monitoring::EntityEvent {
        @optional sequence<ConnectionEventInfo> connections;
    };

    @mutable @nested
    struct DomainRoutePeriodic {
        @optional Service::Monitoring::StatisticVariable in_samples_per_sec;
        @optional Service::Monitoring::StatisticVariable in_bytes_per_sec;
        @optional Service::Monitoring::StatisticVariable out_samples_per_sec;
        @optional Service::Monitoring::StatisticVariable out_bytes_per_sec;
        @optional Service::Monitoring::StatisticVariable latency_millisec;
    };
```

Table 6.5: `DomainRouteConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 12.13.
connections	Sequence of `ConnectionInfo` objects, one for each *Connection* inside the *DataReader*. See Table 6.6.

Table 6.6: `ConnectionInfo`

Field Name	Description
name	Name of the *Connection* instance, as specified in the `name` attribute of the corresponding configuration tag.
class	Indicates the adapter class as `AdapterClassKind`: • DDS_ADAPTER_CLASS: The *Connection* object is a DDS adapter connection, hence it corresponds to a `<participant>` element. • GENERIC_ADAPTER_CLASS: The *Connection* object is a custom, generic adapter connection, hence it corresponds to a `<connection>` element.
plugin_name	Name of the adapter plugin as specified in the `plugin_name` attribute of the corresponding configuraiton tag. For the DDS adapter, this field has the constant value of `rti.routingservice.adapters.dds`.
configuration	String representation of the XML configuration of the object.

Table 6.7: `DomainRouteEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.14.

Table 6.8: `DomainRoutePeriodic`

Field Name	Description
in_samples_per_sec	Statistic variable that provides information about the input samples per second as an aggregation of the same metric across the contained *Sessions*.
in_bytes_per_sec	Statistic variable that provides information about the input bytes per second as an aggregation of the same metric across the contained *Sessions*.
output_samples_per_sec	Statistic variable that provides information about the output samples per second as an aggregation of the same metric across the contained *Sessions*.
output_bytes_per_sec	Statistic variable that provides information about the output bytes per second as an aggregation of the same metric across the contained *Sessions*.
latency_millisec	Statistic variable that provides information about the latency in milliseconds as an aggregation of the same metric across the contained *Sessions*.

6.2.3 Session

Listing 6.3: *Session* Types

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

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

@mutable @nested
struct SessionPeriodic {
    @optional Service::Monitoring::StatisticVariable in_samples_per_sec;
    @optional Service::Monitoring::StatisticVariable in_bytes_per_sec;
    @optional Service::Monitoring::StatisticVariable out_samples_per_sec;
    @optional Service::Monitoring::StatisticVariable out_bytes_per_sec;
    @optional Service::Monitoring::StatisticVariable latency_millisec;
};
```

Table 6.9: `SessionConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 12.13.

Table 6.10: `SessionEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.14.

Table 6.11: `SessionPeriodic`

Field Name	Description
in_samples_per_sec	Statistic variable that provides information about the input samples per second as an aggregation of the same metric across the contained *Routes*/*TopicRoutes*.
in_bytes_per_sec	Statistic variable that provides information about the input bytes per second as an aggregation of the same metric across the contained *Routes*/*TopicRoutes*.
output_samples_per_sec	Statistic variable that provides information about the output samples per second as an aggregation of the same metric across the contained *Routes*/*TopicRoutes*.
output_bytes_per_sec	Statistic variable that provides information about the output bytes per second as an aggregation of the same metric across the contained *Routes*/*TopicRoutes*.
latency_millisec	Statistic variable that provides information about the latency in milliseconds as an aggregation of the same metric across the contained *Routes*/*TopicRoutes*.

6.2.4 AutoRoute

Listing 6.4: *AutoRoute*/*AutoTopicRoute* Types

```
@mutable @nested
struct AutoRouteStreamPortInfo {
    XmlString configuration;
};

@mutable @nested
struct AutoRouteConfig : Service::Monitoring::EntityConfig {
    @optional AutoRouteStreamPortInfo input;
    @optional AutoRouteStreamPortInfo output;
};

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

@mutable @nested
struct AutoRoutePeriodic  {
    @optional Service::Monitoring::StatisticVariable in_samples_per_sec;
    @optional Service::Monitoring::StatisticVariable in_bytes_per_sec;
    @optional Service::Monitoring::StatisticVariable out_samples_per_sec;
    @optional Service::Monitoring::StatisticVariable out_bytes_per_sec;
    @optional Service::Monitoring::StatisticVariable latency_millisec;
    int64 route_count;
};
```

Table 6.12: `AutoRouteConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 12.13.
input	See Table 6.13.
output	See Table 6.13.

Table 6.13: `AutoRouteStreamPortInfo`

Field Name	Description
configuration	String representation of the XML configuration of the object.

Table 6.14: `AutoRouteEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.14.

Table 6.15: `AutoRoutePeriodic`

Field Name	Description
in_samples_per_sec	Statistic variable that provides information about the input samples per second as an aggregation of the same metric across all current *Routes/TopicRoutes* created from this *AutoRoute/AutoTopicRoute*.
in_bytes_per_sec	Statistic variable that provides information about the input bytes per second as an aggregation of the same metric across all current *Routes/TopicRoutes* created from this *AutoRoute/AutoTopicRoute*.
output_samples_per_sec	Statistic variable that provides information about the output samples per second as an aggregation of the same metric across all current *Routes/TopicRoutes* created from this *AutoRoute/AutoTopicRoute*.
output_bytes_per_sec	Statistic variable that provides information about the output bytes per second as an aggregation of the same metric across all current *Routes/TopicRoutes* created from this *AutoRoute/AutoTopicRoute*.
latency_millisec	Statistic variable that provides information about the latency in milliseconds as an aggregation of the same metric across all current *Routes/TopicRoutes* created from this *AutoRoute/AutoTopicRoute*.
route_count	Current number of *Routes/TopicRoutes* created from this *AutoRoute/AutoTopicRoute*.

6.2.5 Route

Listing 6.5: *Route/TopicRoute* Types

```
@mutable @nested
struct RouteConfig : Service::Monitoring::EntityConfig {
    @optional Service::Monitoring::ResourceGuid auto_route_guid;
};

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

@mutable @nested
struct RoutePeriodic {
    @optional Service::Monitoring::StatisticVariable in_samples_per_sec;
    @optional Service::Monitoring::StatisticVariable in_bytes_per_sec;
    @optional Service::Monitoring::StatisticVariable out_samples_per_sec;
    @optional Service::Monitoring::StatisticVariable out_bytes_per_sec;
    @optional Service::Monitoring::StatisticVariable latency_millisec;
};
```

Table 6.16: `RouteConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 12.13.
auto_route_guid	GUID of the *AutoRoute/AutoTopicRoute* from which this *Route/TopicRoute* was created. This field is set to zero for standalone routes.

Table 6.17: `RouteEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.14.

Table 6.18: `RoutePeriodic`

Field Name	Description
in_samples_per_sec	Statistic variable that provides information about the input samples per second as an aggregation of the same metric across its contained *Inputs*.
in_bytes_per_sec	Statistic variable that provides information about the input bytes per second as an aggregation of the same metric across its contained *Inputs*.
output_samples_per_sec	Statistic variable that provides information about the output samples per second as an aggregation of the same metric across its contained *Outputs*.
output_bytes_per_sec	Statistic variable that provides information about the output bytes per second as an aggregation of the same metric across its contained *Outputs*.
latency_millisec	Statistic variable that provides information about the latency in milliseconds for the route. The latency in a route refers to the total time elapsed during the forwarding of a sample, which includes reading, processing, and writing.
route_count	Current number of *Routes*/*TopicRoutes* created from this *AutoRoute*/*AutoTopicRoute*.

6.2.6 Input/Output

Listing 6.6: *Input*/*Output* Types

```
@mutable @nested
struct TransformationInfo {
    BoundedString plugin_name;
    XmlString configuration;
};

@mutable @nested
struct StreamPortConfig : Service::Monitoring::EntityConfig {
    BoundedString stream_name;
    BoundedString registered_type_name;
    BoundedString connection_name;
    @optional TransformationInfo transformation;
};

@mutable @nested
struct StreamPortEvent : Service::Monitoring::EntityEvent{
    @optional Service::BuiltinTopicKey endpoint_key;
};

@mutable @nested
struct StreamPortPeriodic {
    @optional Service::Monitoring::StatisticVariable samples_per_sec;
    @optional Service::Monitoring::StatisticVariable bytes_per_sec;
};

/*
```

(continues on next page)

(continued from previous page)

```
 * Input
 */
@mutable @nested
struct InputConfig : StreamPortConfig {
};

@mutable @nested
struct InputEvent: StreamPortEvent {
};

@mutable @nested
struct InputPeriodic : StreamPortPeriodic {
};

/*
 * Output
 */
@mutable @nested
struct OutputConfig : StreamPortConfig {
};

@mutable @nested
struct OutputEvent: StreamPortEvent {
};

@mutable @nested
struct OutputPeriodic : StreamPortPeriodic {
};
```

Table 6.19: `InputConfig` and `OutputConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 12.13.
stream_name	Input/output stream name as specified in the configuration. For DDS *Inputs*/*Outputs*, this value matches the underlying *Topic* name.
registered_type_name	Input/Output registered type name. This is the name used to register the type of the input/output stream.
connection_name	Name of the *Connection* from which the *Input*/*Output* is created. The value of this field can be used to determine the adapter plugin (DDS or generic) from which the underlying *StreamReader*/*StreamWriter* are created.
transformation	Optional field. If present, it provides information about the installed *Transformation*. See Table 6.20. For *Inputs*, this field will never be present.

Table 6.20: `TransformationInfo`

Field Name	Description
plugin_name	Name of the adapter plugin as specified in the `plugin_name` attribute of the corresponding configuration tag.
configuration	String representation of the XML configuration of the object.

Table 6.21: `InputEvent` and `OutputEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.14.

Table 6.22: `InputPeriodic` and `OutputPeriodic`

Field Name	Description
samples_per_sec	Statistic variable that provides information about the samples per second provided by this input/output: • If the resource is *Input*, this field provides the value of the samples returned by the underlying StreamReader::read() operation. • If the resource is *Output*, this field provides the value of the samples provided to the underlying StreamWriter::write() operation.
bytes_per_sec[1]	Statistic variable that provides information about the bytes per second provided by this input/output. The bytes refer only to the serialized samples, excluding protocol headers (RTPS, UDP, etc).

[1] The throughput measured in bytes can only be computed if the samples are *DynamicData* samples. If not, only the throughput, measured in samples per second, is available. This statement applies to all the statistic variables described in this chapter that measure throughput in bytes per second.

Chapter 7

Software Development Kit

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

The SDK is divided in the following modules:

- *RTI Routing Service* Service API: This module offers a set of APIs that allow you to instantiate *Routing Service* instances in your application. This allows you to run *Routing Service* as a library, as described in Section 3.2.
- *RTI Routing Service* Adapter API: *Adapter*s are pluggable components that allow *Routing Service* to consume and produce data for different data domains (e.g. *Connext DDS*, MQTT, raw Socket, etc.). This module offers a set of pluggable APIs to develop custom *Adapter*s, which you can use through shared libraries or through the *Service API*. By default, *Routing Service* is distributed with a builtin DDS adapter that is part of the service library.
- *RTI Routing Service* Processor API: *Processor*s are event-oriented pluggable components that allow you to control the forwarding process that occurs within a *Route*. This module offers a set of pluggable APIs to develop custom *Processor*s, which you can use through shared libraries or through the *Service API*.
- *RTI Routing Service* Transformation API: *Transformation*s are data-oriented pluggable components that allow you to perform conversions of the representation and content of the data that goes through *Routing Service*. This module offers a set of pluggable APIs to develop custom *Transformation*s, which you can use through shared libraries or through the Service API.

Table 7.1 shows which modules are available for each API, along with links to the API documentation.

Table 7.1: API Documentation for the SDK

Language API	Available Modules
RTI Routing Service C API	• Service • Adapter • Processor • Transformation
RTI Routing Service C++ API	• Service • Adapter • Processor • Transformation
RTI Routing Service Java API	• Service • Adapter

Chapter 8

Propagating Content Filters

Routing Service can be configured to propagate the content filter information associated with user *DataReaders* to the user *DataWriters*.

When this functionality is enabled, the user *DataWriters* receive information about the data sets subscribed to by the user *DataReaders*. The *DataWriters* can use that information to do writer-side filtering[1] and propagate only the samples belonging to the subscribed data sets. This results in more efficient bandwidth usage as well as in less CPU consumption in the *Routing Service* instances and user *DataReaders*.

Figure 8.1 shows a scenario where communication between *DataWriters* and *DataReaders* is relayed through one or more *Routing Services* that do not propagate content filters. The user *DataWriters* will send on the wire all the samples they publish, since they cannot make assumptions about what the user *DataReaders* want. This default behavior incurs unnecessary bandwidth and CPU utilization since the filtering will occur on the DDS *DataWriter* $\mathrm{SW_N}$.

Enabling filter propagation makes it possible to perform writer-side filtering from the user *DataWriters*, since they receive a composed filter that represents the data set subscribed to by all the user *DataReaders*, as shown in Figure 8.2.

8.1 Enabling Filter Propagation

Filter propagation is disabled by default in *Routing Service*. You can enable filter propagation with the `<filter_propagation>` tag available under the *TopicRoute* configuration (see Section 4.5.6) and *AutoTopicRoute* configuration (see Section 4.5.8).

8.2 Filter Propagation Behavior

Without filter propagation, the only way to enforce writer-side filtering in a scenario involving one or more *Routing Services* between the user *DataWriters* and user *DataReaders* is by statically configuring the content filter individually for each DDS *StreamReader*. This method has two main disadvantages:

[1] The ability to perform writer-side filtering is subject to some restrictions. For the sake of this discussion, we will assume that the configuration of *DataReaders*, *DataWriters*, and *Routing Services* is such that writer-side filtering is allowed

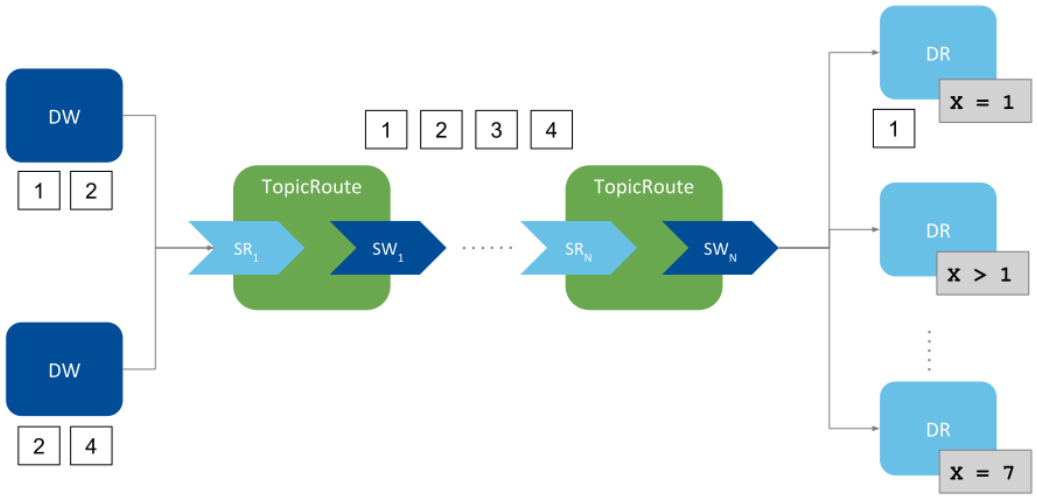


Figure 8.1: Without propagation, user *DataWriters* send all the samples; filtering occurs on the last route's *StreamWriter*

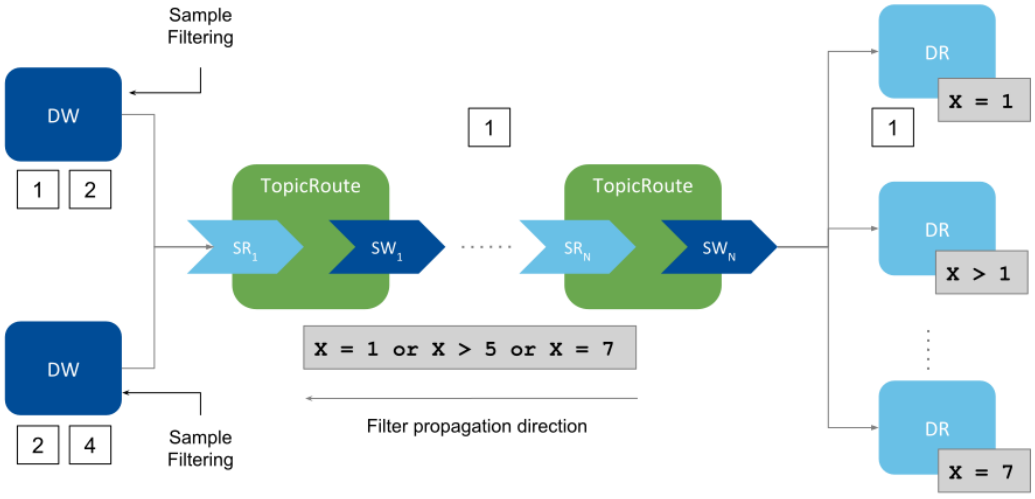


Figure 8.2: With propagation, user *DataWriters* receive a composed filter that allows writer-side filtering, thus sending only the samples of interest to the *DataReaders*

1. It requires knowing beforehand the data set subscribed to by the user *DataReaders*.
2. The filters in the *StreamReaders* are not automatically updated based on changes to the filters in the user *DataReaders*. This may affect not only bandwidth utilization but also correctness. For example, a user *DataReader* may not receive a sample because it has been filtered out by one of the *StreamReaders*.

Filter propagation can address the above issues by dynamically updating the *StreamReaders* filters. The composed filter associated with a *StreamReader* in a *Route* is built by aggregating the filter information associated with all *DataReaders* that match the *Route*'s StreamWriter, as shown in Figure 8.3.

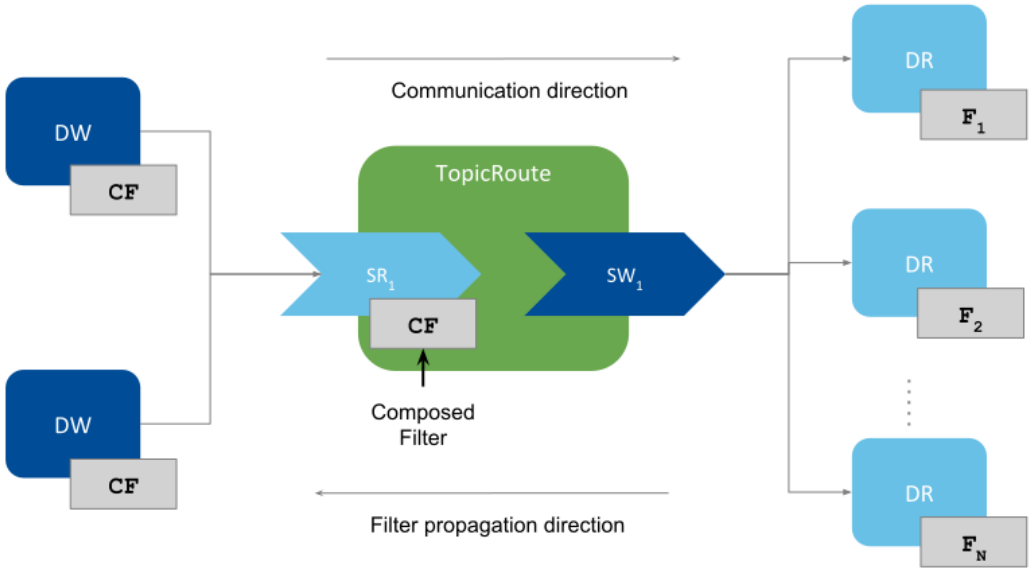


Figure 8.3: Filter Propagation Through Routing Service

The composed filter (CF) is the union of the matching *DataReaders* filters; it allows passing any sample that passes at least one of the *DataReader* filters.

$$CF = F_1 \cup F_2 ... \cup F_N$$

For the SQL filter, the union operator is OR:

$$CF_{SQL} = F_{SQL1} \cup F_{SQL2} ... \cup F_{SQLN}$$

Filter propagation occurs within a *Route* as follows: the *Route* output *StreamWriter* gathers the filter information coming from all of its matching *DataReaders* and provides the resulting composed filter to the *Route* input *StreamReader*, whose *DataReader* is responsible for sending this information to all of its matching *DataWriters*.

8.3 Filter Propagation Events

The following events will cause a *StreamReader*'s filter to be updated and propagated:

- *Route StreamReader creation*: The initial filter is set to the *stop-band* filter, which is a special kind of filter that does not let any sample pass. This filter is propagated upon *StreamReader* creation and it will remain unchanged until a matching *DataReader* to the *Route StreamWriter* is discovered.
- *Discovery of a matching DataReader in a DataReader*: The filter of the discovered *DataReader* will be aggregated to the existing *StreamReader*'s filter, which will be propagated after being updated. If the discovered *DataReader* does not have a filter (subscribes to all the samples) or it has a non-SQL filter, the *StreamReader*'s filter is set to the *all-pass* filter (a special filter that lets all sample pass). The all-pass filter will remain set until there are no matching *DataReaders* to the *Route StreamWriter* without a filter or with a non-SQL filter.
- *A matching DataReader changes its filter, either in the expression or in the parameters*: The *StreamReader*'s filter is updated to incorporate the latest changes and is propagated afterwards.

8.4 Restrictions

Filter propagation cannot be enabled when:

- Using *Routes* or *AutoRoutes*, since they are meant to work with other adapters different than the builtin DDS one.
- A transformation is present in the *TopicRoute*'s output.
- Using remote administration, if the *TopicRoute* was enabled and started with filter propagation initially disabled.
- If the *StreamReader*'s *ContentFilter* class is not the builtin SQL filter. Filter propagation is not currently supported with other filter classes.

Chapter 9

Topic Query Support

Routing Service is fully compatible with *TopicQueries* (see Topic Queries in the *RTI Connext DDS Core Libraries User's Manual*). You can enable this functionality in *TopicRoutes* and *AutoTopicRoutes* with two different query modes:

- **Dispatch mode**: The *TopicRoute*'s *DataWriter* configured with TRANSIENT_LOCAL durability will accept matching *TopicQueries* and dispatch them from its own sample cache.
- **Propagation mode**: *TopicQueries* are propagated from the user *DataReaders* to the user *DataWriters*. These *DataWriters* will be the final endpoints that dispatch the propagated *TopicQueries*.

Routing Service allows propagating *TopicQueries* from *DataReaders* to *DataWriters* acting as a proxy of *TopicQueries*. *Routing Service* supports *TopicQuery* proxy in either of the above modes. It is not possible to enable both modes within the same *TopicRoute*. However, you can create multiple *TopicRoutes*/*AutoTopicRoutes* with different *TopicQuery* proxy modes.

You can enable a *TopicQuery* proxy with the `<topic_query_proxy>` tag available under the *TopicRoute* configuration (see Section 4.5.6) and *AutoTopicRoute* configuration (see Section 4.5.8).

The following sections describe the *Routing Service* proxy modes. Figure 9.1 summarizes the symbols you will see in the figures that illustrate the modes' behaviors.

9.1 Dispatch Mode

Dispatch mode refers to enabling *TopicQuery* dispatch in a TRANSIENT_LOCAL *TopicRoute*'s *DataWriter*. This is done by configuring its TopicQueryDispatchQosPolicy. It no different than enabling a *TopicQuery* for a *DataWriter* in a user application.

Figure 9.2 shows a simple scenario. A *TopicQuery* (TQ_n) issued by a user *DataReader* (DR_n) will be received by the *TopicRoute*'s *StreamWriter*. The *StreamWriter* will process the *TopicQuery* and dispatch it, providing the corresponding samples from the available history in the *StreamWriter*. As a result, the user *DataReader* will receive live samples (S_{Live}) and *TopicQuery* samples (S_{TQ}).

Dispatch mode can be useful when the user *DataWriter* on the publication side is part of an application with low-resources requirements, such as low power consumption and small memory

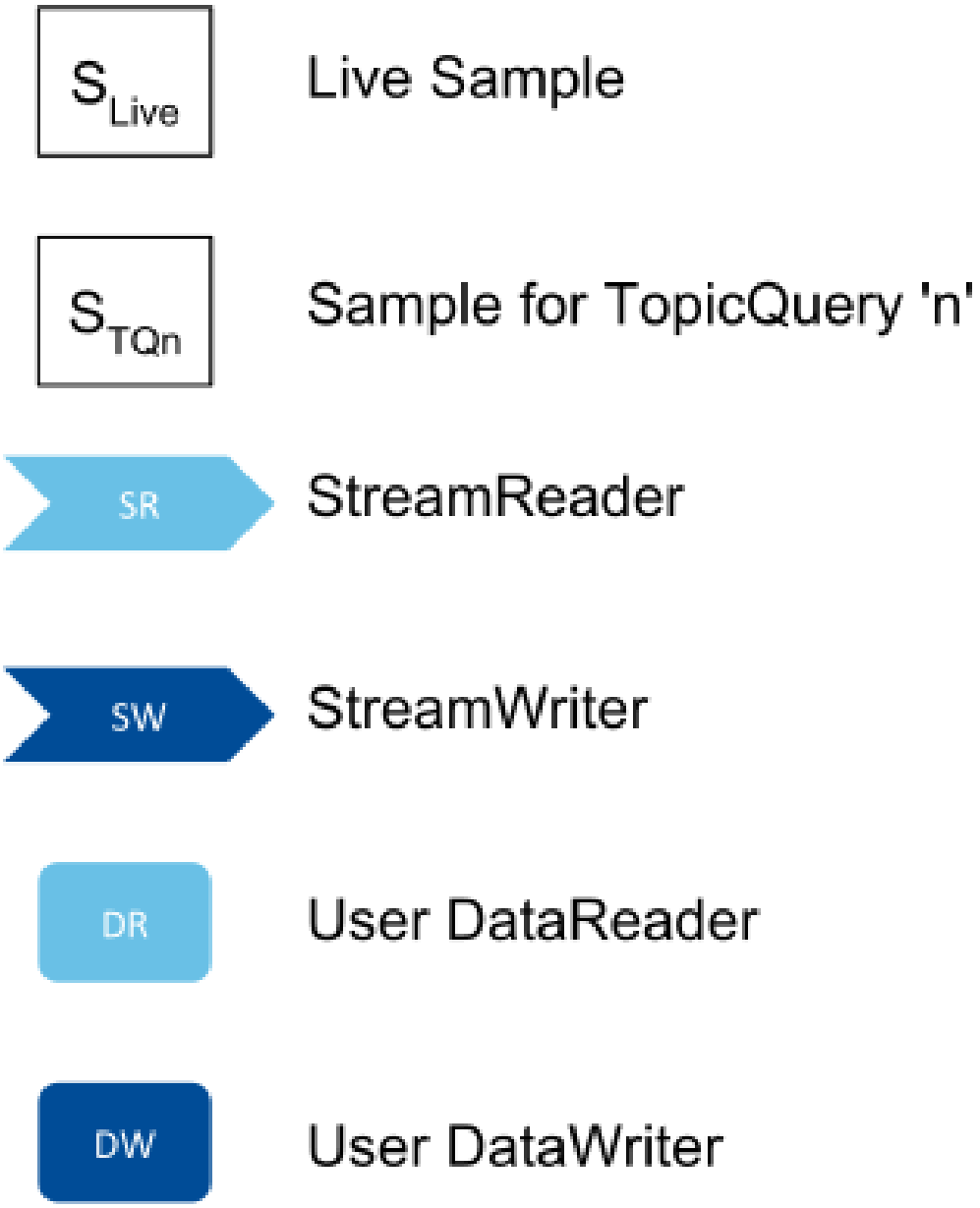


Figure 9.1: Symbol Legend for Proxy Modes Figures

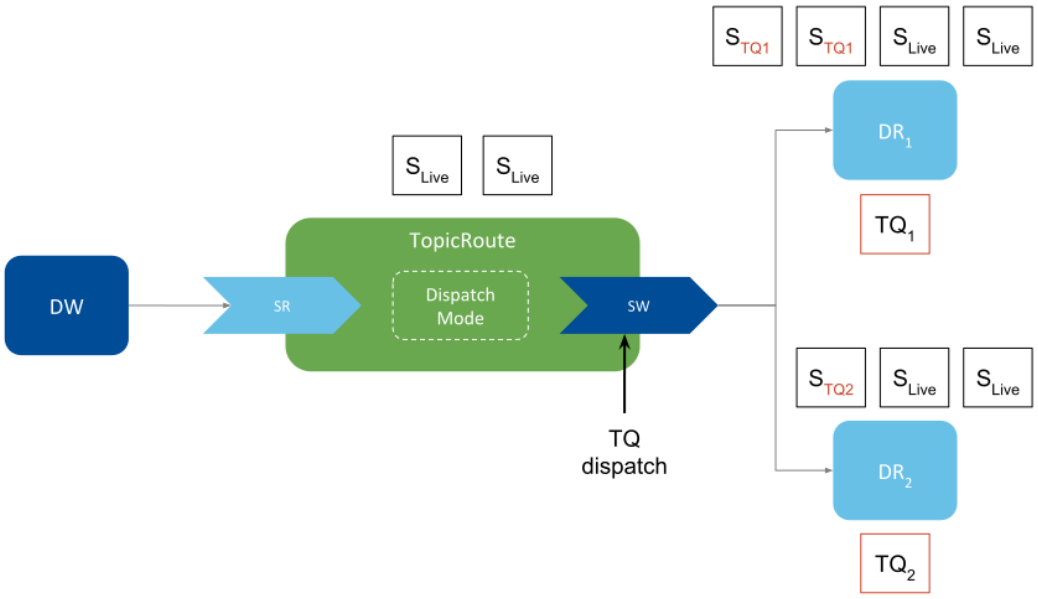


Figure 9.2: *TopicRoute* Enabling *TopicQuery* Proxy in Dispatch Mode

capacity. In this case, a *Routing Service* instance connected to the application can cache a set of data published by the user *DataWriter* and dispatch the *TopicQueries* issued by user *DataReaders*.

To enable *TopicQuery* proxy dispatch mode, use the following configuration tags within a *TopicRoute*/*AutoTopicRoute* configuration:

```
<topic_query_proxy>
    <mode>DISPATCH</mode>
</topic_query_proxy>
```

The above configuration will cause the Durability QoS setting for the *TopicRoute*'s output *DataWriter* to be TRANSIENT_LOCAL and will enable *TopicQuery* dispatch. If you want to configure advanced dispatch features, you can set other options in the TopicQueryDispatchQosPolicy within the corresponding *DataWriter* QoS tag.

9.2 Propagation Mode

Propagation mode refers to having *Routing Service* act as a proxy of *TopicQueries*. The *TopicRoutes* propagate the *TopicQueries* issued by the matching user *DataReaders* to the matching user *DataWriters*. Then the samples generated for both the *TopicQuery* and live stream are 'propagated' to the original user *DataReaders*. Figure 9.3 shows a simple scenario.

The *TopicRoute* propagates the *TopicQuery* requests from user *DataReaders* on the subscription side to the user *DataWriters* on the publication side. User *DataWriters* eventually dispatch the *TopicQuery* requests and generate samples for the *TopicQuery* stream. The samples for a specific *TopicQuery* are routed to the corresponding original user *DataReader* that issued such *TopicQuery*.

For a given *TopicRoute*, the propagation of *TopicQuery* requests and samples for both the *TopicQuery* and live stream occurs sequentially. The expected traffic pattern consists of *TopicQuery*

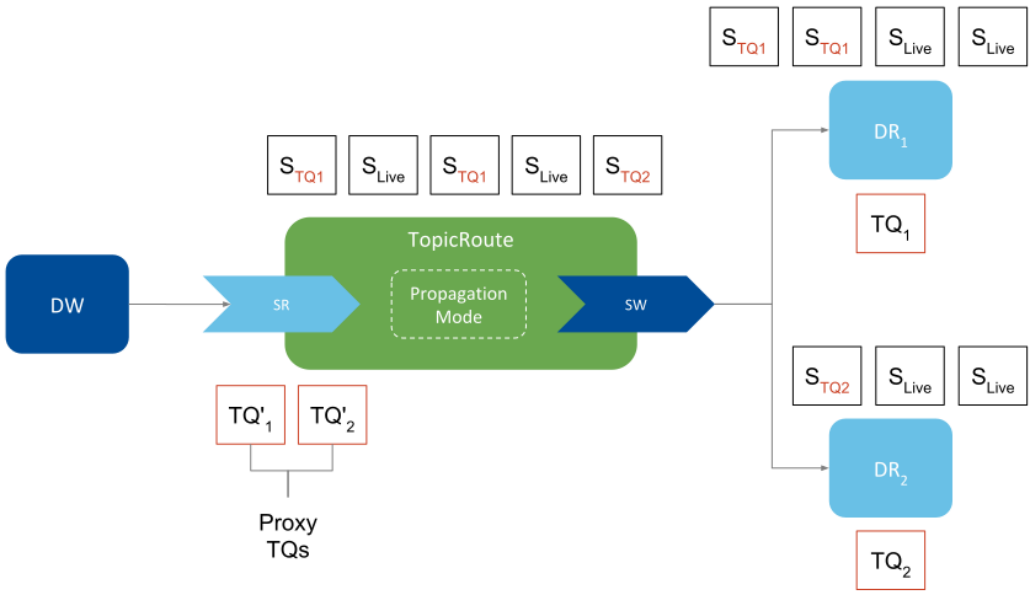


Figure 9.3: *TopicRoute* Enabling *TopicQuery* Proxy in Propagation Mode

requests, *TopicQuery* samples, and live samples interleaved.

TopicQuery propagation is also compatible with filter propagation (see Section 8). You can enable both at the same time and expect live samples to be filtered accordingly, and *TopicQuery* samples to be unaffected by the filters.

To enable *TopicQuery* proxy dispatch mode, you can use the following configuration tags within a *TopicRoute*/*AutoTopicRoute* configuration:

```
<topic_query_proxy>
    <mode>PROPAGATION</mode>
</topic_query_proxy>
```

Note that the above configuration will cause the *TopicRoute*'s output *DataWriter* durability QoS setting to be VOLATILE.

9.3 Restrictions

TopicQuery proxy in PROPAGATION mode cannot be enabled when:

- Using *Routes* or *AutoRoutes*, since they are meant to work with other adapters different than the builtin DDS one.
- A transformation is present in the *TopicRoute*'s output.
- The *TopicRoute* has a custom processor.

Chapter 10

Traversing Wide Area Networks

Many systems today already rely on *Connext DDS* to distribute their information across a Local Area Network (LAN). However, more and more of these systems are being integrated in Wide Area Networks (WANs). With *Routing Service*, you can scale *Connext DDS* real-time publish/subscribe data-distribution beyond the current local networks and make it available throughout a WAN.

Out of the box, *Routing Service* only uses UDPv4 and Shared Memory transports to communicate with other *Routing Services* and *Connext DDS* applications. This configuration is appropriate for systems running within a single LAN. However, using UDPv4 introduces several problems when trying to communicate with *Connext DDS* applications running in different LANs:

- UDPv4 traffic is usually filtered out by the LAN firewalls for security reasons.
- Forwarded ports are usually TCP ports.
- Each LAN may run in its own private IP address space and use NAT (Network Address Translation) to communicate with other networks.

To overcome these issues, *Routing Service* is distributed with a TCP transport that is NAT friendly. The transport can be configured via XML using the PropertyQosPolicy of the *Routing Services* participants.

Figure 10.1 shows a typical scenario where two *Routing Services* are used to bridge two *Connext DDS* applications running in two different LANs.

10.1 TCP Configuration elements

The TCP transport distributed with *Routing Service* can be used to address multiple communication scenarios that range from simple communication within a single LAN to complex communication scenarios across LANs where NATs and firewalls may be involved.

10.1.1 TCP Transport Initial Peers

With the TCP transport, the addresses of the initial peers (NDDS_DISCOVERY_PEERS) that will be contacted during the discovery process have the following format:

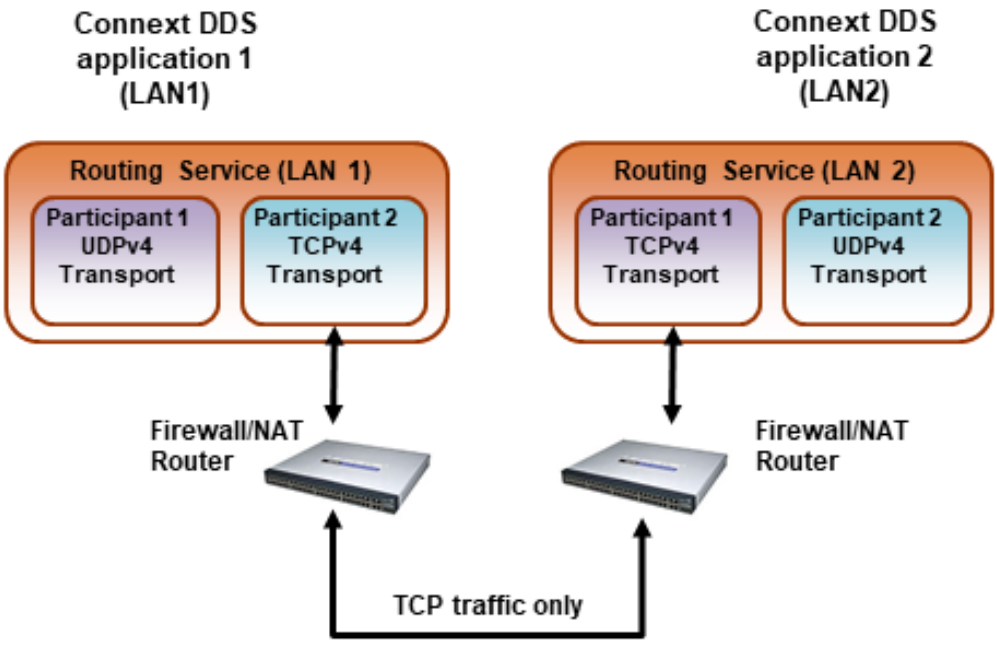


Figure 10.1: WAN Communication Using TCP Transport

```
For WAN communication: tcpv4_wan://<IP address or hostname>:<port>
For LAN communication: tcpv4_lan://<IP address or hostname>:<port>
For WAN+TLS communication: tlsv4_wan://<IP address or hostname>:port
For LAN+TLS communication: tlsv4_lan://<IP address or hostname>:port
```

Example: Setting discovery peers for TCP wan/lan

```
setenv NDDS_DISCOVERY_PEERS tcpv4_wan://10.10.1.165:7400,tcpv4_wan://10.10.1.111:7400,
↪tcpv4_lan://192.168.1.1:7500
```

When the TCP transport is configured for LAN communication (with the **parent.classid** property), the IP address is the LAN address of the peer and the port is the server port used by the transport (the **server_bind_port** property).

When the TCP transport is configured for WAN communication (with the **parent.classid** property), the IP address is the WAN or public address of the peer and the port is the public port that is used to forward traffic to the server port in the TCP transport.

When TLS is enabled, the transport settings are similar to WAN and LAN over TCP. See Figure 10.2.

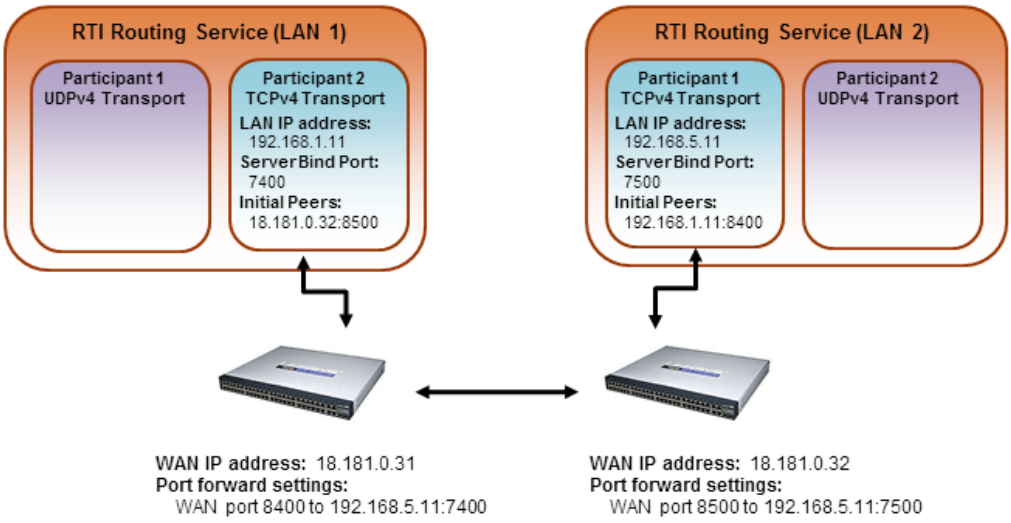


Figure 10.2: Initial Peers in WAN Communication

10.1.2 TCP Transport Property

You can configure the TCP transport in *Routing Service* in the same manner as a *Connext DDS* application. Transports are configured at the *DomainParticipant* level by means of the PropertyQosPolicy. For details, see transport plugins in *RTI Connext DDS Users Manual*. For a list of available transport properties for TCP, see RTI TCP Transport configuration.

10.2 Support for External Hardware Load Balancers in TCP Transport Plugin

For two *Connext DDS* applications to communicate, the TCP Transport Plugin needs to establish 4-6 connections between the two communicating applications. The plugin uses these connections to exchange DDS data (discovery or user data) and TCP Transport Plugin control messages.

With the default configuration, the TCP Transport Plugin does not support external load balancers. This is because external load balancers do not forward the traffic to a unique TCP Transport Plugin server, but they divide the connections among multiple servers. Because of this behavior, when an application running a TCP Transport Plugin client tries to establish all the connections to an application running a TCP Transport Plugin server, the server may not receive all the required connections.

In order to support external load balancers, the TCP Transport Plugin provides a session-ID negotiation feature. When session-ID negotiation is enabled (by setting the **negotiate_session_id** property to true), the TCP Transport Plugin will perform the negotiation depicted in Figure 10.3.

During the session-ID negotiation, the TCP Transport Plugin exchanges three types of messages:

- **Session-ID Request:** This message is sent from the client to the server. The server must respond with a session-ID response.

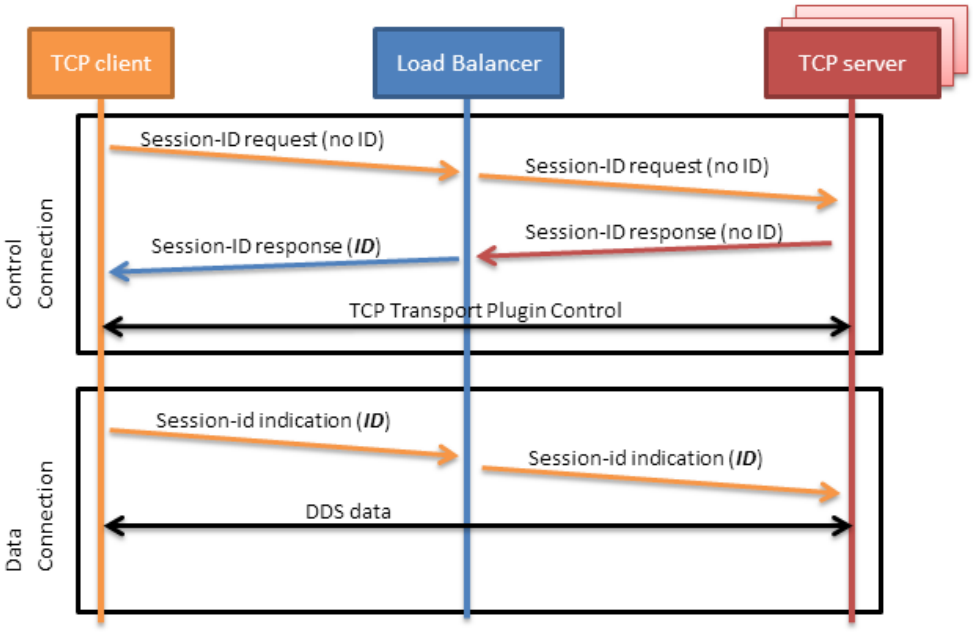


Figure 10.3: Session-ID Negotiation

- **Session-ID Response:** This message is sent from the server to the client as a response to a session-ID request. The client will store the session ID contained in this message.
- **Session-ID Indication:** This message is sent from the client to the server; it does not require a response from the server.

The negotiation consists of the following steps:

1. The TCP client sends a session-ID request with the session ID set to zero.
2. The TCP server sends back a session-ID response with the session ID set to zero.
3. The external load balancer modifies the session-ID response, setting the session ID with a value that is meaningful to the load balancer and identifies the session.
4. The TCP client receives the session-ID response and stores the received session ID.
5. For each new connection, the TCP client sends a session-ID indication containing the stored session ID. This will allow the load balancer to redirect to the same server all the connections with the same session ID.

Figure 10.4 depicts the TCP payload of a session-ID message. The payload consists of 48 bytes. In particular, your load balancer needs to read/modify the following two fields:

- CTRLTYPE: This field allows a load balancer to identify session-ID messages. Its value (two bytes) varies according to the session-ID message type: 0x0c05 for a request, 0x0d05 for a response, or 0x0c15 for an indication.
- SESSION-ID: This field consists of 16 bytes that the load balancer can freely modify according

to its requirements.

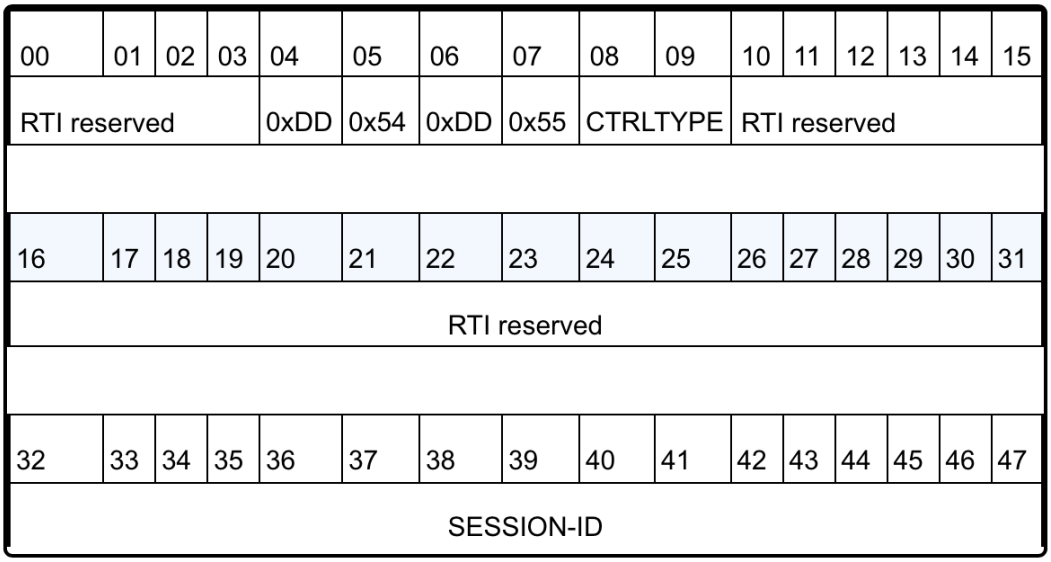


Figure 10.4: TCP Payload for Session-ID Message

To ensure all the TCP connections within the same session are directed to the same server, you must configure your load balancer to perform the two following actions:

1. Modify the SESSION-ID field in the *session-id response* with a value that identifies the session within the load balancer.
2. Make the load-balancing decision according to the value of the SESSION-ID field in the session-ID indication.

Chapter 11

Tutorials

This chapter describes several examples, all of which use *RTI Shapes Demo* to publish and subscribe to topics which are colored moving shapes.

In each example, you can start all the applications on the same computer or on different computers in your network. If you don't have *Shapes Demo* installed already, you should download and install it from RTI's Downloads page or the RTI Support Portal (the latter requires an account name and password). If you are not already familiar with how to start *Shapes Demo* and change its domain ID, please see the *Shapes Demo User's Manual* for details.

Important Notes:

- Please review Section 1.2 to understand where to find the examples (referred to as <path to examples>).
- The following instructions include commands that you will enter in a command shell. These instructions use forward slashes in directory paths, such as bin/rtiroutingservice. If you are using a Windows platform, replace all forward slashes in such paths with backwards slashes, such as bin\rtiroutingservice.
- If you run *Shapes Demo* and *Routing Service* on different machines and these machines do not communicate over multicast, you will have to set the environment variable NDDS_DISCOVERY_PEERS to enable communication. For example, assume that you run *Routing Service* on Host 1 and *Shapes Demo* on Host 2 and Host 3. In this case, the environment variable would be set as follows:

  Host 1:

  ```
  set NDDS_DISCOVERY_PEERS=<host2>,<host3> (on Windows systems)
  export NDDS_DISCOVERY_PEERS=<host2>,<host3> (on UNIX-based systems)
  ```

  Host 2:

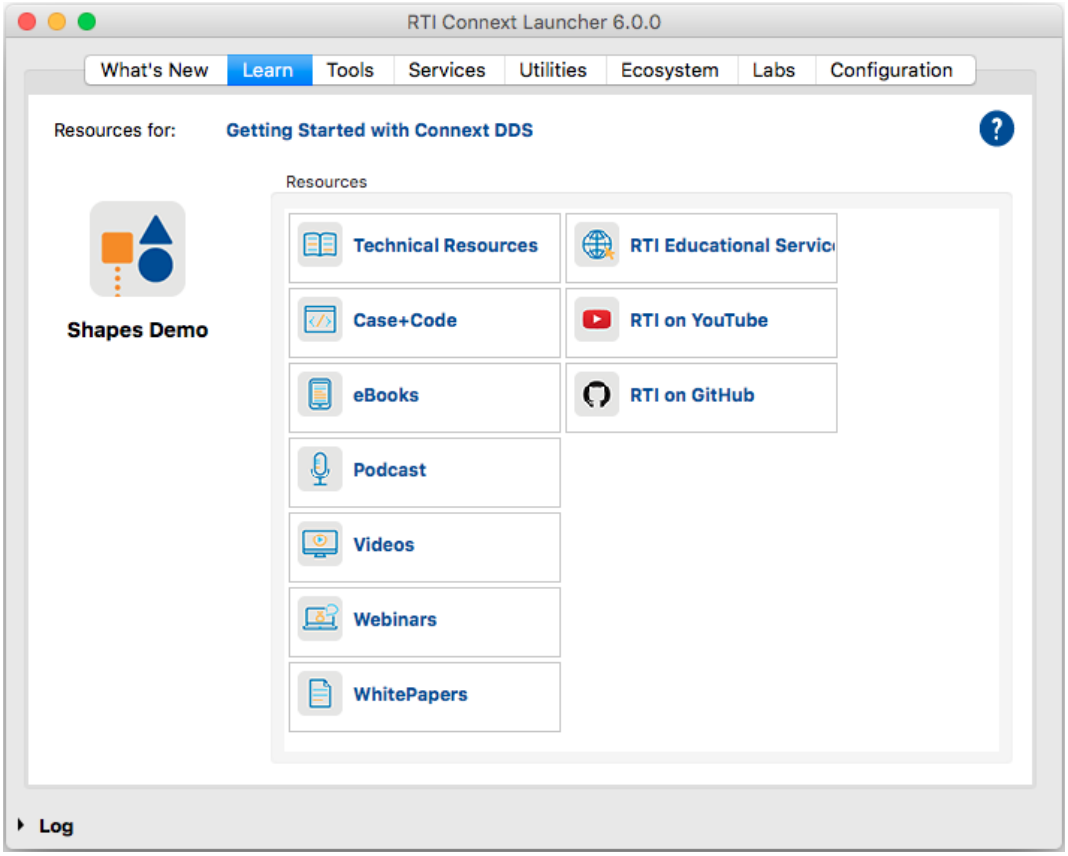
  ```
  set NDDS_DISCOVERY_PEERS=<host1>
  ```

  Host 3:

```
export NDDS_DISCOVERY_PEERS=<host1>
```

11.1 Starting Shapes Demo

You can start *Shapes Demo* from the Learn tab in *RTI Launcher*.



Or from a command shell:

```
<NDDSHOME>/bin/rtishapesdemo
```

NDDSHOME is described in Section 1.2.

11.2 Example: Routing All Data from One Domain to Another

This example uses the default configuration file[1] for *Routing Service*, which routes all data published on domain 0 to subscribers on domain 1.

1. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.
2. Start a second copy of *Shapes Demo*. We'll call this the Subscribing Demo. Then:
   - Open its Configuration dialog (under Controls).

[1] <NDDSHOME>/resource/xml/RTI_ROUTING_SERVICE.xml

- Press **Stop**.
- Change the domain ID to 1.
- Press **Start**.

3. In the Publishing Demo, publish some Squares, Circles, and Triangles.
4. In the Subscribing Demo, subscribe to Squares, Circles and Triangles.

   Notice that the Subscribing Demo does not receive any shapes. Since we haven't started *Routing Service* yet, data from domain 0 isn't routed to domain 1.
5. Start *Routing Service* by entering the following in a command shell:

   ```
   cd <NDDSHOME>
   bin/rtiroutingservice -cfgName default
   ```

   Now you should see all the shapes in the Subscribing Demo.
6. Stop *Routing Service* by pressing **Ctrl-c**.

   You should see that the Subscribing Demo stops receiving shapes.

Additionally, you can start *Routing Service* (Step 5) with the following parameters:

- **-verbosity 3**, to see messages from *Routing Service* including events that have triggered the creation of routes.
- **-domainIdBase** *X*, to use domains *X* and *X+1* instead of 0 and 1 (in this case, you need to change the domain IDs used by *Shapes Demo* accordingly). This option adds X to the domain IDs in the configuration file.

Note: **-domainIdBase** only affects the domain IDs of DomainRoute participants; it does not affect the domain IDs of participants used for monitoring or administration.

11.3 Example: Changing Data to a Different Topic of Same Type

In this example, *Routing Service* receives samples of topic Square and republishes them as samples of topic Circle.

1. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.
2. Start a second copy of *Shapes Demo*. We'll call this the Subscribing Demo. Then:
   - Open its Configuration dialog (under Controls).
   - Press **Stop**.
   - Change the domain ID to 1.
   - Press **Start**.
3. In the Publishing Demo, publish some Squares, Circles, and Triangles.
4. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
-cfgFile <path to examples>/routing_service/shapes/topic_bridge.xml \
-cfgName example
```

5. In the Subscribing Demo, subscribe to Squares, Circles and Triangles.

   Notice that the Subscribing Demo does not receive any shapes. Since we haven't started *Routing Service* yet, data from domain 0 isn't routed to domain 1.

6. Stop *Routing Service* by pressing **Ctrl-c**.

7. Try writing your own topic route that republishes triangles on domain 0 to circles on domain 1. Create some Triangle publishers and a Circle subscriber in the respective *Shapes Demo* windows.

11.4 Example: Changing Some Values in Data

So far, we have learned how to route samples from one topic to another topic of the same data type. Now we will see how to change the value of some fields in the samples and republish them.

1. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.

2. Start a second copy of *Shapes Demo*. We'll call this the Subscribing Demo. Then:

   - Open its Configuration dialog (under Controls).
   - Press **Stop**.
   - Change the domain ID to 1.
   - Press **Start**.

3. In the Publishing Demo, publish some Squares, Circles, and Triangles.

4. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
-cfgFile <path to examples>/routing_service/shapes/topic_bridge_w_transf1.
↪xml \
-cfgName example
```

5. In the Subscribing Demo, subscribe to Squares, Circles and Triangles.

   Notice that the (x,y) coordinates of the shapes are inverted form what appears in the Publishing Demo.

6. Stop *Routing Service* by pressing **Ctrl-c**.

7. Try changing the transformation to assign the output **shapesize** to the input **x**.

11.5 Example: Transforming the Data's Type and Topic with an Assignment Transformation

This example shows how to transform the data topic and type. We will use *rtiddsspy* to verify the result. *rtiddsspy* is a utility provided with *Connext DDS*; it monitors publications on any DDS domain.

1. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.
2. In the Publishing Demo, publish some Squares, Circles, and Triangles.
3. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
-cfgFile <path to examples>/routing_service/shapes/topic_bridge_w_transf2.
↪xml \
-cfgName example
```

4. We will use the *rtiddsspy* utility to verify the transformation of the data topic and type. Run these commands:

```
cd <NDDSHOME>
bin/rtiddsspy -domainId 0 -printSample
bin/rtiddsspy -domainId 1 -printSample
```

   You will notice that the publishing samples received by *rtiddsspy* for domain 0 are of type ShapeType and topic Square. The subscribing samples received by *rtiddsspy* for domain 1 are of type Point and topic Position. Notice that the two data structures are different.

5. Stop *Routing Service* by pressing **Ctrl-c**.

11.6 Example: Transforming the Data with a Custom Transformation

Now we will use our own transformation between shapes. *Routing Service* allows you to install plug-ins that implement the Transformation API to create custom transformations. To build a custom transformation, you must have the *Connext DDS* libraries installed.

Note: This example assumes your working directory is <path to examples>/routing_service/shapes/transformation/[make or windows]. If your working directory is different, you will need to modify the configuration topic_bridge_w_custom_transf.xml to update the paths.

---

1. Compile the transformation in <path to examples>/routing_service/shapes/transformation/[make or windows]:
   - On UNIX-based systems:
     - Set the environment variable NDDSHOME (see Section 1.2). For details on how to set it, see the *RTI Connext DDS Core Libraries Getting Started Guide*.

  - **Enter:**

```
cd <path to examples>/routing_service/shapes/transformation/make
gmake -f Makefile.<architecture>
```

- On Windows systems:
  - Set the environment variable NDDSHOME (see Section 1.2). For details on how to set it, see the *RTI Connext DDS Core Libraries Getting Started Guide*.
  - Open the Visual Studio solution under <path to examples>\routing_service\shapes\transformation\windows.
  - Select the Release DLL build mode and build the solution.

2. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.
3. In the Publishing Demo, publish some Squares.
4. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
-cfgFile <path to examples>/routing_service/shapes/topic_bridge_w_custom_
↪transf.xml \
-cfgName example
```

5. Start a second copy of *Shapes Demo*. We'll call this the Subscribing Demo. Then:
   - Open its Configuration dialog (under Controls).
   - Press **Stop**.
   - Change the domain ID to 1.
   - Press **Start**.
6. In the Subscribing Demo, subscribe to Squares.

   Notice that squares on domain 1 have only two possible values for **x**.
7. Stop *Routing Service* by pressing **Ctrl-c**.
8. Change the fixed 'x' values for the Squares in the configuration file and restart *Routing Service*.
9. Stop *Routing Service* by pressing **Ctrl-c**.
10. Edit the source code (in shapestransf.c) to make the transformation multiply the value of the field by the given integer constant instead of assigning the constant.

Hint: Look for the function **ShapesTransformationPlugin_createOutputSample()**, called from **ShapesTransformation_transform()** and use **DDS_DynamicData_get_long()** before **DDS_DynamicData_set_long()**.

---

11. Recompile the transformation (the new shared library will be copied automatically) and run *Routing Service* as before.

11.7 Example: Using Remote Administration

In this example, we will configure *Routing Service* remotely. We won't see data being routed until we remotely enable an *AutoTopicRoute* after the application is started. Then we will change a QoS value and see that it takes effect on the fly.

1. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.
2. In the Publishing Demo, publish some Squares, Circles and Triangles.
3. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
-cfgFile <path to examples>/routing_service/shapes/administration.xml \
-cfgName example -appName MyRoutingService
```

4. Start a second copy of *Shapes Demo*. We'll call this the Subscribing Demo. Then:
   - Open its Configuration dialog (under Controls).
   - Press **Stop**.
   - Change the domain ID to 1.
   - Press **Start**.
5. In the Subscribing Demo, subscribe to Squares, Circles and Triangles.

   Notice that no data is routed to domain 1.
6. On a different or the same machine, start the *Routing Service* remote shell:

```
cd <NDDSHOME>
bin/rtirssh -domainId 0
```

Note: We use domain 0 in the shell because *Routing Service* is configured in administration.xml to receive remote commands on that domain. You could have started *Routing Service* with the **-remoteAdministrationDomainId** command-line option and then used domain **X** for the shell.

---

7. In the shell, enter the following command:

```
enable MyRoutingService RemoteConfigExample::Session::Shapes
```

   Notice that the shapes are now received on domain 1. The above command consists of two parts: the name of the *Routing Service*, which you gave when you launched the application with the option **-appName**, and the name of the entity you wanted

to enable. That name is formed by appending its parent entities' names starting from the domain route as defined in the configuration file administration.xml.

You could have run *Routing Service* without **-appName**. Then the name would be the one provided with **-cfgName** ("example"). You could also have used **-identifyExecution** to generate the name based on the host and application ID. In this case, you would have used this automatic name in the shell.

8. Examine the file **<path to examples>/routing_service/shapes/time_filter_qos.xml** on the *Routing Service* machine. It contains an XML snippet that defines a QoS value for an auto topic route's DataReader. Execute the following command in the shell:

```
update MyRoutingService RemoteConfigExample::Session::Shapes \
        <path to examples>/routing_service/shapes/time_filter_qos.xml
```

Notice that the receiving application only gets shapes every 2 seconds. The *AutoTopicRoute* has been configured to read (and forward) samples with a minimum separation of 2 seconds.

Routing Service can be configured remotely using files located on the remote machine or the shell machine. In the next step you will edit the configuration files on both machines. Then you will see how to specify which of the two configuration files you want to use.

Note: If you are running the shell and *Routing Service* on the same machine, skip steps 9 and 10.

---

9. Edit the XML configuration files on both machines:
   - In **<path to examples>/routing_service/shapes/time_filter_qos.xml** on the service machine, change the minimum separation to 0 seconds.
   - In **<path to examples>/routing_service/shapes/time_filter_qos.xml** on the shell machine, change the minimum separation to 5 seconds.
10. Run the following commands in the shell:
    - Enter the following command. Notice the use of **remote** at the end—this means you want to use the XML file on the service machine (the remote machine, which is the default if nothing is specified).

```
update MyRoutingService RemoteConfigExample::Session::Shapes \
    <path to examples>/routing_service/shapes/time_filter_qos.xml remote
```

Note: The path to the XML file in this example is relative to the working directory from which you run *Routing Service*.

Since no time filter applies, the shapes are received as they are published.

- Enter the following command. This time use **local** at the end—this means you want to use the XML file on the shell machine (the local machine).

```
update MyRoutingService RemoteConfigExample::Session::Shapes \
    <path to examples>/routing_service/shapes/time_filter_qos.xml local
```

Note: The path to the XML file in this example is relative to the working directory from which you run the *Routing Service* shell.

You will see that now the shapes are only received every 5 seconds.

- Enter the following command. Once again, we use *remote* at the end to switch back to the XML file on the remote machine.

```
update MyRoutingService RemoteConfigExample::Session::Shapes \
    <path to examples>/routing_service/shapes/time_filter_qos.xml remote
```

Shapes are once again received as they are published

11. Disable the *AutoTopicRoute* again by entering:

```
disable MyRoutingService RemoteConfigExample::Session::Shapes
```

The shapes are no longer received on Domain 1.

Note: At this point, you could still update the *AutoTopicRoute*'s configuration. You could also change immutable QoS values, since the *DataWriter* and *DataReader* haven't been created yet. These changes would take effect the next time you called enable.

---

12. Run these commands in the shell and see what happens after each one:

```
enable MyRoutingService RemoteConfigExample::Session::SquaresToCircles
disable MyRoutingService RemoteConfigExample::Session::SquaresToCircles
enable MyRoutingService RemoteConfigExample::Session::SquaresToTriangles
```

These commands change the output topic that is published after receiving the input Square topic. As you can see, you can use the shell to switch *TopicRoutes* after *Routing Service* has been started.

13. Perform a remote shutdown of the service. Run the following command:

```
shutdown MyRoutingService
```

You should receive a reply indicating that the shutdown sequence has been initiated. Verify in the terminal in which *Routing Service* was running that the process is exiting or has already exited.

14. Stop the shell by running this command in the shell:

```
exit
```

11.8 Example: Monitoring

You can publish status information with *Routing Service*. The monitoring configuration is quite flexible and allows you to select the entities that you want to monitor and how often they should publish their status.

1. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.
2. In the Publishing Demo, publish two Squares, two Circles and two Triangles.
3. Start a second copy of *Shapes Demo*. We'll call this the Subscribing Demo. Then:
   - Open its Configuration dialog (under Controls).
   - Press **Stop**.
   - Change the domain ID to 1.
   - Press **Start**.
4. In the Subscribing Demo, subscribe to Squares, Circles and Triangles.

   At this point you will not see any shapes moving in the Subscribing Demo. It isn't receiving shapes from the Publishing Demo because they use different domain IDs.
5. On a different or the same machine, start the *Routing Service* remote shell:

   ```
   cd <NDDSHOME>
   bin/rtirssh -domainId 0
   ```

   Note: We use domain 0 in the shell because *Routing Service* is configured in administration.xml to receive remote commands on that domain. You could have started *Routing Service* with the **-remoteAdministrationDomainId** command-line option and then used domain **X** for the shell.
6. Start *Routing Service* by entering the following in a command shell:

   ```
   cd <NDDSHOME>
   bin/rtiroutingservice
   -cfgFile <path to examples>/routing_service/shapes/monitoring.xml \
   -cfgName example -appName MyRoutingService
   ```

   This configuration file routes Squares and Circles using two different *TopicRoutes*.
7. Now you can subscribe to the monitoring topics (see Section 6). You can do it in your own application, or by using *RTI Admin Console* or *rtiddsspy*. Enter the following in a terminal:

   ```
   cd <NDDSHOME>
   bin/rtiddsspy -domainId 2 -printSample
   ```

> **Note:** We use domain 2 because *Routing Service* is configured in **monitoring.xml** to publish status information on that domain. You could have started *Routing Service* with the **-remoteMonitoringDomainId X** command-line option and then used domain **X** for *rtiddsspy*.

8. Depending on the publication period of the entity in the XML file we used, you will receive status samples at different rates. In the output from *rtiddsspy*, check the statistics about the two topic routes we are using.

   We will focus on the input samples per second. The number of samples per second in our case is 32. That value depends on the publication rate of *Shapes Demo* configurable with the option **-pubInterval <milliseconds between writes>**.

9. Create two additional Square publishers in the Publishing Demo (domain 0).

10. Check *rtiddsspy* again for new status information.

    In the *TopicRoute* for Squares, we are receiving double the amount of data.

11. Look at the status of the *DataReader* in the output from *rtiddsspy*.

    It contains an aggregation of the two contained *TopicRoutes*, giving us a mean of nearly 48 samples per second.

12. We can update the monitoring configuration at run time using the remote administration feature. In the configuration file, we enabled remote administration on domain 0.

    On a different or the same machine, start the *Routing Service* remote shell:

```
cd <NDDSHOME>
bin/rtirssh -domainId 0
```

13. We are receiving the status of the *TopicRoute* Circles every five seconds. To receive it more often, use the following command:

```
update MyRoutingService DomainRoute::Session::Circles \
        topic_route.entity_monitoring.status_publication_period.sec=2
```

14. In some cases, you might want to know only about one specific *TopicRoute*. If you only want to know about the topic route Circles but not Squares, you can disable monitoring for Squares:

```
update MyRoutingService DomainRoute::Session::Squares \
    topic_route.entity_monitoring.enabled=false
```

15. To enable it again, enter:

```
update MyRoutingService DomainRoute::Session::Squares \
    topic_route.entity_monitoring.enabled=true
```

16. If you are no longer interested in monitoring this service, you can completely disable it with the following command:

```
update MyRoutingService routing_service.monitoring.enabled=false
```

Now you won't receive any more status samples.

17. You can enable it again any time by entering:

```
update MyRoutingService routing_service.monitoring.enabled=true
```

18. Stop *rtiddsspy* by pressing **Ctrl-c**.

19. Stop the shell:

```
exit
```

20. Stop Routing Service by pressing **Ctrl-c**.

11.9 Example: Using the TCP Transport

This example shows how to use *Routing Service* to bridge data between different LANs over TCP. *Routing Service* will act as the gateway in a LAN with which other *Connext DDS* applications can communicate to send or receive data. Section 10 has more information about scenarios and detailed configuration parameters.

You will run two copies of *Routing Service*. One copy will run on a machine that is behind a firewall/network router with a public IP (First Peer); the other will run on a machine in another LAN (Second Peer).

- On the First Peer (behind a firewall/router with a public IP):

  1. In the First Peer's network, configure the firewall to forward the TCP ports used by *Routing Service*.

     In this example, we will use port 7400. You do not need to configure your firewall for every single *Connext DDS* application in your LAN; doing it just once for *Routing Service* will allow other applications to communicate through the firewall.

  2. Include the Second Peer's public IP address and port in the `NDDS_DISCOVERY_PEERS` environment variable.

     For example, on a UNIX-based system:

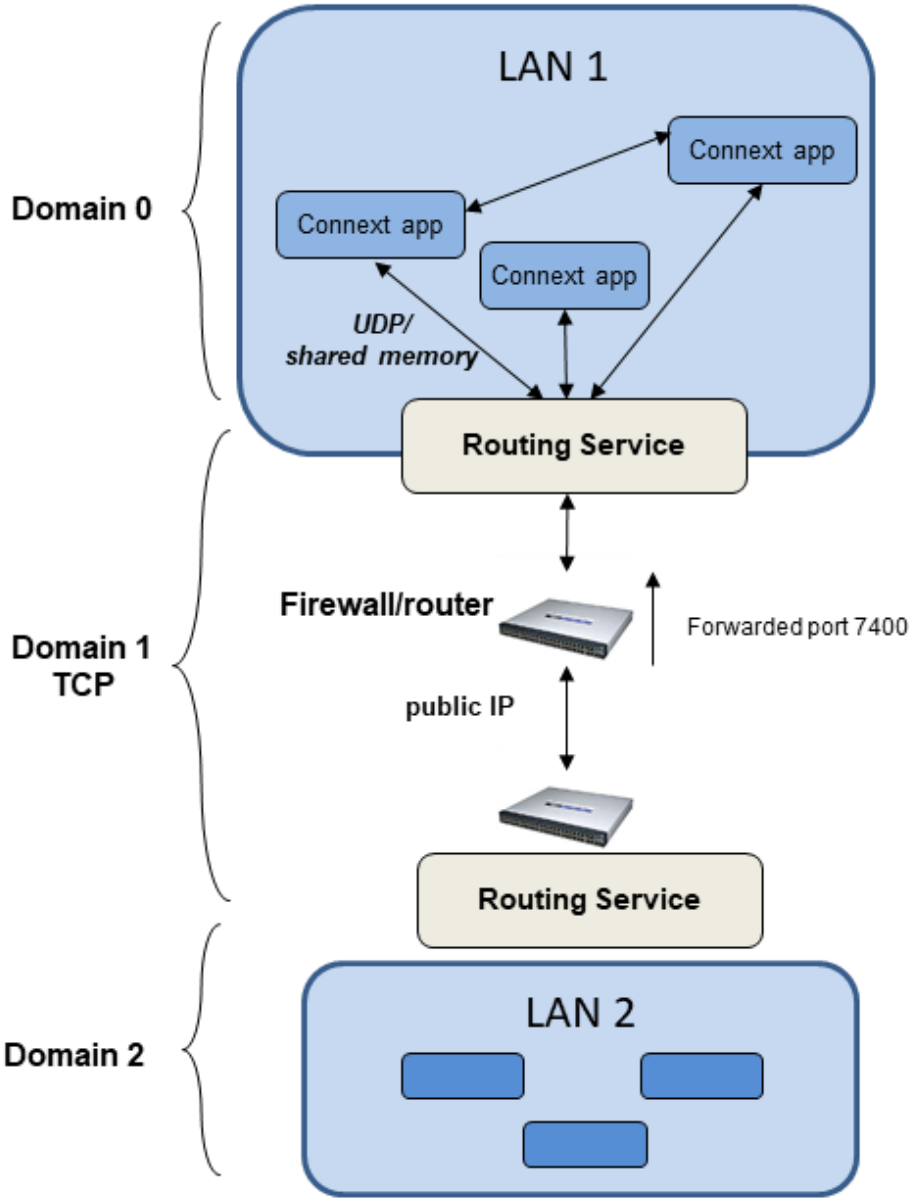
```
export NDDS_DISCOVERY_PEERS=tcpv4_wan://<server's public IP address>:
↪<port>
```

     On a Windows system:

```
set NDDS_DISCOVERY_PEERS=tcpv4_wan://<server's public IP address>:
↪<port>
```

     When you configure `NDDS_DISCOVERY_PEERS`, make sure to use a transport class prefix (tcpv4_wan, udpv4, shmem) for each entry. See discovery peer configuration for details.

  3. Set the public IP address and port in the configuration file:


LAN 1
Connext app
Connext app
Connext app
UDP/
shared memory
Routing Service
Domain 0
Firewall/router
Forwarded port 7400
public IP
Domain 1
TCP
Routing Service
LAN 2
Domain 2

  - Open the file **<path to examples>/routing_service/shapes/tcp_transport.xml.**
  - The file contains several configurations. Find the configuration with name "TCP_1". Then find the "public_address" property (<name>dds.transport.TCPv4.tcp1.public_address</name>) within that configuration.
  - Set the local public IP address and port. For example, to set the address to 10.10.1.150 and port 7400:

```
<element>
    <name>dds.transport.TCPv4.tcp1.public_address</name>
    <value>10.10.1.150:7400</value>
</element>
```

  - Save and close the file.

4. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
    -cfgFile <path to examples>/routing_service/shapes/tcp_transport.xml \
    -cfgName TCP_1
```

5. On any computer in this LAN, start *Shapes Demo* and publish some shapes on domain 0.

- On the Second Peer (a machine in any other LAN):

1. Include the First Peer's public IP address and port in the `NDDS_DISCOVERY_PEERS` environment variable the same way you did before.
2. Set the public IP address and port in the configuration file:
   - Open the file **<path to examples>/routing_service/shapes/tcp_transport.xml.**
   - The file contains several configurations. Find the configuration with name "TCP_2". Then find the "public_address" property (<name>dds.transport.TCPv4.tcp1.public_address</name>) within that configuration.
   - Set the local public IP address and port. For example, to set the address to 10.10.1.10 and port 7400:

```
<element>
    <name>dds.transport.TCPv4.tcp1.public_address</name>
    <value>10.10.1.10:7400</value>
</element>
```

   - Save and close the file.
3. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
  -cfgFile <path to examples>/routing_service/shapes/tcp_transport.xml \
  -cfgName TCP_2
```

4. On any computer in this LAN, start *Shapes Demo* and create subscribers on domain 2. Do not use an already running instance of *Shapes Demo*—you need a new one that uses a different domain ID. You should receive what is being published in the server's LAN.

- **Notes:**
  - **Running Shapes Demo on a Different Computer**

    If the computer running *Shapes Demo* is different than the computer running the client *Routing Service*, add the address of the client (IP address or host name) to the *Shapes Demo* discovery peers before starting *Shapes Demo*. To do so, set the `NDDS_DISCOVERY_PEERS` environment variable.
  - **Using Two Computers in the Same LAN**

    If both machines are in the same LAN, run both *Routing Service* with the configuration file **tcp_transport_lan.xml** and use **tcpv4_lan://** as the peer prefix in the environment variable `NDDS_DISCOVERY_PEERS`. At least one of the peer descriptors must contain the port number.

    For example, suppose the first peer is 192.168.1.3, the second peer is 192.168.1.4, and you want to use port 7400. On the first peer set `NDDS_DISCOVERY_PEERS` to **tcpv4_lan://192.168.1.4:7400** and on the second peer set it to **tcpv4_lan://192.168.1.3:7400**. You don't need to specify an IP address in the configuration file.
  - **Running the Example on One Computer**

    To run the example on the same machine, open the file **<path to examples>/routing_service/ shapes/tcp_transport_lan.xml** and change the property **dds.transport.TCPv4.tcp1.server_bind_port** within **TCP_1** to 7401. Run both *Routing Service* with the modified **tcp_transport_lan.xml** configuration file and use **tcpv4_lan://** as the peer prefix in the environment variable `NDDS_DISCOVERY_PEERS`. You will also need to specify port 7401 in the **tcpv4_lan** peer in the `NDDS_DISCOVERY_PEERS` environment variable of the *Routing Service* in the Second Peer to reflect this port change in the configuration file.
  - **Using a Secure Connection over WAN**

    To run the example using a secure connection between the two *Routing Service* instances, use the configuration file **tcp_transport_tls.xml**. You will also need to set the peer prefix to **tlsv4_wan://** in the `NDDS_DISCOVERY_PEERS` environment variable. The **tcp_transport_tls.xml** file is based on **tcp_transport.xml** and uses a WAN configuration to establish communication. 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.

Note: To run this example, you need the *RTI TCP Transport*, which is shipped with

RTI Connext DDS. Additionally, you will need to install the optional packages RTI TLS support and OpenSSL.

– **Using a Secure Connection over LAN**

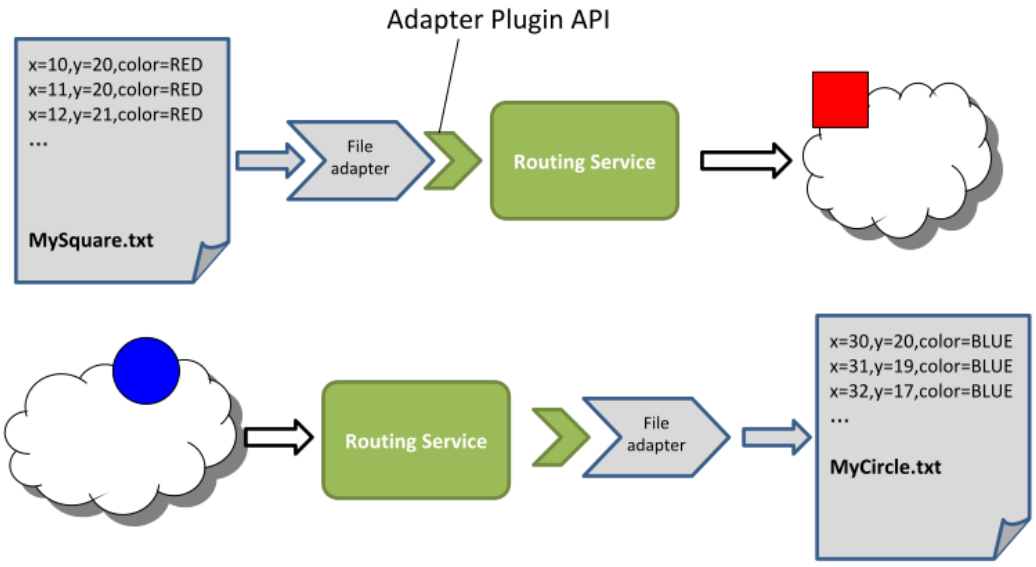
Similar to the previous point, but instead you will use the file **tcp_transport_tls_lan.xml** and prefix **tlsv4_lan://**.

11.10 Example: Using a File Adapter

The previous examples showed how to use *Routing Service* with *Connext DDS*. In this one you will learn how to use *RTI Routing Service Adapter SDK* to create an adapter that writes and reads data from files. *Routing Service* allows you to bridge data from different data domains with a pluggable adapter interface.

To learn how to implement your own adapter, you can follow this example and the next examples and inspect the code that is distributed with these adapters. The file adapter can read data from files with a specific format and provide it to *Routing Service*, or receive data from *Routing Service* and write it into files.

In this example, we will first write topic data (a colored square and circle) into a file and then use that file to write it back into *Connext DDS*, allowing us to modify the data with a text editor.



- Compile the Adapter in **adapters/file/src**:
  - Set NDDSHOME to point to your *RTI Connext DDS* installation.
  - On UNIX-based systems, enter:

```
cd <path to examples>/routing_service/adapters/file/make
gmake -f Makefile.<architecture>
```

The adapter shared library, **libfileadapter.so**, will be copied to **<path to examples>/routing_service/adapters/file**.

- On Windows systems:
  - Open the Visual Studio solution under **<path to examples>/routing_service/adapters/file/windows**. For example, if you are using Visual Studio 2013, open **fileadapter-vs2013.sln**.
  - Build the solution. The adapter shared library, **fileadapter.dll**, will be copied to **<path to examples>/routing_service/adapters/file**.

- From *Connext DDS* to files:
  1. Run *Shapes Demo* and *Routing Service* as in the previous examples:
     - Start *Shapes Demo* on domain 0 (the default domain).
     - Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
  -cfgFile <path to examples>/routing_service/shapes/file_bridge.xml \
  -cfgName dds_to_file
```

  2. In *Shapes Demo*, publish some Squares.
  3. Wait a few seconds and then stop *Routing Service* by pressing **Ctrl-c**.
  4. A file called **MySquare.txt** should have been created in the current directory. Open it with a text editor of your choice. It should contain several lines, each consisting of a list of `<field>=<value>` elements. Each line represents a sample (Square) published by *Shapes Demo* and written by *Routing Service* and the file adapter.

     On UNIX-based systems, you can see how new samples are appended to the file by running the following command while *Routing Service* runs:

```
tail - f MySquare.txt
```

- From a file to *Connext DDS*:
  1. In *Shapes Demo*, delete the Square publisher and create a Square subscriber.
  2. Start *Routing Service* by entering the following in a command shell:

```
cd <NDDSHOME>
bin/rtiroutingservice
     -cfgFile <path to examples>/routing_service/shapes/file_bridge.xml \
     -cfgName file_to_dds
```

     You should see squares being received by *Shapes Demo*. These samples come from what we recorded before.

     You might have noticed that the rate at which the shape moves is much slower. This is the rate at which the file adapter is providing data to *Routing Service*. To change this rate, open **file_bridge.xml** and look for `<route name="square_file">` within

`<routing service name="file_to_dds">`. In the `<property>`` tag, change the property **ReadPeriod** from 1000 (milliseconds) to 100.

3. Stop *Routing Service* and restart it as described in previous steps. The squares should be received and displayed about ten times faster.
4. Other properties that you can configure in the file adapter are: In the input, FileName, MaxSampleSize, Loop and SamplesPerRead; in the output, FileName.
5. You can also edit the text file and publish the new data. Open **MySquare.txt** and replace all the occurrences of "shapesize=30" with "shapesize=100".
6. Stop *Routing Service* and restart it as described in previous steps. The squares will have the same position and color, but they will be bigger now.

- Customize the File Adapter:

  In the example, the file adapters use a specific format, which you already saw in the file **MySquare.txt**. Now try adapting the example to your own format.

  The code that reads/writes from the file is in **adapters/file/src/LineConversion.c**.

  1. Edit the function `RTI_RoutingServiceFileAdapter_read_sample` to implement how file data maps into a sample.
  2. Edit the function `RTI_RoutingServiceFileAdapter_write_sample` to implement how a sample is written to a file.

11.11 Example: Using a Shapes Processor

This example shows how to implement a custom *Processor* plug-in, build it into a shared library and load it with *Routing Service*.

This example illustrates the realization of two common enterprise patterns: aggregation and splitting. There is a single plug-in implementation, *ShapesProcessor* that is a factory of two types of *Processor*, one for each pattern implementation:

- *ShapesAggregator*: *Processor* implementation that performs the aggregation of two *ShapeType* objects into a single *ShapeType* object.
- *ShapesSplitter*: *Processor* implementation that performs the separation of a single *ShapeType* object into two *ShapeType* objects.

In the example, these processors are instantiated as part of a *TopicRoute*, in which all its inputs and outputs represent instantiations of the *Connext DDS Adapter StreamReader* and *StreamWriter*, respectively.

You can find the full example in the RTI Community Examples Repository.

Chapter 12

Common Infrastructure

12.1 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 12.1 shows a general view of this concept.

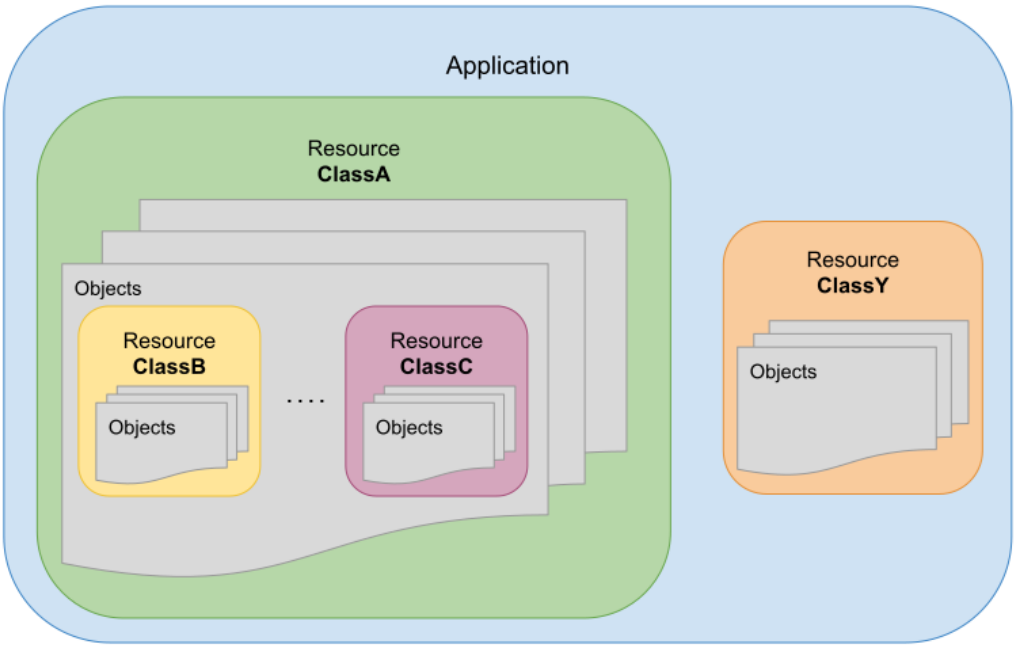


Figure 12.1: Application modeled as a set of related Resources

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.

12.1.1 Example: Simple Resource Model of a Connext DDS Application

Figure 12.2 depicts a UML class diagram to provide a generic resource model for *Connext DDS* applications.

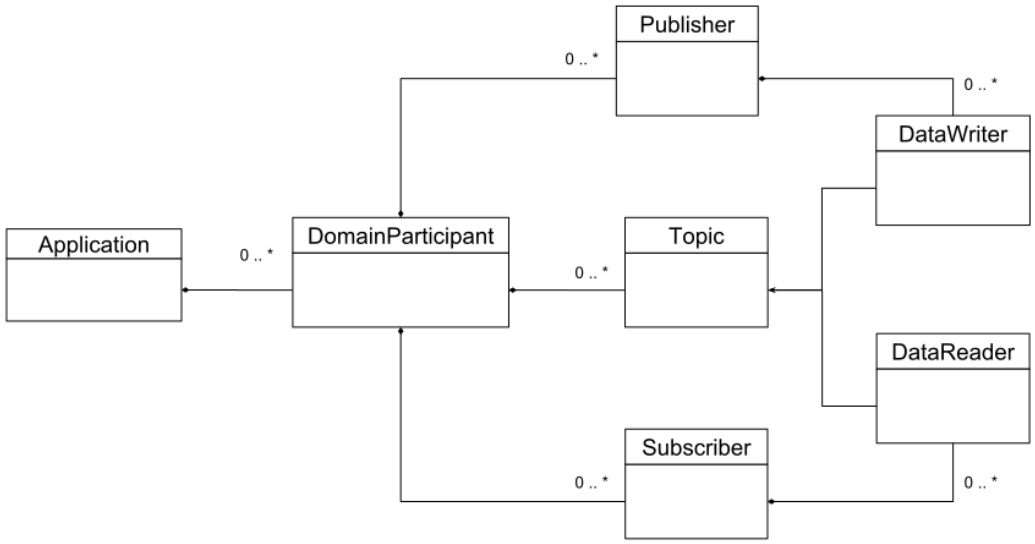


Figure 12.2: Connext DDS application resource model

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.

12.1.2 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:

1. Fixed and specified by the resource model of the parent Resource class.
2. 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 DDS Application

Consider the *Connext DDS* application resource model in Section 12.1.1. 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"
```

12.2 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,* and *RTI Queuing Service.* 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* (Section 12.1) 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 12.3. 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*.

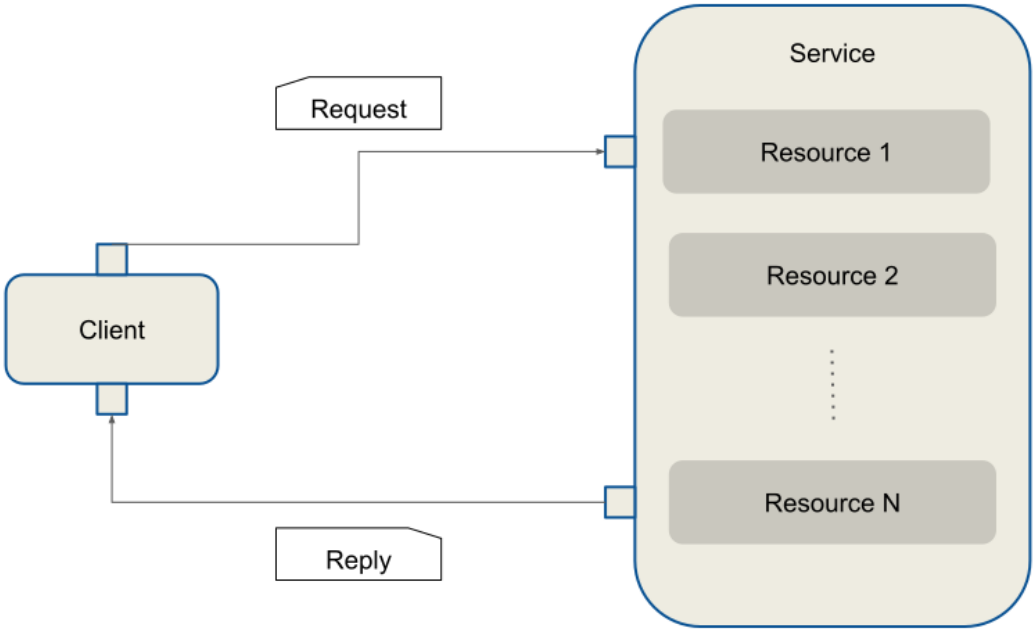


Figure 12.3: General View of the *RTI Remote Administration Platform* Architecture

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.

12.2.1 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 12.1: 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 Section 12.2.1). • `resource_identifier`: Addresses a concrete service resource. Each concrete service has its own set of resources (see Section 12.1.2). • `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 12.2.

Table 12.2: 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
```

12.2.2 Communication

The information exchange between client and server is based on the DDS request-reply pattern, as shown in Figure 12.4. The client maps to a *Requester*, whereas the server maps to a *Replier*.

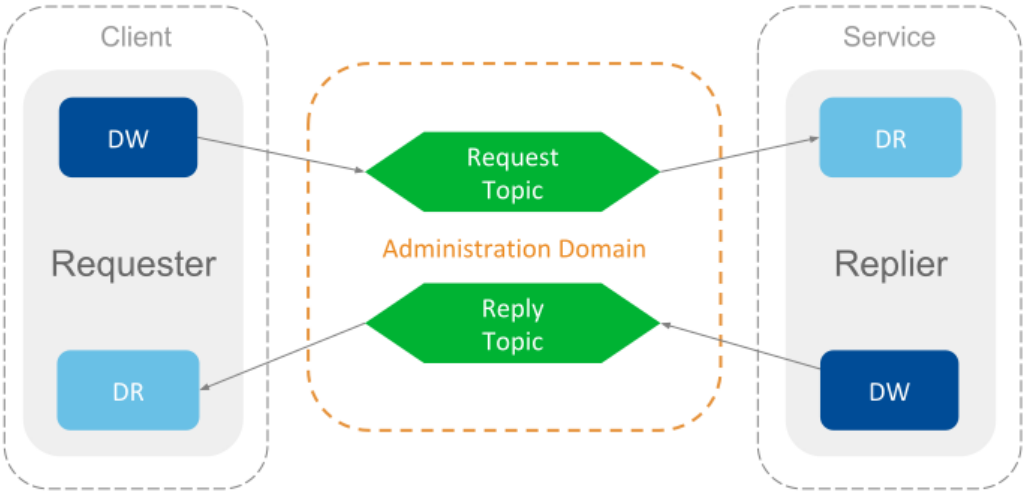


Figure 12.4: Communication in *RTI Remote Administration Platform* is Based on DDS Request-Reply

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 12.3:

Table 12.3: Remote Administration *Topic*s

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

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

Listing 12.1: `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 12.4: `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 12.2: `CommandReply` Type

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

Table 12.5: `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: Accessing from Connext DDS Application

This example shows a Modern C++ snippet on how to use *Connext DDS* Request-Reply to disable a *Routing Service* instance.

```
using namespace rti::request;
using namespace dds::core;
using namespace RTI::Service;
using namespace RTI::Service::Admin;

const unsigned int WAIT_TIMEOUT_SEC_MAX = 10;
const unsigned int ADMIN_DOMAIN_ID = 55;

int main(int, char *[]) {

    try {

        dds::domain::DomainParticipant participant(ADMIN_DOMAIN_ID);

        // create requester params
        rti::request::RequesterParams requester_params(participant);
        requester_params.request_topic_name(COMMAND_REQUEST_TOPIC_NAME);
        requester_params.reply_topic_name(COMMAND_REPLY_TOPIC_NAME);

        // Wait for Routing Service Discovery
        dds::core::status::PublicationMatchedStatus matched_status;
        unsigned int wait_count = 0;

        std::cout << "Waiting for a matching replier..." << std::endl;
        int wait_count = 0;
        while (matched_status.current_count() < 1
                && wait_count < WAIT_TIMEOUT_SEC_MAX) {
            matched_status = requester.request_datawriter().publication_matched_status();
            wait_count++;
            rti::util::sleep(Duration(1));
        }

        if (matched_status.current_count() < 1) {
            throw dds::core::Error("No matching replier found.");
        }

        /*
         * Setup command
         */
        CommandRequest request;
        request.action(CommandActionKind::RTI_SERVICE_COMMAND_ACTION_UPDATE);
        request.resource_identifier("/routing_services/MyRouter/state");
        dds::topic::topic_type_support<EntityState>::to_cdr_buffer(
                reinterpret_cast<std::vector<char> &>(request.octet_body()),
                EntityState(EntityStateKind::DISABLED));
```

(continues on next page)

(continued from previous page)

```
        /*
         * Send disable
         */
        requester.send_request(request);
        CommandReply reply = requester.receive_reply(
                Duration(WAIT_TIMEOUT_SEC_MAX));
        if (reply.retcode() == CommandReplyRetcode::OK_RETCODE) {
            std::cout << "Command returned: " << reply.string_body() << std::endl;
        } else {
            std::cout << "Unsuccessful command returned value "
                      << reply.retcode() << "." << std::endl;
            throw dds::core::Error("Error in replier");
        }

    } catch (const std::exception& ex) {
        std::cout << "Exception: " << ex.what() << std::endl;
        return -1;
    }

    return 0;
}
```

12.2.3 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.

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

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.

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.

12.3 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 12.6 provides a summary of these topics.

Table 12.6: 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 12.5 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*.

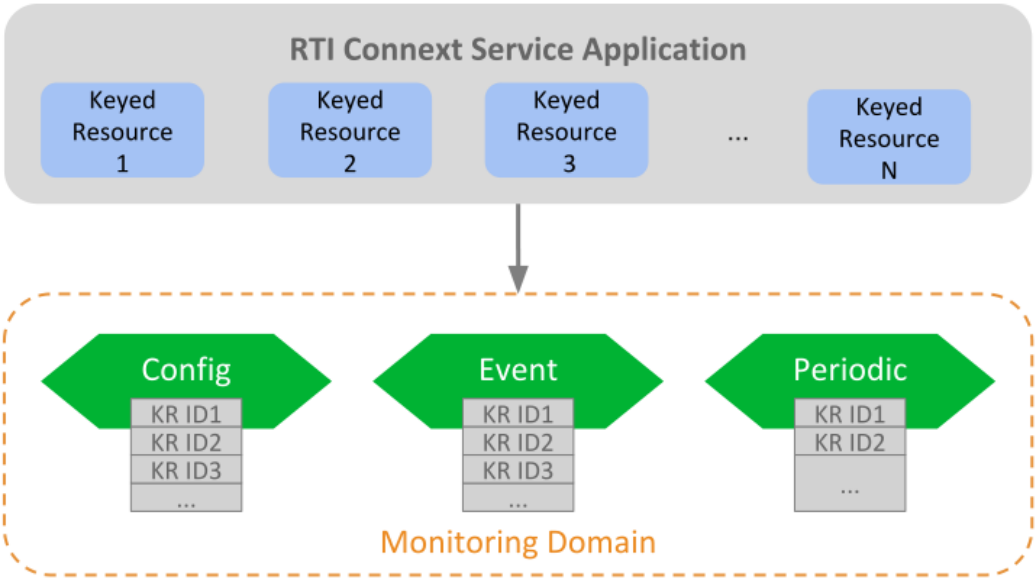


Figure 12.5: Monitoring Distribution *Topic*s of *RTI* Services

12.3.1 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 12.6.

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

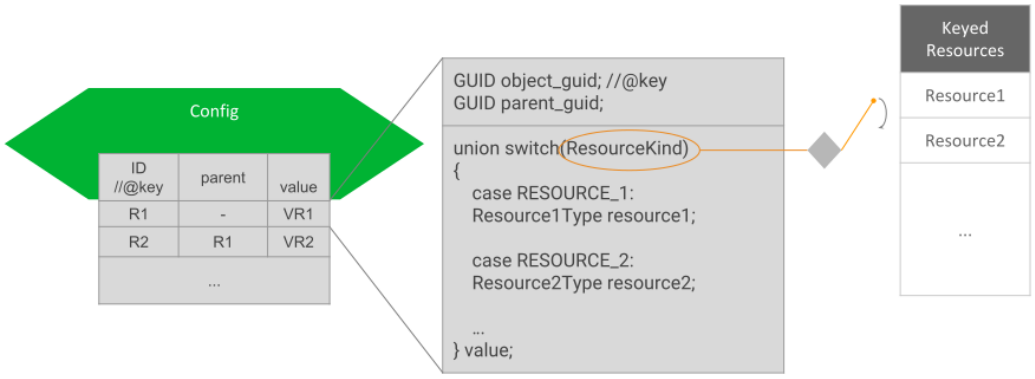


Figure 12.6: Monitoring Distribution *Topic* Definition

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;
```

(continues on next page)

(continued from previous page)

```
};

@nested
struct BoatType {
    float knots;
    float latitude;
    float longitude;
};

@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 12.7: 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

12.3.2 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.

12.3.3 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 Section 12.3.4).

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 `<statistics_publication_period>`.The value of a statistic variable corresponds to the time window of a publication period.

12.3.4 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 12.3: Statistics

```
@appendable @nested
struct StatisticMetrics {
    int64 period_ms;
    int64 count;
    float mean;
    float minimum;
    float maximum;
    float std_dev;
};

@appendable @nested
struct StatisticVariable {
    StatisticMetrics publication_period_metrics;
};
```

Table 12.8: `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 12.4: 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;
};
```

(continues on next page)

(continued from previous page)

```
@appendable @nested
struct HostConfig {
    BoundedString name;
    uint32 id;
    int64 total_memory_kb;
    int64 total_swap_memory_kb;
};
```

Table 12.9: `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 12.10: `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 12.5: Process Types

```
@appendable @nested
struct ProcessConfig {
    uint64 id;
};
@mutable @nested
struct ProcessPeriodic {
```

(continues on next page)

(continued from previous page)

```
        @optional StatisticVariable cpu_usage_percentage;
        @optional StatisticVariable physical_memory_kb;
        @optional StatisticVariable total_memory_kb;
        int32 uptime_sec;
    };
```

Table 12.11: `ProcessConfig`

Field Name	Description
id	Identifies the process where the service is running. The meaning of this value is platform dependent.

Table 12.12: `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 that the processor 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 12.6: Base Entity Types

```
@mutable @nested
struct EntityConfig {
    ResourceId resource_id;
    XmlString configuration;
};
@mutable @nested
struct EntityEvent{
    EntityStateKind state;
};
```

Table 12.13: `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 12.14: `EntityEvent`

Field Name	Description
state	State of the resource entity expressed as an enumeration of type `EntityStateKind`.

Network Performance Metrics

Listing 12.7: Network Performance Type

```
@appendable @nested
struct NetworkPerformance {
    @optional StatisticVariable samples_per_sec;
    @optional StatisticVariable bytes_per_sec;
    @optional StatisticVariable latency_millisec;
};
```

Table 12.15: `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.

12.4 Plugin Management

Some RTI services allows for custom behavior through the use of *pluggable* components or *plugins* . The type of plugins is described in Section 7. A plugin is represented as a top-level service-owned object whose main role is a factory of other pluggable components, which themselves are responsible for providing the user-defined behavior.

Figure 12.7 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 12.7 also shows that there are two mechanisms through which a *Service* obtains a plugin object: a *shared library* or the Service API. Both mechanisms are complementary and are described with more detail in the next sections.

12.4.1 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 Section 12.4.1). 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 a RTI service is deployed through an already linked executable, such as the shipped command-line executable (Section 3.1).

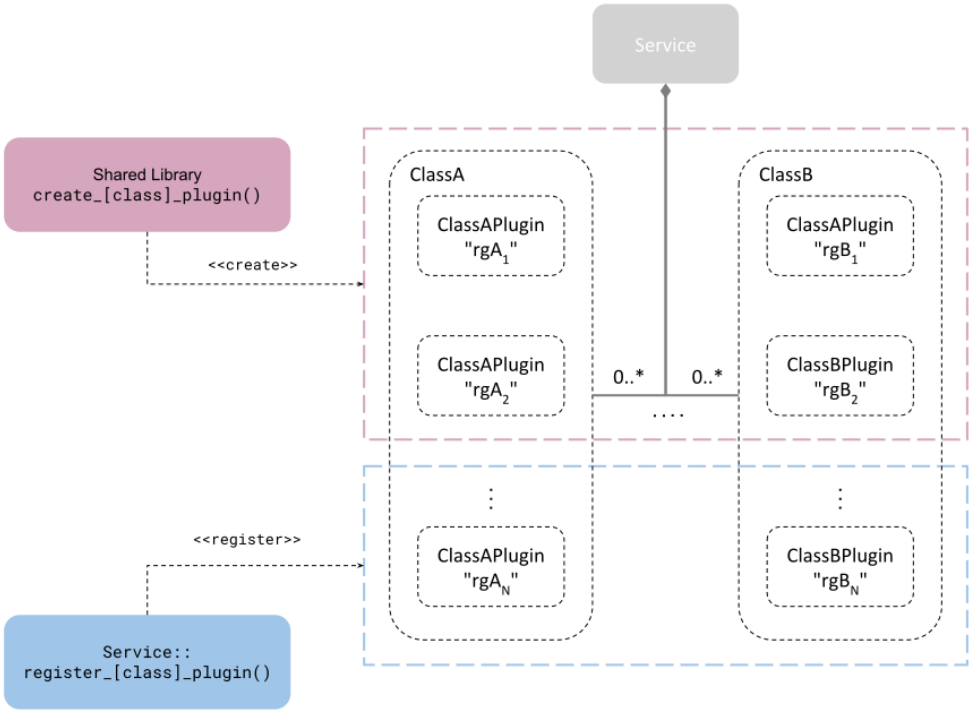


Figure 12.7: Plugin object management

The plugin lifecycle is as follows:

1. After start, *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. *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 12.16 and Table 12.17 describe the configuration of each different plugin language.

Table 12.16: 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**: • UNIX-based systems: **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 Section 3) • [**executable_dir**]: Directory where the executable lives	1
<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 12.17: Configuration tags for Java plugins

Tags within <java_[class]_plugin>	Description	Multi-plicity
<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

12.4.2 Service API

The user provides the plugin object via the Service 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.

Plugin lifecycle goes as follows:

1. 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.

Chapter 13

Release Notes

13.1 Supported Platforms

RTI Routing Service is supported on the platforms in Table 13.1.

It can also be deployed as a C library linked into your application. This is true for all platforms in Table 13.1 except INTEGRITY®.

Table 13.1: Supported Platforms

Platform	Operating System
INTEGRITY	INTEGRITY 10.0.2 on x86 CPU
	INTEGRITY 11.0.4 on x86 CPU
Linux®	All Linux platforms in the *RTI Connext DDS Core Libraries Release Notes* for the same version number, except Wind River® Linux 7.
macOS®	All macOS platforms in the *RTI Connext DDS Core Libraries Release Notes* for the same version number.
QNX®	All QNX Neutrino® 7.0 platforms in the *RTI Connext DDS Core Libraries Release Notes* for the same version number.
Windows®	All Windows platforms in the *RTI Connext DDS Core Libraries Release Notes* for the same version number.

Routing Service is also supported on the platforms in Table 13.2; these are target platforms for which RTI offers custom support. If you are interested in these platforms, please contact your local RTI representative or email sales@rti.com.

Table 13.2: Custom Platforms

Platform	Operating System
Linux	Wind River Linux 8 on Arm® v7 and PPC e6500
	Yocto Project 2.5 on Arm v7
QNX	QNX Neutrino 6.5 on PPC (e500v2)
	QNX Neutrino 7 on Arm v7

13.2 Compatibility

For backward compatibility information between *Routing Service* 6.0.1 and previous releases, please see the *Migration Guide* on the RTI Community portal.

Routing Service can be used to forward and transform data between applications built with *Connext DDS*, as well as *RTI Data Distribution Service* 4.5[b-e], 4.4d, 4.3e, and 4.2e except as noted below.

- *Routing Service* is not compatible with applications built with *RTI Data Distribution Service* 4.5e and earlier releases when communicating over shared memory. For more information, please see the Transport Compatibility section in the *Migration Guide* on the RTI Community portal.
- Starting in *Connext DDS* 5.1.0, the default `message_size_max` for the UDPv4, UDPv6, TCP, Secure WAN, and shared-memory transports changed to provide better out-of-the-box performance. *Routing Service* also uses the new value for `message_size_max`. Consequently, *Routing Service* is not out-of-the-box compatible with applications running older versions of *Connext DDS*. Please see the *RTI Connext DDS Core Libraries Release Notes* for instructions on how to resolve this compatibility issue with older *Connext DDS* applications.
- The types of the remote administration and monitoring topics in 5.1.0 are not compatible with 5.0.0. Therefore:
  - The 5.0.0 *RTI Routing Service* shell, *RTI Admin Console* 5.0.0, and *RTI Connext DDS* 5.0.0 user applications performing monitoring/administration are not compatible with *RTI Routing Service* 5.1.0.
  - The 5.1.0 *RTI Routing Service* shell, *RTI Admin Console* 5.1.0, and *RTI Connext DDS* 5.1.0 user applications performing monitoring/administration are not compatible with *RTI Routing Service* 5.0.0.

13.3 What's New in 6.0.1

13.3.1 New platforms

Routing Service now includes support for these platforms:

- macOS 10.14 (x64)
- QNX Neutrino 7.0 (x64 and Arm v8 x64)
- Red Hat® Enterprise Linux 8 (x64)
- Wind River Linux 8 (PPC e6500) (custom target platform)
- Windows 10 (x86, x64) with Visual Studio® 2019
- Windows Server 2016 (x86, x64) with Visual Studio 2019
- Yocto Project® 2.5 (Arm v7) (custom target platform)

For more information on these platforms, see the *RTI Connext DDS Core Libraries Platform Notes* for this release.

13.3.2 Removed platforms

- macOS 10.11
- Windows 7
- Windows Server 2008 R2

13.3.3 Earlier detection of invalid configurations

Routing Service successfully loaded configurations that contained an invalid definition. Specifically, it loaded a configuration in which a DomainRoute that contained one or more Sessions but no Participants/Connections. Nevertheless, the service failed to start because the corresponding DomainRoute object failed to instantiate. This problem has been resolved and the service will not load an invalid DomainRoute configuration.

13.3.4 Added Support for Proxy of TopicQueries in Routes with Multiple Inputs and Outputs

TopicQuery proxy mode is now supported in TopicRoutes that contain more than one input and/or more than one output.

13.4 What's Fixed in 6.0.1

13.4.1 QoS Topic Filters were not supported

Routing Service entities previously ignored QoS Topic Filters. This problem has been resolved. The entities listed below will obtain the QoS within the selected profile having a matching topic filter based on the actual name of the Topic they are associated with.

- Route's DDS inputs and outputs
- AutoRoute's DDS input and output
- Remote Administration Replier
- Monitoring DataWriters

For example, consider the following QoS profile, which uses topic filters:

```
<qos_library name="QosLib">
    <qos_profile name="QosProfile"">
        <datawriter_qos>...</datawriter_qos>
        <datawriter_qos topic_filter="MyTopic">
            ...
        </datawriter_qos>
        <datareader_qos topic_filter="MyTopic">
            ...
        </datareader_qos>
        <datawriter_qos topic_filter="rti/service/admin/command_reply">
            ...
        </datawriter_qos>
        <datareader_qos topic_filter="rti/service/admin/command_request">
```

(continues on next page)

(continued from previous page)

```
            ...
        </datareader_qos>
        <datawriter_qos topic_filter="rti/service/monitoring/periodic">
            ...
        </datawriter_qos>
    </qos_profile>
</qos_library>
```

And the following *Routing Service* configuration:

```
<routing_service name="MyRouter">
    <administration>
        <writer_qos  base_name="QosLib::QosProfile" />
        <reader_qos base_name="QosLib::QosProfile" />
    </administration>
    <monitoring>
        <writer_qos  base_name="QosLib::QosProfile" />
    </monitoring>

    ....
          <topic_route>
              <input>
                 <topic_name>MyTopic>
                 <reader_qos  base_name="QosLib::QosProfile" />
              </input>
              <output>
                 <topic_name>MyTopic>
                 <writer_qos  base_name="QosLib::QosProfile" />
              </output>
         </topic_route>

         <auto_topic_route>
              <input>
                 <reader_qos  base_name="QosLib::QosProfile" />
              </input>
              <output>
                 <writer_qos  base_name="QosLib::QosProfile" />
              </output>
         </auto_topic_route >
    ...
</routing_service>
```

The corresponding DDS entities below will be created with the QoS that matches the topic filter based on the topic name:

DDS Entity	QoS from QosLib::QosProfile	Topic filter
TopicRoute's input DataReader	reader QoS	MyTopic
TopicRoute's output DataWriter	writer QoS	MyTopic
AutoTopicRoute's input DataReader for topic name=MyTopic	reader QoS	MyTopic
AutoTopicRoute's output DataWriter for topic name=MyTopic	writer QoS	MyTopic
AutoTopicRoute's input DataReader for topic name=Other	reader QoS	NULL
AutoTopicRoute's output DataWriter for topic name=MyTopic	writer QoS	NULL
Administration Replier's DataReader	reader QoS	rti/service/admin/command_request
Administration Replier's DataWriter	writer QoS	rti/service/admin/command_reply
Monitoring Periodic DataWriter	writer QoS	rti/service/monitoring/periodic
Monitoring Config and Event DataWriter	writer QoS	NULL

[RTI Issue ID ROUTING-37]

13.4.2 Executable did not log build ID for DDS libraries

The executable version of *Routing Service* did not log the build ID of DDS libraries, no matter which verbosity level was specified. This was a regression from the previous version where the build ID was logged for the warning verbosity. This problem has been resolved.

[RTI Issue ID ROUTING-601]

13.4.3 Remote create operation failed with resource identifiers formatted as noted in User's Manual

The remote create operation failed when the resource identifier was formatted as stated in the user's manual API reference: `<parent_resource_id>/collectionA`

The operation failed with an unsupported resource error. Instead, the operation succeeded when the resource identifier was provided as follows: `<parent_resource_id>`

This problem has been resolved. The remote create operation now accepts the resource identifier formatted as indicated in the user's manual remote API reference. Although syntactically incorrect, it also continues to accept the previous format to preserve backwards compatibility.

[RTI Issue ID ROUTING-603]

13.4.4 Unbounded generation of file handles if monitoring enabled on QNX platforms

If monitoring was enabled, a new file handle was periodically generated and never released. Eventually this situation led to the following errors:

```
DL Debug: : RTIOsapiInterfaceTracker_updateInterfacesUnsafe:!get interfaces failed
DL Error: : ROUTERProcess_getMemoryUsage:!get open /proc/self/as (QNX)
DL Error: : ROUTERProcess_getProcessInfo:!memory usage
DL Error: : ROUTERMonitorableApplication_sampleStatus:!get process info
```

This problem, which only affected QNX platforms, has been resolved.

[RTI Issue ID ROUTING-614]

13.4.5 Inconsistent state if remote operation performed on disabled DomainRoute

If a remote operation was performed on a disabled DomainRoute, *Routing Service* entered an inconsistent state that may have caused a crash during shutdown or while processing other remote operations. This problem has been resolved.

[RTI Issue ID ROUTING-616]

13.4.6 Changing Session period through Route's API updated the period, but with a delay

Calling the route API to change the period of the parent Session within Processor implementations resulted in a period update delay that was at least equal to the current period. (This periodic event can be enabled via the `<periodic_action>` tag in XML.)

The observed behavior was that the new period value did not occur until after one more periodic event notifications occurred at the current period value. This problem has been resolved.

[RTI Issue ID ROUTING-620]

13.4.7 Added operations in Processor API to access DataReader/Writer of a DDS input/output

The Processor API has been extended with operations to access the underlying DataReader and DataWriter of a DDS input and output, respectively. Namely, the following two operations have been added:

```
dds::sub::DataReader<Data> rti::routing::processor::Input::dds_data_reader();

dds::pub::DataWriter<Data> rti::routing::processor::Output::dds_data_writer();
```

You can use these new operations when the Input and Output hold an instance of a DDS StreamReader and DDS StreamWriter, respectively (the case of `<topic_route/auto_topic_route>`, or `<dds_input>` and `<dds_output>`). In this case, you can use these operations as follows:

```
using dds::core::xtypes::DynamicData;

dds::sub::DataReader<DynamicData> input_reader =
      route.input<DynamicData>(0).dds_data_reader();
...
dds::pub::DataWriter<DynamicData> output_writer =
      route.output<DynamicData>(0).dds_data_writer();
```

[RTI Issue ID ROUTING-623]

13.4.8 Unexpected routes created after disabling and enabling AutoRoutes

Issuing two successive commands to disable and enable an AutoRoute caused the generation of unexpected Routes, if there were already existing and matching streams. This problem has been resolved.

[RTI Issue ID ROUTING-626]

13.4.9 Routing Service failed to detect configuration with duplicate names

When loading a configuration file, *Routing Service* did not detect that the configuration contained more than one entity with the same name and within the same parent configuration. It allowed this configuration to be loaded.

For example, the following is invalid because the `<route>` entities within the same session have the same name:

```
<session>
    <route name="Route">...</route>
    <route name="Route">...</route>
</session>
```

This problem has been resolved. Now *Routing Service* will prevent such a configuration from being loaded and will log a message indicating the reason.

[RTI Issue ID ROUTING-634]

13.4.10 Executable always ignored logging QoS

Logging settings specified in the DomainParticipantFactory QoS were always ignored when running *Routing Service* with the shipped executable. This problem has been resolved. Now the logging settings are applied and overwritten only when the `-logFormat` option is provided.

[RTI Issue ID ROUTING-641]

13.4.11 Out of memory error if Monitoring enabled on QNX platforms

On QNX platforms, you may have received the following output when enabling service monitoring:

```
ROUTERProcess_getMemoryUsage:!get Number of entries greater than the limit,
you should consider increasing such limit
```

This issue occurred because some memory was statically reserved to allocate the data structure necessary to measure memory usage. As the application grew, however, the structure should have also grown, but didn't. This problem has been resolved. Now the structure is reserved dynamically and can measure bigger applications.

[RTI Issue ID ROUTING-643]

13.4.12 Segmentation fault when reading from custom processor if underlying StreamReader didn't return SampleInfo list

Reading data from an `rti::routing::Processor::Input` caused a segmentation fault if its underlying `rti::routing::adapter::StreamReader` returned a NULL sample info list. This problem has been resolved.

[RTI Issue ID ROUTING-645]

13.4.13 Failure to remotely create entity resulted in XML object inserted in loaded DOM

If a remote operation to create an entity failed due to an error instantiating the entity object, the associated XML object remained loaded in the service configuration Document Object Model (DOM). This could cause issues such as:

- An entity duplication error if a retry to create the entity was issued.
- Failures when remotely restarting *Routing Service* due to an attempt to instantiate the object associated with the XML configuration object.

This problem has been resolved and the service DOM will remain unchanged in case of a creation failure.

[RTI Issue ID ROUTING-646]

13.4.14 Undefined behavior if entity names contained characters ":" or "/"

Routing Service's behavior was undefined if it loaded a configuration that defined elements with a `name` attribute containing the characters `/` or `:`. This issue was caused by a conflict in the generation resource identifiers, which uses `/` as a resource separator and `:` as a custom method separator.

This problem has been resolved. If the name of an Entity contains any of these characters (for example, when specified in the `name` attribute of their corresponding element in XML), the `name` will be enclosed in double quotation marks (`"`).

For example:

```
...
<routing_service name="Service">
    <domain_route name="DomainRoute/Wan"> ... </domain_route>
    <domain_route name="DomainRoute::Lan"> ... </domain_route>
</routing_service >
...
```

will generate as Resource identifiers:

```
/routing_services/Service/"DomainRoute/Wan"
/routing_services/Service/"DomainRoute::Lan"
```

[RTI Issue ID ROUTING-652]

13.4.15 XML variables outside of <routing_service> were not expanded

Routing Service failed to expand any XML variables defined in any element that was not part of the `<routing_service>` tag, such as those in a `<qos_profile>`. This problem has been resolved. All XML variables will be expanded, regardless of their location.

[RTI Issue ID ROUTING-653]

13.5 Previous releases

13.5.1 What's New in 6.0.0

New platforms

Routing Service is now supported on the platforms in Table 13.3. For more information, see the *Connext DDS Core Libraries Platform Notes*.

Table 13.3: New Platforms

Platform	Operating System
Linux	Ubuntu 18.04 LTS on x64
	Wind River Linux 8 on ARMv7 (custom support)

Support for multiple connections in a domain route

Routing Service has been enhanced to support the creation of more than two connections within a domain route. This includes the ability to specify more than two builtin DDS adapter DomainParticipants.

The XML tags <connection_1>, <connection_2>, <participant_1> and <participant_2> have been deprecated. *Routing Service* will still load legacy configurations, but support may be dropped in future releases. Instead of these tags, new tags <connection> and <participant> have been introduced. They are defined just like the legacy tags, with the addition of a mandatory 'name' attribute. This attribute is used by a route's input and output to indicate the connection the inputs and outputs are created from. For example, the snippet below defines a domain route with two participants and a custom, adapter-based connection:

```
<domain_route name="MyDomainRoute">
    <participant name="participant_0">
        ...
    </participant>
    <participant name="participant_1">
        ...
    </participant>
    <connection
        name="customConnection"
        plugin_name="MyAdapteRPlugin>
       ...
    </connection>
    ...
<domain_route>
```

See Section 4.5.4.

Support for multiple inputs and outputs in routes or topic routes

Routing Service has been enhanced to support the creation of multiple inputs and outputs in a route. This enhancement includes the ability to specify more than one builtin DDS adapter DataReader (<input>, <dds_input>) and DataWriter (<output>, <dds_output>).

For example, the snippet below defines a topic route with a single input in domain 0 and two outputs in domains 1 and 2 (assume the participants for each input and output have been previously defined):

```
<topic_route name="Route">
    <input participant="domain_0">
        <topic_name>TopicA</topic_name>
        <registered_type_name>
            MyType
        </registered_type_name>
        ...
    </input>
    <output participant="domain_1">
        <topic_name>TopicB</topic_name>
        <registered_type_name>
            MyType
        </registered_type_name>
        ...
    </output>
    <output participant="domain_2">
        <topic_name>TopicA</topic_name>
        <registered_type_name>
            MyType
        </registered_type_name>
        ...
    </output>
</topic_route>
```

See Section 4.5.6.

Support for C++ Adapter, Transformation and Service APIs

Adapter, Transformation and Service APIs are now supported in C++.

See Section 7.

New pluggable processor API

One of the star features of *Routing Service* is a new pluggable component, the *Processor*, which allows you to control the operation and data forwarding logic of *Route*s.

A *Processor* defines an interface with access to the *Route* and its inputs and outputs, allowing you to read, manipulate, and write user data under *events* such as data arrival or periodic notification.

For example, the following snippet shows a simple example of data aggregation:

```
on_data_available(Route& route)
{
    auto output = route.output<DynamicData>("info");
    auto info = output.create_data();
    for(const auto& status : route.input<DynamicData>("status_in").take()) {
        auto periodic  = route.input<DynamicData>("periodic_in")
                .select(status.info().instance_handle()).take();
        info.value<int>("id", status.data().get<int>("id"));
        info.value<int>("config", status.data().get<string>("config"));
        info.value<int>("latency", periodic[0].data().get<int>("latency"));
        output.write(info);
    }
}
```

See Section 7.

Redesigned remote administration architecture

Routing Service's remote administration functionality has been redesigned to have a new, consistent, more usable and scalable architecture. Among the changes in this evolution, it includes:

- A homogenous API based on REST that standardizes the mechanism to invoke remote operations on the service entities.
- A common DDS communication model based on *Connext DDS Request-Reply*, which is service-independent and represents a common communication channel for all RTI services.
- Enhanced administration *Topic* definitions for increased scalability, with support for content-filtering and instance management.

See Section 5.

Redesigned remote monitoring architecture

Routing Service's remote monitoring functionality has been redesigned to have a new, consistent, more usable and scalable architecture. Among the changes in this evolution, it includes:

- A common DDS communication model that defines only **three** *Topic*s to provide all the monitoring information from the service entities. This DDS model is service-independent and a common communication channel for all RTI services.
- Enhanced monitoring *Topic* definitions to represent a more scalable, instance-based mapping between the service entities and their associated monitoring entities.

See Section 6.

Support for advanced logging

Routing Service internal logging has improved to provide more detailed logging context. With advanced logging, all the operation logs will be part of an activity context that contains information about the entity and the operation. This facilitates the identification of potential issues in the configuration or runtime behavior.

For example, the following log message:

```
[/routing_services/default|CREATE]
```

represents a context of a *Routing Service* create operation. Then any logs generated by operations occurring in this context will be appended to the activity context. For example:

```
[/routing_services/default|CREATE] load URL group='../../resource/xml/RTI_ROUTING_
↪SERVICE.xml'
```

Support for XML variables expansion from command-line and service API

XML configuration variables defined in the form

```
<element>$(MY_VAR)</element>
```

can be expanded, in addition to the existing runtime shell environment, from two new mechanisms:

- A new command-line option, using the notation `-DMY_VAR=my_value`
- A new `ServiceProperty::user_environment` member, which represents a map of variable name-value pairs.

See Section 3.

Paused and disabled state is cleared after disabling an entity

In previous releases, *Routing Service* remembered the paused state of an entity that was disabled. When that entity was re-enabled, it entered the paused or disabled state. This behavior has changed, so that now the paused state is cleared and if the entity is re-enabled, it will not enter the paused state.

This same behavior affected the child entities of the disabled entity. After re-enabling the entity, *Routing Service* remembered its state and re-enabled the child entities. This behavior has changed, so that after enabling an entity, each of its child entities is enabled based its own 'enabled' attribute.

Removed warning caused by multiple registrations of a type

Routing Service with Routes using the DDS adapter may have output the following message:

```
PRESParticipant_registerType:name 'MyTypeRegisteredName ' is not unique
```

This was caused by multiple registrations of the same type with the same registered name. This behavior has changed and the warning has been removed.

13.5.2 What's Fixed in 6.0.0

Remotely disabling TopicRoute/Route could fail while forwarding data

A remote command to disable a TopicRoute or Route may have resulted in an error if the TopicRoute or Route was receiving data at a high rate. In this case, the TopicRoute or Route remained enabled and kept forwarding data. This problem has been resolved.

[RTI Issue ID ROUTING-531]

Routing Service in debug mode did not link with debug version of Distributed Logger

The debug version of *Routing Service* linked with the release version of the RTI Distributed Logger library, rather than the debug version. This may have lead to unexpected behavior, including potential segmentation faults. This problem has been resolved.

[RTI Issue ID ROUTING-533]

Route stream matching not applied correctly in presence of certain partitions

When partitions in either the publication or subscription side were composed of patterns only, route streams would not match correctly. This problem has been resolved.

[RTI Issue ID ROUTING-538]

Crash on shutdown if types provided through both discovery and XML

Routing Service could have randomly crashed during shutdown if it obtained types from both discovery and XML configuration. This problem has been resolved.

[RTI Issue ID ROUTING-539]

Sample loan not returned to DDS input upon DDS_DataReader::get_key() failure

TopicRoutes did not return the sample loan to the DDS inputs if the StreamReader::read() operation failed due to DataReader::get_key() failures.

This situation could occur very rarely and only if the same StreamReader::read() call returned valid samples in combination with samples discarded by a get_key() failure.

This problem could cause memory growth and errors upon service shutdown. For example, the following logs were the output upon service shutdown:

```
[D0330|DELETE_CONTAINED]PRESPsService_destroyAllLocalEndpointsInGroupWithCursor:!delete␣
↪local reader
[D0330|DELETE_CONTAINED]PRESPsService_destroyAllEntities:!destroyAllLocalEndpointsInGroupWithCursor
[D0330|DELETE_CONTAINED]PRESParticipant_destroyAllEntities:!delete service entities
[D0330|DELETE_CONTAINED]PRESParticipant_destroyAllEntities:!delete topic
[D0330|DELETE_CONTAINED]DDS_DomainParticipant_delete_contained_entities:!delete␣
↪contained entities
```

This problem has been resolved.

[RTI Issue ID ROUTING-552]

Enabling monitoring through ServiceProperty::enable_monitoring only worked if <monitoring> tag present

Enabling monitoring by setting ServiceProperty::enable_monitoring did not take effect unless the <monitoring> tag was also part of the XML configuration. This problem has been resolved.

[RTI Issue ID ROUTING-564]

Logged message included inaccurate number of dropped samples

Routing Service logged a message for a StreamWriter::write failure that included an inaccurate number of dropped samples. For example:

```
ROUTERRoutingProcessor_routeToMatchingOutputs: write error on output
at index 1. 1024 samples have been dropped
```

This problem has been resolved.

[RTI Issue ID ROUTING-572]

Deserialization errors may have occurred under some conditions

Routing Service generated deserialization errors if a publication type was compatible but different than the TopicRoute's input type, and the TopicRoute's input was created after the discovery of such publication type. This situation was unlikely to occur. This problem has been resolved.

[RTI Issue ID ROUTING-578]

TopicRoutes with TopicQuery proxy mode enabled forwarded live data only to first output

A TopicRoute with multiple outputs that enabled TopicQuery proxy mode forwarded live data only to the first output. This problem has been resolved; live data is now forwarded to all outputs.

[RTI Issue ID ROUTING-579]

Routing Service Java API did not work with some TypeCodes

Types larger than 65,535 bytes and types that used extensible type features only available in TypeObject weren't properly serialized when using *Routing Service's* Java API. This issue caused participant creation to produce a BAD_TYPECODE exception for large types and could have prevented communication between Java applications and applications using a different language binding for types using extensibility features. This problem has been resolved.

[RTI Issue ID ROUTING-583]

Chapter 14

Copyrights

Trademarks

Copy and Use Restrictions

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/