RTI Routing Service

User's Manual

Version 7.2.0

Your systems.
Working as one.

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®* 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* 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.

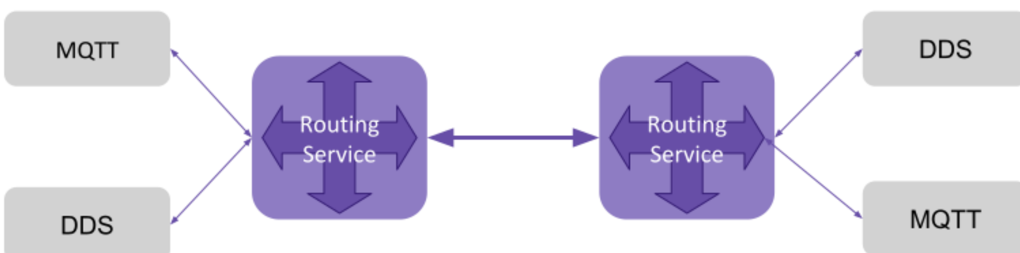


Figure 1.1: Routing Service Overview

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

Key benefits of *Routing Service*:

- It can significantly reduce the time and effort spent integrating and scaling *Connext* 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* 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*.

  The default installation paths are:

  - macOS® systems: `/Applications/rti_connext_dds-version`
  - Linux® systems, non-root user: `/home/your user name/rti_connext_dds-version`
  - Linux systems, root user: `/opt/rti_connext_dds-version`

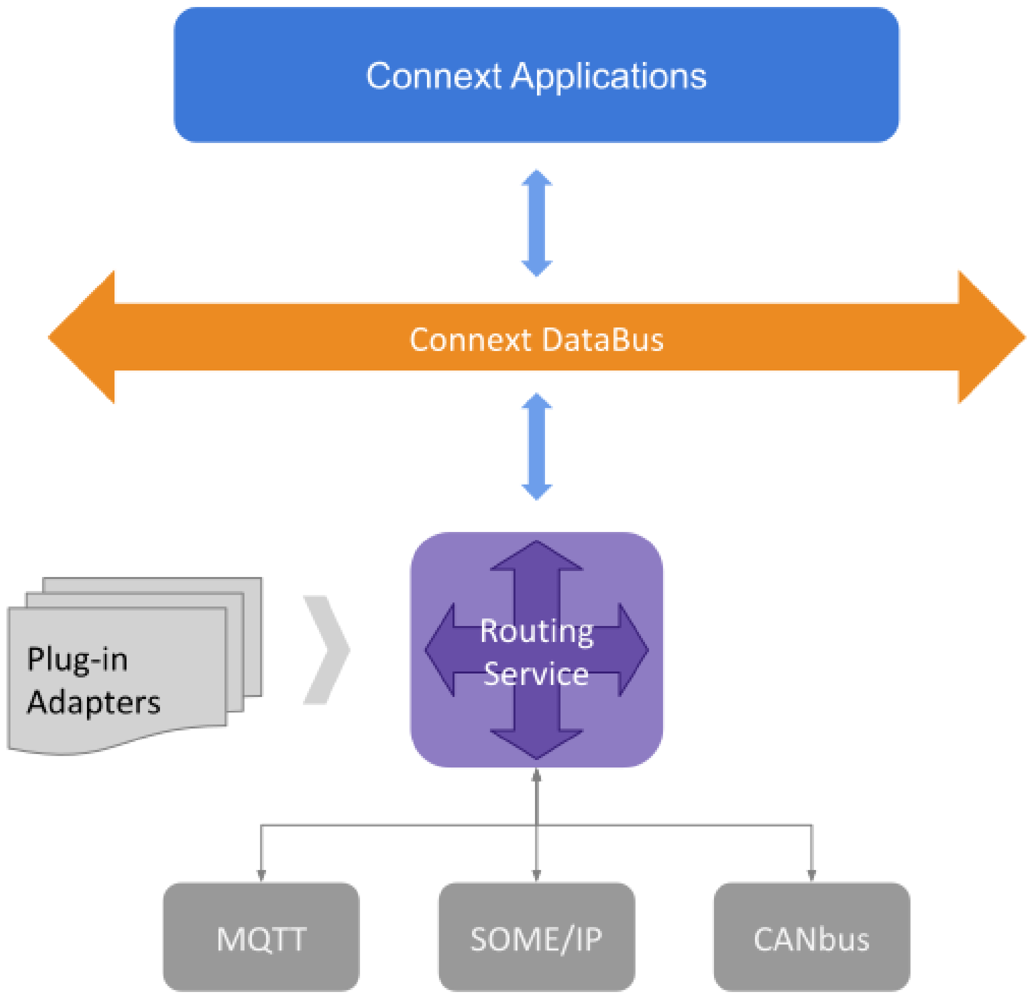


Figure 1.2: Quickly build and deploy bridges between natively incompatible protocols and technologies using *Connext*

  - Windows® systems, user without Administrator privileges: `<your home directory>\rti_connext_dds-version`
  - Windows systems, user with Administrator privileges: `C:\Program Files\rti_connext_dds-version`

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

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

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

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

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

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

  Default path to the examples:

  - macOS systems: `/Users/your user name/rti_workspace/version/examples`
  - Linux systems: `/home/your user name/rti_workspace/version/examples`
  - Windows systems: `your Windows documents folder\rti_workspace\version\examples`. Where `'your Windows documents folder'` depends on your version of Windows. For example, on Windows 10 systems, the folder is `C:\Users\your user name\Documents`.

1.3 Files Mentioned in Documentation

Chapter 2

Routing Data: Connecting and Scaling Systems

This chapter is devoted to present the most elemental function of *Routing Service*: routing data across multiple DDS domains. Routing data refers to the process of propagating the *Topic* user data from one domain to another, allowing systems to interconnect and scale.

Figure 2.1 shows the most basic view of the *Routing Service* model. You can think of it as a black box composed of multiple *Input DataReaders* and *Output DataWriters*, each associated with a specific *Topic*. Data flows from the input *DataReaders* to the output *DataWriters*. The input *DataReaders* receive data from the *publication side*, whereas the output *DataWriters* send data to the *subscription side*.

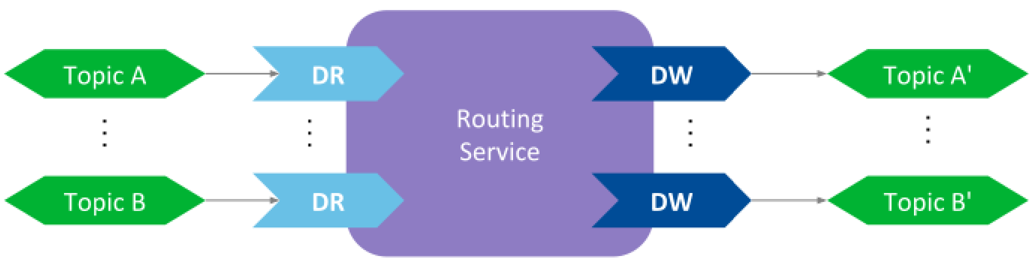


Figure 2.1: Basic model of *Routing Service*

The *Routing Service* engine takes the data from an input *DataReader* and passes it along to a specific output *DataWriter*, as if there was a link connecting input and output. This activity is known as the *forwarding process*. *Routing Service* allows configuring this forwarding process.

The following sections will guide you through all the *Routing Service* entities involved in the forwarding process and how they are configured.

Note: All the following sections assume you are already familiar with basic DDS concepts. Additionally you should be familiar with the *RTI Shapes Demo* tool. Refer to *Tutorials* if you need more information.

2.1 Routing a *Topic* between two different domains

The most basic use case of *Routing Service* is about forwarding the data for a specific *Topic* from one domain to another. This process is known as *routing a Topic*. Figure 2.2 illustrates this concept.

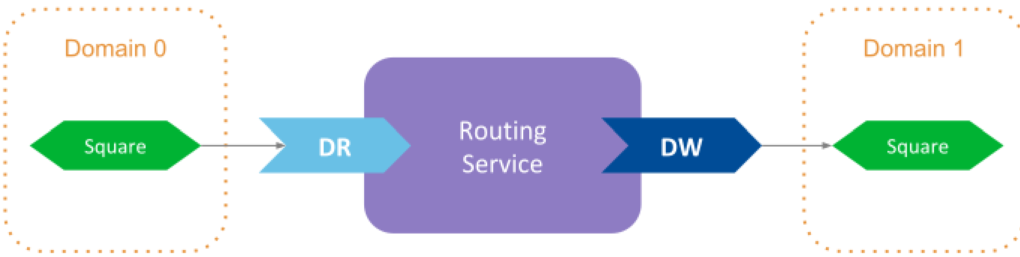


Figure 2.2: Basic *Topic* routing among domains for a *Topic* with name Squares

The samples for the *Topic* named `Square` in domain 0 are forwarded to the same *Topic* but in domain 1. You will first run *Example: Routing a single specific Topic* in your machine to see the functionality in action. Then we will break down all the parts related to *Routing Service*.

Let's review step-by-step each element that appears in the *Routing Service* XML configuration, understanding its purpose and what each of its entities is modeling.

2.1.1 Define the service configuration element

The first step is to define the top-level element for the *Routing Service* configuration:

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

This element defines a configuration profile for *Routing Service*. It must appear within the tag `<dds>`–the root tag for all the elements related to *Connext*–. The configuration shall contain a `name` attribute that uniquely identifies the service, and determines the *service configuration name*. You can define multiple service configurations in one XML file, and **select one to instantiate a** ***Routing Service*** by providing the configurations name with the `-cfgName` option (or `ServiceProperty::cfg_name` member when using the Library API).

As we'll see further below, the `name` attribute is an important concept since it establishes the *configuration name* of a *Routing Service* entity. This name can be used from other elements in the configuration to refer to a specific entity.

See also:

Usage How to run *Routing Service* using the shipped executable or embedding it into your application with the Library API.

Routing Service Tag Reference for the XML configuration of the service element.

2.1.2 Specify which domains to join

Within the top-level *Routing Service* configuration we need to specify which *domains Routing Service* will be joining. The specification of the domains occurs within the *DomainRoute*, which represents a *mapping* between multiple DDS domains through a collection of *DomainParticipants*.

In our example, we are joining domains 0 and 1 and we relay on the default participant QoS settings, so the XML looks as follows:

```
<domain_route name="DomainRoute">

    <participant name="domain0">
        <domain_id>0</domain_id>
    </participant>

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

    ...
</domain_route>
```

You can specify as many *DomainParticipants* as needed. An important aspect to pay attention is the **configuration name assigned to each participant**. This name is what uniquely identifies a domain and is referenced later by *Inputs* and *Outputs* to indicate the *DomainParticipant* from which the *DataReader* and *DataWriter* are created, respectively.

Note: The value specified with `<domain_id>>` in the XML participant configuration can be offset with the `-domainIdBase` command-line option. The participant will be created with domain ID = `<domain_id>` + `-domainIdBase`.

In addition, the `name` attribute of the participant configuration is used to form the name assigned to the actual *DomainParticipant* by setting the EntityName QoS.

See also:

Table 8.8 in *Domain Route* How *Routing Service* constructs the name assigned to the *DomainParticipant*.

Figure 2.3 shows the *DomainRoute* resource model, denoting the association with the service and participant entities.

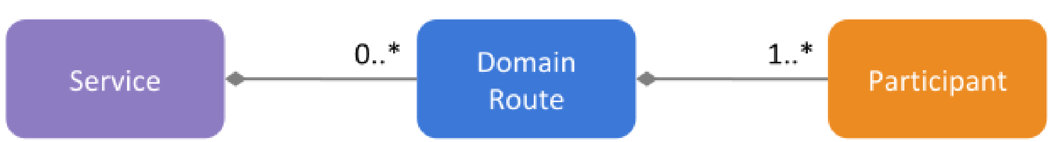


Figure 2.3: *DomainRoute* resource model

2.1.3 Define a processing context

One of the main aspects that contributes to the high performance of *Routing Service* is the ability to parallelize the processing of the data streams. You can create *threading contexts* to execute of all the activities related to the processing of the data streams. A threading context involves **one or more threads**–a thread pool–, and is specified by the *Session* entity.

In our example we define a single *Session* to take care of processing the data for the single *Topic* that is forwarded:

```
<session name="Session">

    ...
</session>
```

The *Session* must appear inside the *DomainRoute* and you can specify as many *Sessions* as you want. In our configuration we rely on the default values, which define a single-threaded context. You could specify a thread pool if, for example, you wanted to parallelize the forwarding of multiple *Topics*.

Figure 2.4 shows the *Session* resource model.

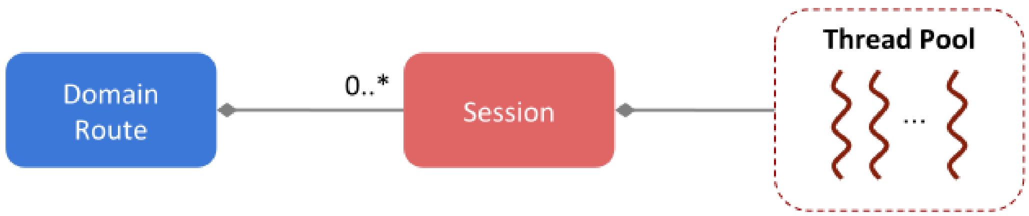


Figure 2.4: *Session* resource model

See also:

Session configuration in *Session* Reference for the XML configuration of the *Session* element.

2.1.4 Define the data flow

The last step consists of defining the flow of *data streams*. For the *Topic routing* use case, we need to indicate that the **data from a *Topic* in the publication side shall be routed to the same topic in the subscription side**. The *TopicRoute* is the entity that allows you to define these data flows for the forwarded data.

A *TopicRoute* is a data processing unit composed of the DDS *Inputs* and *Outputs* that receive and send the data, respectively. Hence a *TopicRoute* effectively represents the establishment of a *route* that data streams follow. Data from the publication side is forwarded to the subscription side.

In our example we just define a *TopicRoute* with a single *Input*–containing a *DataReader*– and a single *Output*–containing *DataWriter*–.

```
<topic_route name="RouteSquare">
    <input participant="domain0">
        <topic_name>Square</topic_name>
        <registered_type_name>ShapeType</registered_type_name>
```

(continues on next page)

(continued from previous page)

```
    </input>
    <output participant="domain1">
         <topic_name>Square</topic_name>
         <registered_type_name>ShapeType</registered_type_name>
    </output>
</topic_route>
```

Notice how the *Input* and *Output* are *attached* to a concrete *DomainParticipant* using the the `participant` attribute. The value of this attribute is the name of one the participant configurations defined in the parent *DomainRoute*. This is how you indicate to which domain the *Input* and *Output* are connected to–or from which *DomainParticipant* the *DataReader* and *DataWriter* are created, respectively–.

In our example, the *Input* is attached to the participant configuration with name `domain0` for domain 0, whereas the *Output* is attached to `domain1` for domain 1.

Additionally, for each *Input* and *Output* we need to specify at least two elements:

- **Name of their associated** ***Topic***. This indicates the name of the topic for which the *DataReader* and *DataWriter* are created. In this example, both entities are created from the `Square` topic.
- **Registered name of the type** associated with the *Topic*. This is the name used to identify the type of the user-data samples that are read and written by the input *DataReader* and output *DataWriter*. *Routing Service* needs to obtain the information prior to create the *DataReader* and *DataWriter*. There are two provide the type information:
  - Manually by defining the type in XML
  - Through discovery from any of the *DomainParticipant* within the parent *DomainRoute*. This is the mechanism our example relies to get the type and in this case the type is identified by the registered name `ShapeType` (you can find the the definition of this type in `[NDDSHOME]/resource/idl/ShapeType.idl`)

  You can learn more about type registration and how to configure it in *Specifying Types*.

For this case of routing a *Topic*, both the **input and output topic its associated type are the same**. This is often the situation when you want to simply route data across domains for system integration and scalability. Nevertheless, *Routing Service* is flexible to allow using different topics and types. In that case you will need to *plug* custom code to perform the routing. *Controlling Data: Processing Data Streams* addresses this use case.

Figure 2.5 shows the *TopicRoute* resource model.

See also:

TopicRoute **configuration in** ***Route*** Reference for the XML configuration of the *TopicRoute* element.

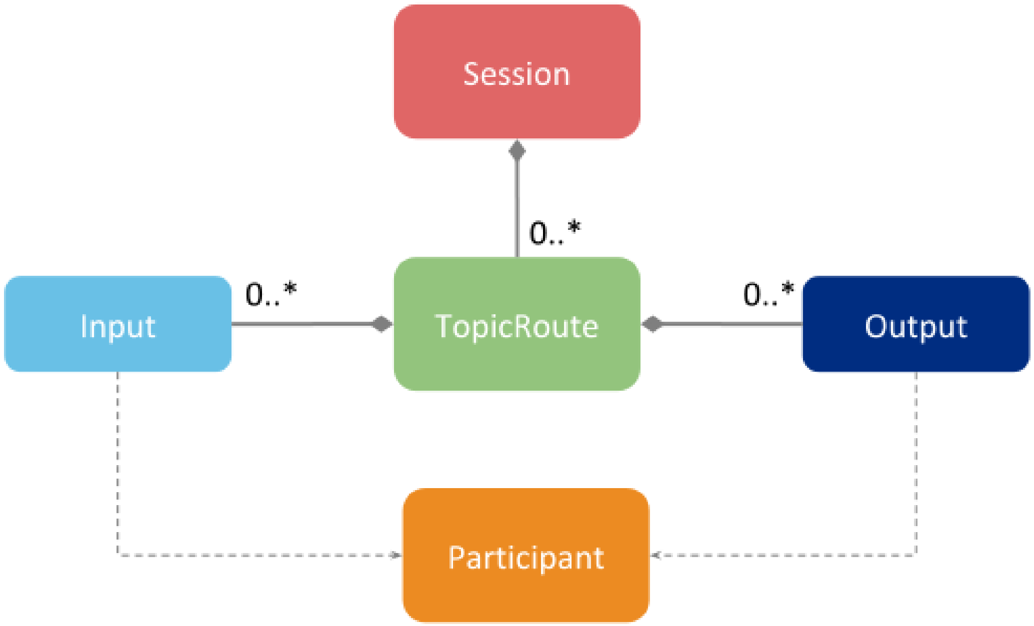


Figure 2.5: *TopicRoute* resource model

2.2 Routing a group of *Topics*

In section *Routing a Topic between two different domains* we learned how to route a specific *Topic*. We showed how to create a dedicated *TopicRoute* to forward the data for a concrete *Topic*. You can replicate this process for each *Topic* you want to route.

However, this process may become repetitive and in some cases avoidable. When such is the case, you can use the *AutoTopicRoute* to **automate the routing for a group of** *Topics*. An *AutoTopicRoute* allows you to specify a set of *potential TopicRoutes* that *Routing Service* will create on-demand upon dynamic *discovery* of the *Topic* to be routed.

Figure 2.6 shows the concept of the *AutoTopicRoute*. An *AutoTopicRoute* specifies a *regular expression* that is applied upon the discovery of any new *Topic*. The *AutoTopicRoute* creates a new *TopicRoute* for each newly discovered *Topic* whose name matches with the *AutoTopicRoute*'s expression.

An *AutoTopicRoute* allows defining a set of potential *TopicRoutes* that have a single *Input* and a single *Output*, both tied to their corresponding domain. A regular expression can be specified separately for publication and subscription *Topics*. Hence, when the *AutoTopicRoute* matches either with a publication or subscription *Topic*, it will create a *TopicRoute* to route the matched *Topic*.

Let's first run *Example: Routing All Data from One Domain to Another* to see this functionality. This example shows how to configure a *Routing Service* to route *all* the *Topics* from domain 0 to domain 1 using an *AutoTopicRoute*. To accomplish that, we have defined the *AutoTopicRoute* as follows:

```
<auto_topic_route name="RouteAll">
    <publish_with_original_info>true</publish_with_original_info>
    <input participant="domain1">
        <allow_topic_name_filter>*</allow_topic_name_filter>
```

(continues on next page)

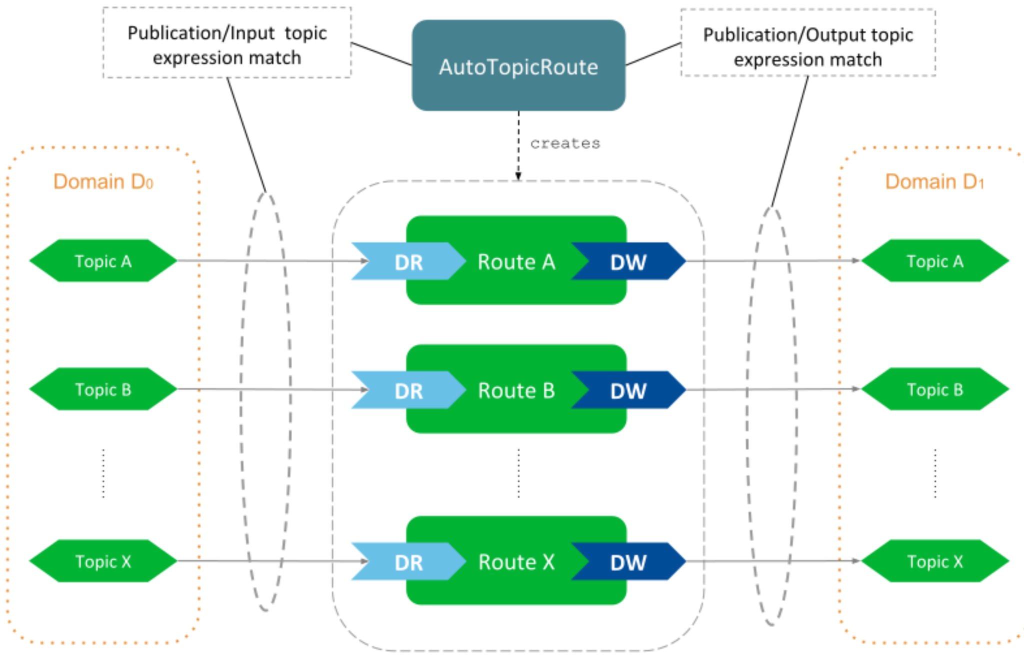


Figure 2.6: *AutoTopicRoute* concept

(continued from previous page)

```
        <allow_registered_type_name_filter>*</allow_registered_type_name_
↪filter>
        <!--
             Exclude RTI monitoring, administration and logging
             topics. They all start with 'rti/'
         -->
        <deny_topic_name_filter>rti/*</deny_topic_name_filter>
    </input>
    <output participant="domain2">
        <allow_topic_name_filter>*</allow_topic_name_filter>
        <allow_registered_type_name_filter>*</allow_registered_type_name_
↪filter>
        <!--
             Exclude RTI monitoring, administration and logging
             topics. They all start with 'rti/'
        -->
        <deny_topic_name_filter>rti/*</deny_topic_name_filter>
    </output>
</auto_topic_route>
```

The configuration of the *AutoTopicRoute* is such that matches the name and registered type name of every *Topic* on either `domain1` or `domain2`, *except* the *Topics* whose name starts with `rti/`.

An *AutoTopicRoute* allows you to specify two sets of regular expressions for both the input and output of the potential *TopicRoutes*:

- `allow_topic_name_filter` and `allow_registered_type_name_filter` specify the set of *Topic* names and types that are accepted for the dynamic creation of *TopicRoutes*. If **both expressions evaluate as true**, a new *TopicRoute* will be created, unless one of the deny filter evaluates as true.

- `deny_topic_name_filter` and `deny_registered_type_name_filter` specify the set of *Topic* names and types for which the creation of *TopicRoutes* is denied. If **any of the expressions evaluate as true**, the creation of the *TopicRoute* will be rejected. These expressions are evaluated after the allow filters, and only if these evaluated as true.

The configuration for the input and output of the *AutoTopicRoute* can contain a *DataReader* and *DataWriter* QoS respectively. You can leverage the concept of **QoS topic filters** to use a different QoS profile based on the name of the matched *Topic* (See *Applying topic filters to DDS Inputs and Outputs*).

You can also observe from the example that the *AutoTopicRoute* is defined under a *Session*. This means that all the created *TopicRoutes* will run under that context. Figure 2.7 shows the *AutoTopicRoute* resource model.

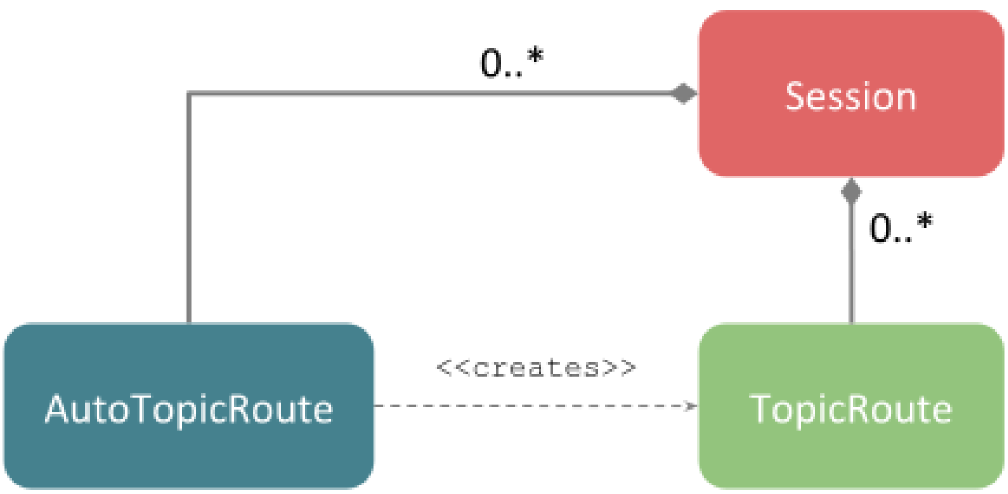


Figure 2.7: *AutoTopicRoute* resource model

See also:

TopicRoute* configuration in *Auto Route Reference for the XML configuration of the *AutoTopicRoute* element.

2.3 Using custom QoS Profiles

In the previous sections, we showed scenarios in which all the DDS entities of *Routing Service* are created with default QoS. That is, all the QoS policies are set with the initial default values as specified in the *Connext* documentation (see QoS Reference Guide).

For the majority of the cases though, you may want to specify your custom QoS values for the DDS entities of *Routing Service*. You can easily do that in XML by defining your QoS Profiles and **inherit from them** when specifying the configuration of QoS for each DDS entity.

Let's take a look to each step individually.

2.3.1 Defining a QoS Library

You can define XML QoS profiles for *Routing Service* the same way you can do it for a regular *Connext* application. You can define QoS libraries containing profiles directly under the `<dds>` root element. For example:

```
<dds>
    <qos_library name="MyQosLibrary">
         <qos_profile name="MyQoSProfile">
             <domain_participant_qos>
                 ...
             </domain_participant_qos>

             <subscriber_qos>
                 ...
             </subscriber_qos>

             <publisher_qos>
                 ...
             </publisher_qos>

             <datareader_qos>
                 ...
             </datareader_qos>

             <datawriter_qos>
                 ...
             </datawriter_qos>
        </qos_profile>
    </qos_library>
</dds>
```

As we will see shortly in the next step, within the *Routing Service* configuration you can reference these profiles in order to configure the corresponding underlying DDS entities.

You can define as many QoS libraries as you want, each with multiple profiles. Additionally, the definition of QoS libraries can appear either in the same file that contains the *Routing Service* configuration or in a separate one. For information on how to configure QoS in XML, see Configuring QoS with XML in the RTI Connext User's Manual.

See also:

Loading XML configurations in *Configuring RTI Services* How lo load XML configurations in *Routing Service*.

2.3.2 Specifying QoS for DDS entities

You can configure the QoS for each DDS entity that *Routing Service* creates. To accomplish this, each *Routing Service* entity that creates an underlying DDS entity provides a corresponding tag to specify its QoS.

For example, to configure the QoS for the *DomainParticipants* of a *DomainRoute*, you can specify a <domain_participant_qos> tag as follows:

```xml
<domain_route name="DomainRoute">

    <participant name="domain0">
        <domain_participant_qos base_name="MyQosLibrary::MyQosProfile">
            <!-- You can override inline any value -->
            ...
        </domain_participant_qos>
        ...
    </participant>

    ...
```

The QoS tag can have a base_name attribute to inherit from any available QoS profile, including builtin QoS profiles. Additionally, inline values for QoS policies can be specified in order to override default values or set by the base profile.

Table 2.1 shows the a list of *Routing Service* entities and the DDS entities they create, along with the tags that configure them.

Table 2.1: Configuration of the *Routing Service*'s underlying DDS entities.

Routing Service Entity	DDS Entity	QoS tag
DomainRoute	*DomainParticipant*	<domain_participant_qos> Example: <domain_route> <participant> <domain_participant_qos base_name="...">
Session	*Publisher*	<publisher_qos> Example: <session> <publisher_qos base_name="...">
	Subscriber	<subscriber_qos> Example: <session> <subscriber_qos base_name="...">

continues on next page

Table 2.1 – continued from previous page

Routing Service Entity	DDS Entity	QoS tag
TopicRoute's *Input* or *AutoTopicRoute*'s *Input*	*DataReader*	`<datareader_qos>` Example: `<topic_route> <input> <datareader_qos base_name= "...">`
TopicRoute's *Output* or *AutoTopicRoute*'s *Output*	*DataWriter*	`<datawriter_qos>` Example: `<topic_route> <output> <datawriter_qos base_name= "...">`

2.3.3 Applying topic filters to DDS *Inputs* and *Outputs*

You can leverage the concept of topic filters to select a QoS for a DDS *Input*'s *DataReader* and *Output*'s *DataWriter*. You simply need to define a QoS profile containing top-level QoS with a topic filter each, and then inherit from this profile when you specify the QoS for the input *DataReader* and output *DataWriter*. *Routing Service* will select the appropriate QoS when it creates the *DataReader* and *DataWriter* based on the name of their associated *Topic*.

For example, consider a system where there are three types of *Topic* categories: user data, monitoring, and administration. Each category has different QoS requirements. You could define a QoS Profile that contains three different *DataReader* QoS configurations, one for each category:

```
<qos_library name="MyQosLibrary">
     <qos_profile name="MyQoSProfileWithFilters">

         <datareader_qos topic_filter="UserData_*"> ...  </datareader_qos>

         <datareader_qos topic_filter="Monitoring_*"> ...  </datareader_qos>

         <datareader_qos topic_filter="Admin_*"> ...  </datareader_qos>

         <!-- Same idea for the datawriter_qos -->
         ...
    </qos_profile>
</qos_library>
```

Then you can define an *AutoTopicRoute* to route all the *Topics* in the system by simply indicating that the input *DataReader* shall be created using with the QoS obtained from our profile:

```
<auto_topic_route name="RouteAll">
    <input participant="domain0">
         <datareader_qos base_name="MyQosLibrary::MyQoSProfileWithFilters">
```

(continues on next page)

(continued from previous page)

```
    </input>
    <output participant="domain1">
        <datawriter_qos base_name="MyQosLibrary::MyQoSProfileWithFilters">
    </output>
</auto_topic_route>
```

When the *AutoTopicRoute* creates a *TopicRoute* for a matching publication or subscription *Topic*, the QoS for the *TopicRoute*'s input and output is resolved by matching the topic filter against the *Topic* name.

The topic filter is applied at the time the *AutoTopicRoute* matches with a publication or subscription *Topic*, so the right topic name can be used to match against the topic filter. The selected QoS will be used to create the input *DataReader* and output *DataWriter* of the generated *TopicRoute*.

2.4 Traversing Wide Area Networks

In the previous sections we learned to how to route *Topics* between domains, understanding the steps required to join the domains, and defining the *TopicRoutes* or *AutoTopicRoutes* to route the data. In this section, we will focus on routing data between domains separated geographically.

Many systems today have the need to communicate over Wide Area Networks (WAN). This may be the case to connect systems separated geographically. More importantly, it may be the case to provide system connectivity to and within the *cloud*. Access to data centers is often common when there's a requirement for data analytics.

You can use *Routing Service* to provide WAN connectivity between sub-systems composed of multiple applications communicating over a Local Area Network (LAN). This architecture allows you to scale the global system efficiently creating multiple databus layers dispersed over the WAN. Figure 2.8 shows this use case.

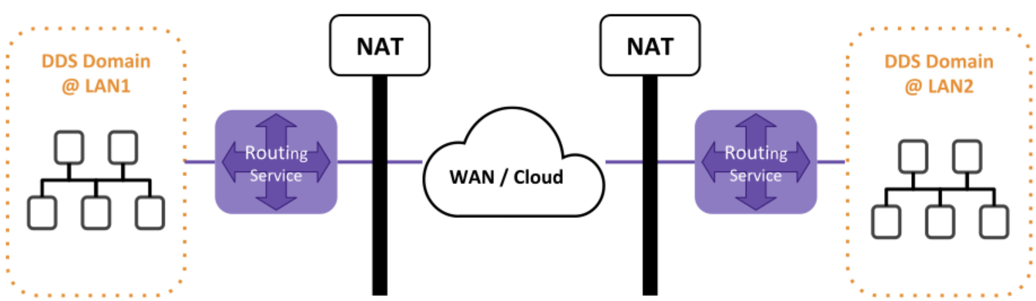


Figure 2.8: WAN traversal with *Routing Service*.

Routing Service can act as an entiry/exit *gateway* to provide connectivity to a WAN or cloud-based data center. The applications running in a LAN only need to know how to reach their gateway *Routing Service*. **Only the gateway services need to know to contact each other**, and they shall be **publicly accessible through the WAN**. This model simplifies the network configuration under presence of NATs/Firewalls, since they just need to be configured to forward the traffic only between the gateway *Routing Service*.

You can benefit from this architecture by configuring *Routing Service* to use a *WAN-enabled Transport* to provide communication outside of the private LAN or shared memory network. Figure 2.9 illustrates this setup.

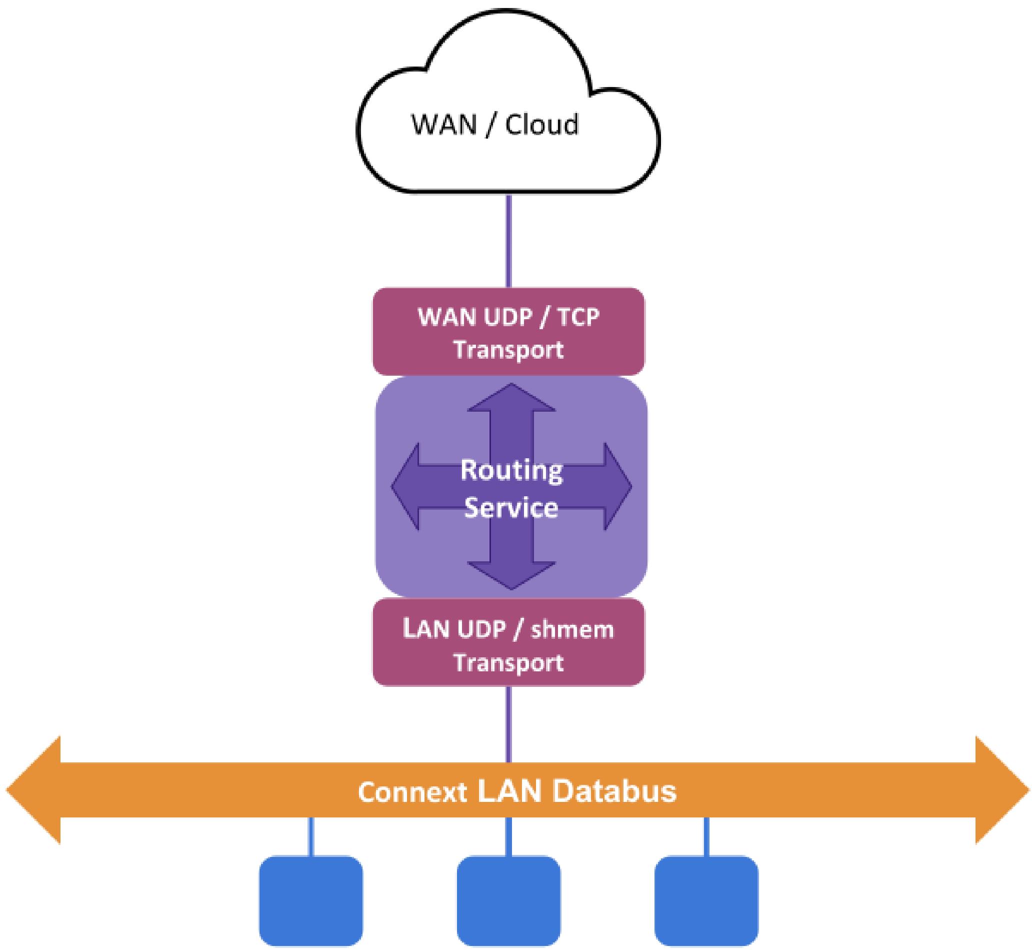


Figure 2.9: *Routing Service* as WAN/Cloud gateway

We will demonstrate how this is possible through the *Example: WAN Connectivity using the TCP transport*. This example will help you understand how *Routing Service* can route *Topics* between two geographically separated DDS domains comprised of a set of *Connext* applications connected in a LAN. The example scenario is shown in Figure 2.10.

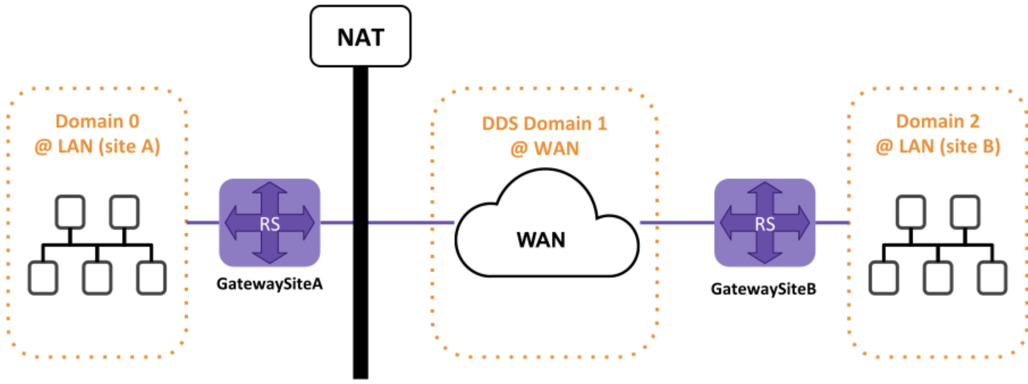


Figure 2.10: Example using the TCP transport to traverse WAN

First run the example to see the communication flowing between the *RoutingServices*. You can run all the steps in the same machine for a quicker setup. Let's go through the steps to configure the gateway *Routing Service*.

Note: For better understanding of this section, we recommend you get familiar with the RTI TCP Transport.

2.4.1 Define a QoS profile that configures the RTI TCP transport

The configuration of the transport is done through the Property QoS for the *DomainParticipant*. It requires specifying a set of properties to load the transport library (if it's an external transport plugin) and specific values to configure its behavior. To avoid repeating the same configuration for each participant in *Routing Service*, we define a base profile with all the common properties:

```
<qos_library name="QosLib">
    <qos_profile name="TcpWanProfile">
        <!--
            We define here the common properties to configure the␣
↪TCP transport,
            which includes mostly the loading of the transport␣
↪implementation library.
            Specific values for public address and port are set␣
↪appropriate on each
            Routing Service.
        -->
        <domain_participant_qos>
            <transport_builtin>
                <mask>MASK_NONE</mask>
            </transport_builtin>
            <property>
```

(continues on next page)

(continued from previous page)

```
                <value>
                    <element>
                        <name>dds.transport.load_plugins</name>
                        <value>dds.transport.TCPv4.tcp1</value>
                    </element>
                    <element>
                        <name>dds.transport.TCPv4.tcp1.library</name>
                        <value>nddstransporttcp</value>
                    </element>
                    <element>
                        <name>dds.transport.TCPv4.tcp1.create_
↪function</name>
                        <value>NDDS_Transport_TCPv4_create</value>
                    </element>
                    <element>
                        <name>dds.transport.TCPv4.tcp1.parent.classid
↪</name>
                        <value>NDDS_TRANSPORT_CLASSID_TCPV4_WAN</
↪value>
                    </element>
                    <element>
                        <name>dds.transport.TCPv4.tcp1.public_address
↪</name>
                        <value>$(PUBLIC_ADDRESS)</value>
                    </element>
                    <element>
                        <name>dds.transport.TCPv4.tcp1.server_bind_
↪port</name>
                        <value>$(BIND_PORT)</value>
                    </element>
                    <element>
                        <name>dds.transport.TCPv4.tcp1.disable_nagle
↪</name>
                        <value>1</value>
                    </element>
                </value>
            </property>
            <discovery>
                <initial_peers>
                    <element>$(REMOTE_RS_PEER)</element>
                </initial_peers>
            </discovery>
        </domain_participant_qos>
    </qos_profile>
</qos_library>
```

In addition to the transport configuration, the profile also sets the value for the *initial peers* required for the *DomainParticipant* of the *Routing Service* **to reach the peer remote gateway**.

For the definition of this profile, we're leveraging the *XML configuration variables* to reduce even more code duplication. Namely, we define the following variables that are set accordingly when running each *Routing Service*:

- `PUBLIC_ADDRESS`: the public IP address and public port where the *Routing Service* is reachable.
- `BIND_PORT`: the host port that the TCP connection of the *Routing Service* is bound to. This value is important to create a port forwarding rule between public port and host port in the NAT configuration.
- `REMOTE_RS_PEER`: shall contain the discovery peer of the remote *Routing Service* to communicate over the WAN. In this example, the remote peer is the public address and public port of the *Routing Service* gateway for the remote site. This value is used as the *initial peers* of the *DomainParticipant* that provides WAN connectivity. See discovery peer configuration for details on setting discovery peers.

See also:

Transport Plugins Documentation for the Connext Transport Plugin conncept.

RTI TCP Transport properties Available configuration properties for the RTI TCP Transport.

2.4.2 Specify the domains to join and which transport to use

This is the key step that makes possible to forward data from a DDS application in a LAN to the WAN. The main idea is to define two different *DomainParticipants* to provide access to the different networks. Figure 2.11 shows the entity model of the *DomainRoute* with its two *DomainParticipants*, each using a different underlying transport to communicate with different networks.

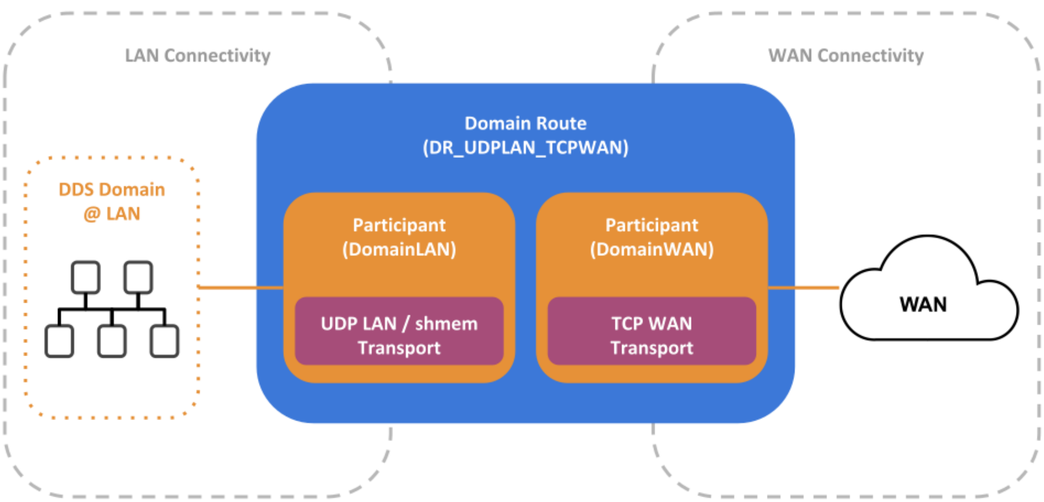


Figure 2.11: Configuration of the *DomainRoute* to forward data over the WAN

The `DomainLAN` *DomainParticipant* is configured to join domain 0 and use the default UDPv4 LAN and shared memory transports to communicate with the applications on the site A LAN. Alternatively, The `DomainWAN` *DomainParticipant* is configured to join domain 1 and use the *RTI TCP Transport* to communicate over the WAN. `DomainWAN` is the *gateway DomainParticipant* that **communicates with the remote *Routing Service* gateway at a different location**.

The definition of these participants appear in a *DomainRoute* as follows:

```xml
<domain_route name="DR_UDPLAN_TCPWAN">
    <!--
        With default participant QoS, which uses UDP LAN and Shared␣
↪memory
        as trasnports
    -->
    <participant name="DomainLAN">
        <domain_id>0</domain_id>
    </participant>

    <participant name="DomainWAN">
        <domain_id>1</domain_id>
        <!--
            With participant QoS configured to use the TCP transport.
↪ Requires
            setting the variableS PUBLIC_ADDRESS AND BIND_PORT to␣
↪the actual
            values used in to route the traffic to this RS.
        -->
        <domain_participant_qos base_name="QosLib::TcpWanProfile"/>
    </participant>
</domain_route>
```

You can observe how the `DomainWAN` participant is configured with a QoS that inherits from the `QosLib::TcpWanProfile`, which configures the RTI TCP transport, in addition to other discovery settings. The QoS for this participant provides two additional transport properties to configure the TCP server public address and bind port.

2.4.3 Specify the *Topics* to be routed

In this example we want to route all the topics between the LAN domains, and we want the communication to be bidirectional. We'll do this by defining two *AutoTopicRoutes* to forward any *Topic* for a different communication direction each. We'll place both under a single *Session* configured with default settings:

```xml
<session name="Session">
    <auto_topic_route name="FromLANtoWAN">
        <input participant="DomainLAN">
            <deny_topic_name_filter>rti/*</deny_topic_name_filter>
        </input>
        <output participant="DomainWAN">
            <deny_topic_name_filter>rti/*</deny_topic_name_filter>
        </output>
    </auto_topic_route>

    <auto_topic_route name="FromWANtoLAN">
        <input participant="DomainWAN">
            <deny_topic_name_filter>rti/*</deny_topic_name_filter>
        </input>
        <output participant="DomainLAN">
            <deny_topic_name_filter>rti/*</deny_topic_name_filter>
```

(continues on next page)

(continued from previous page)

```
        </output>
    </auto_topic_route>
</session>
```

AutoTopicRoute `FromLANtoWAN` is configured to forward any *Topic* coming from the LAN domain to the WAN domain. `FromWANtoLAN` *AutoTopicRoute* is configured to forward any *Topic* coming from the WAN domain–which connects to the remote LAN Domain–to the local LAN domain.

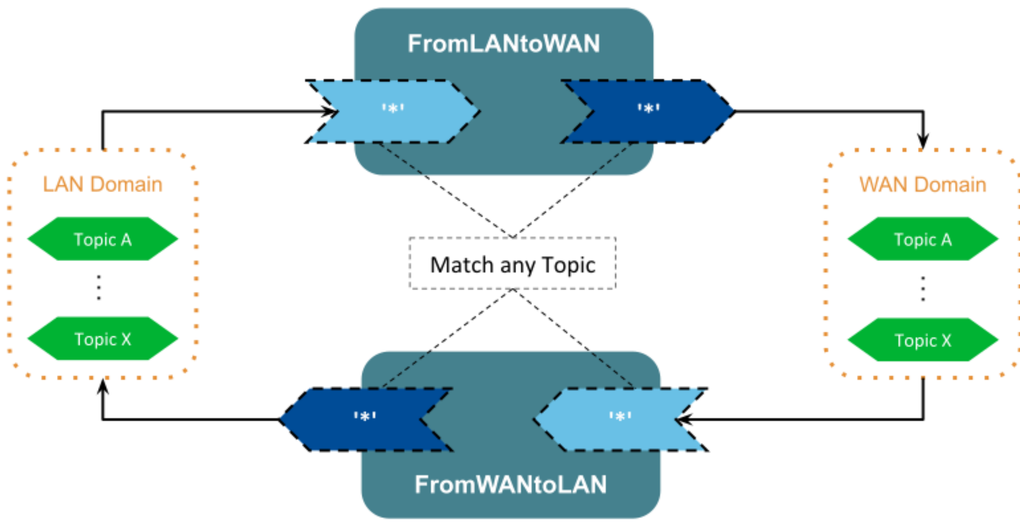


Figure 2.12: Definition of *AutoTopicRoute* to forward topics bidirectionally

Figure 2.12 illustrates the definition of the *AutoTopicRoutes* to forward all topics between the LAN and WAN domains. Each *AutoTopicRoute* is configured with both input and output filters to match any *Topic*. The difference between the *AutoTopicRoutes* is simply the domain assigned to the *Input* and *Output*–the *DomainParticipant* from which the input *DataReader* and output *DataWriter* will be created–.

2.5 Key Terms

Forwarding Process The action of routing data from input to output.

Entity Configuration Name Name assigned to uniquely identify an entity. Specified by the attribute `name`.

Publication Side Side of the communication from where *Routing Service* inputs receive data.

Subscription Side Side of the communication t0 where *Routing Service* outputs write data.

Resource model A model to represent *Routing Service* entities viewed as resources and their relationships.

DomainRoute A collection of *DomainParticipants*.

Session The threading context where the forwarding process takes place.

TopicRoute Processing unit for data streams. Composed of multiple *Inputs* and *Outputs*.

AutoTopicRoute Factory of *TopicRoutes* based on topic name regular expression matching.

Input Entity that reads data from a specific domain. For DDS domains, it contains an underlying *DataReader*.

Output Entity that that writes to a specific domain.For DDS domains, it contains an underlying *DataWriter*.

Transport Internal component of a *DomainParticipant* that provides connectivity to a concrete network technology.

Discovery Peer A DDS address that identifies a remote application.

Chapter 3

Controlling Data: Processing Data Streams

In chapter *Routing Data: Connecting and Scaling Systems* we presented how *Routing Service* can easily connect and scale systems. In order to do so, data is *forwarded* among systems, thus generating data streams flowing from one system to another. The forwarding process is a basic operation that consists of propagating data streams from the input to the output.

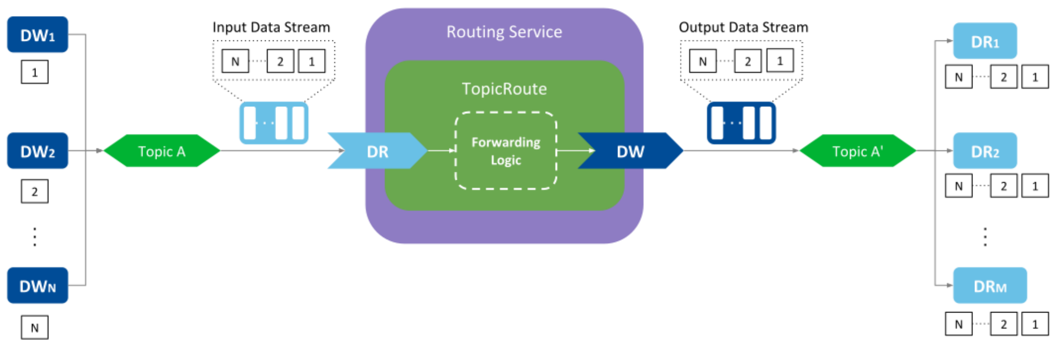


Figure 3.1: Basic forwarding of an input data stream

Figure 3.1 illustrates the forwarding process of *Topic* data. At the publication side, there are `N` *DataWriters* each producing samples for `Topic A`. The *Routing Service* has a *TopicRoute* with a single input *DataReader* and a single output *DataWriter*. At the subscription side there are `M` *DataReaders* all receiving samples from topic `Topic A'`. All the samples the user *DataWriters* produce in the publication side are received by the input *DataReader*, which are then forwarded through the output *DataWriter* **to all the user *DataReaders* in the subscription side**. You can observe that the *TopicRoute* has a component the performs the forwarding logic that involves reading from the input *DataReader* and writing to the output *DataReader*.

The forwarding logic in the *TopicRoute* may be limiting when system connectivity demands other requirements beyond basic data forwarding. You can anticipate the simple read-and-write logic may be inadequate in *TopicRoutes* that define multiple INPUTs| and *Outputs* and the types of the associated *Topics* are different. These cases require the use of a custom logic to process the data streams, and this is the task of the *Processor*. Figure 3.2 shows the concept.

A *Processor* is a *pluggable* component that allows you **control the forwarding process of a** *TopicRoute*. You can create your own *Processor* implementations based on the needs of your system integration, defining your own data flows, and processing the data streams at your convenience.

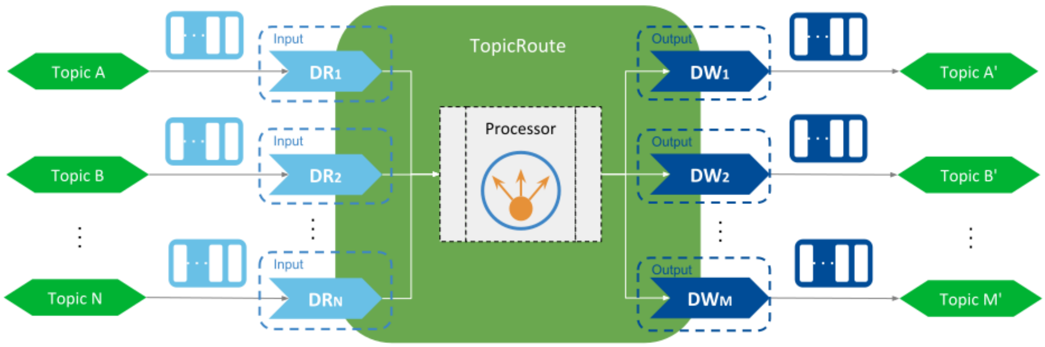


Figure 3.2: *Processor* concept

A *Processor* receives notifications from the *TopicRoute* about relevant events such as the availability of *Inputs* and *Outputs*, state of the *TopicRoute*, or arrival of data. Then a *Processor* can react to any of these events and perform whichever necessary actions. The basic forward logic presented above is actually a builtin *Processor* implementation and that is set as the default in all the *TopicRoutes*.

The following sections will guide you through the process of creating your own *Processor*, how to configure it and install it in *Routing Service*. We will show you this functionality with examples of *Aggregation* and *Splitting* patterns.

Note: All the following sections require you to be familiar with the routing concepts explained in section *Routing Data: Connecting and Scaling Systems*. Also this section requires software programming knowledge in C/C++, understanding of CMake, and shared library management.

See also:

Forwarding Processor Details on the default forwarding *Processor* of the *TopicRoutes*.

3.1 DynamicData as a Data Representation Model

The nature of the architecture of *Routing Service* makes it possible to work with data streams of different *types*. This demands a strategy for dealing with all the possible times both a compilation and run time. This is provided through *DynamicData*.

DynamicData a generic container that holds data for any type by keeping a custom and flexible internal representation of the data. *DynamicData* is a feature specific from *Connext* and is part of the core libraries. Figure 3.3 shows the concept of *DynamicData*.

DynamicData is a container-like structure that holds data of any type. The description of how that type looks like is given by the *TypeCode*, a structure that allows representing any type. Using the *TypeCode* information, a *DynamicData* object can then contain data for the associated type and behave as if it was an actual structure of such type. The *DynamicData* class has a rich interface to access the data members and manipulate its content.

The *Processor* API makes the inputs and outputs to interface with *DynamicData*. Hence the inputs will return a list of *DynamicData* samples when reading, while the outputs expect a *DynamicData* object on the write

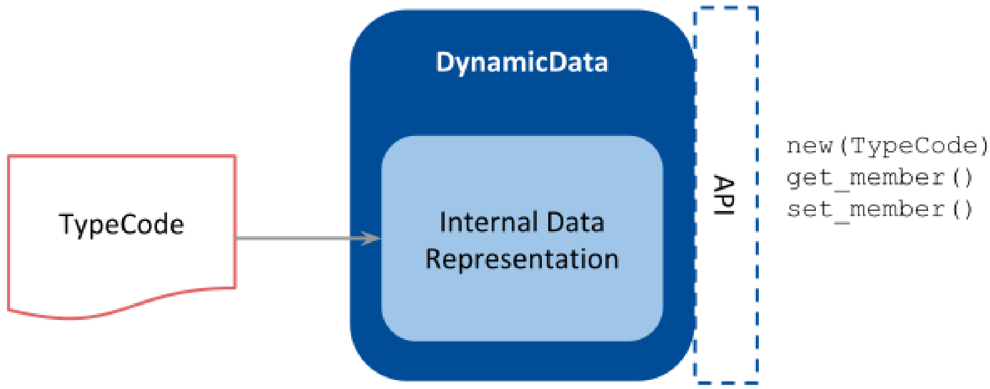


Figure 3.3: *DynamicData* concept

operation. This common representation model has two benefits:

- It allows implementations to work without knowing before hand the types. This is very convenient for general purpose processors, such as data member mappers.
- It allows implementations to work independently from the the data domain where the data streams flow. This is particularly important when a different data other than DDS is used through a custom *Adapter* (*Data Integration: Combining Different Data Domains*).

See also:

Objects of Dynamically Defined Types. Section in *RTI Connext* User's manual about *DynamicData* and *TypeCode*.

DynamicData C++ API reference Online API documentation for the DynamicData class.

3.2 Aggregating Data From Different Topics

A very common scenario involves defining routing paths to combine data from two or more input *Topics* into a single output *Topic*. This pattern is known as *Topic aggregation*. You can leverage the *Processor* component to perform the custom *Topic* aggregation that best suits the needs of your system.

An example of *Topic* aggregation is shown in Figure 3.4. There are two input *Topics*, `Square` and `Circle`, and a single output *Topic*, `Triangle`. All *Topics* have the same type `ShapeType`. The goal is to produce output samples by combining data samples from the two inputs.

Let's review all the tasks you need to do to create a custom *Processor* using the *Example: Using a Shapes Processor* . You can run it first to see it in action but you can also run one step at a time as we explain each.

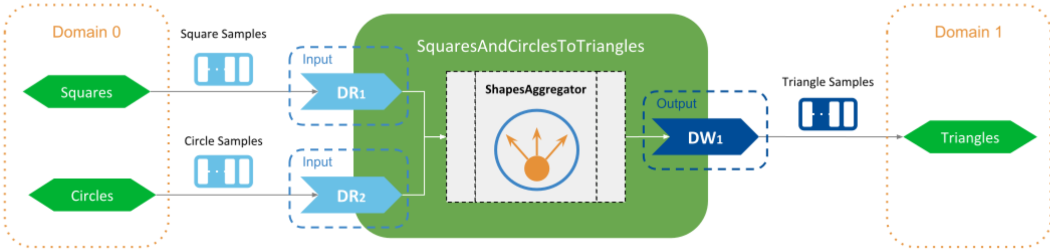


Figure 3.4: Aggregation example of two *Topics*

3.2.1 Develop a Custom *Processor*

Once you know the stream processing pattern you want perform, including what your data inputs and outputs are, you can then write the custom code of the *Processor*. A custom processor must implement the interface `rti::routing::Processor`, which defines the abstract operations that the *TopicRoute* calls upon occurrence of certain events.

In our example, we create a `ShapesAggregator` class to be our *Processor* implementation:

```
class ShapesAggregator : public rti::routing::processor::NoOpProcessor {

    void on_data_available(rti::routing::processor::Route &);

    void on_output_enabled(
        rti::routing::processor::Route &route,
        rti::routing::processor::Output &output);

    ...
}
```

Note how the processor class inherits from `NoOpProcessor`. This class inherits from `rti::routing::processor::Processor` and implements all its virtual methods as no-op. This is a convenience that allows us to implement only the methods for the notification of interest. In this example:

- `on_output_enabled`: Notifies that an output has been enabled and it is available to write. In our example, we create a buffer of the output type (`ShapeType`) that will hold the aggregated content of the input samples.
- `on_data_available`: Indicates that at least one input has data available to read. In our example, this is where the aggregation logic takes place and it will simply generate aggregated output samples that contain the same values as the `Square` samples, except for the field `y`, which is obtained from the `Circle`.

See also:

Processor C++ API reference

Route States Different states of a *TopicRoute* and which *Processor* notifications are triggered under each of them.

3.2.2 Create a Shared Library

Once the *Processor* implementation is finished we need to compile it and generate a *shared library* that *Routing Service* can load. In this example we use `CMake` as the build system to create the shared library. We specify the generation of a library with name `shapesaggregator`:

```
...

add_library(shapesprocessor
        "${CMAKE_CURRENT_SOURCE_DIR}/ShapesProcessor.cxx")

...
```

The generated library contains the compiled code of our implementation, contained in a single file `ShapesAggregator.cxx`. A key aspect of the generated library is that it must export an external function that instantiates the `ShapesAggregator`, and it's the function that *Routing Service* will call to instantiate the *Processor*. This external symbol is denoted *entry point* and you can declare it as follows:

```
RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DECL(ShapesAggregatorPlugin);
```

The macro declares an external exported function with the following signature:

```
struct RTI_RoutingServiceProcessorPlugin*
ShapesAggregatorPlugin_create_processor_plugin(
        const struct RTI_RoutingServiceProperties *,
        RTI_RoutingServiceEnvironment *);
```

which is the signature *Routing Service* requires and will assume for the entry point to create a custom *Processor*. Note that the implementation of this function requires using the macro `RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DEF` in the source file.

3.2.3 Define a Configuration with the Aggregating *TopicRoute*

This is a similar process than the one we explained in section *Routing a Topic between two different domains*. There are two main differences that are particular to the use with a processor.

Configure a plugin library

Within the root element of the XML configuration, you can define a *plugin library* element that contains the description of all the plugins that can be loaded by *Routing Service*. In our case, we define a plugin library with a single entry for our aggregation processor plugin:

```
<plugin_library name="ShapesPluginLib">
    <processor_plugin name="ShapesProcessor">
        <dll>shapesaggregator</dll>
        <create_function>
            ShapesAggregatorPlugin_create_processor_plugin
        </create_function>
```

(continues on next page)

(continued from previous page)

```
    </processor_plugin>
</plugin_library>
```

The values specified for the `name` attributes can be set at your convenience and they shall uniquely identify a plugin instance. We will use these names later within the *TopicRoute* to refer to this plugin. For the definition of our processor plugin we have specified two elements:

- `dll`: The name of the shared library as we specified in the build step. We just provide the library name so *Routing Service* will try to load it from the working directory, or assume that the library path is set accordingly.
- `<create_function>`: Name of the entry point (external function) that creates the plugin object, exactly as we defined in code with the `RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DECL` macro.

Once we have the plugin defined in the library, we can move to the next step and define the *TopicRoute* with the desired routing paths and our *Processor* in it.

> **Warning:** When a name is specified in the <dll> element, *Routing Service* will automatically append a **d** suffix when running the debug version of *Routing Service*.

See also:

Plugins Documentation about the `<plugin_library>` element.

Plugin Management For in-depth understanding of plugins.

Configure a *Routing Service* with the custom routing paths

In this example we need to define a *TopicRoute* that contains the two *Inputs* to receive the data streams from the `Square` and `Circle` *Topics*, and the single output to write the single data stream to the `Triangle` *Topic*. The key element in our *TopicRoute* is the specification of a custom *Processor*, to indicate that the *TopicRoute* should **use an instance of our plugin to process the route's events and data**:

```
<topic_route name="SquaresAndCirclestoTriangles">
    <processor plugin_name="ShapesPluginLib::ShapesAggregator">
        ...
    </processor>
    <input name="Square" participant="domain0">
        <topic_name>Square</topic_name>
        <registered_type_name>ShapeType</registered_type_name>
        <datareader_qos base_name="RsShapesQosLib::RsShapesQosProfile"/>
    </input>
    <input name="Circle" participant="domain0">
        <topic_name>Circle</topic_name>
        <registered_type_name>ShapeType</registered_type_name>
        <datareader_qos base_name="RsShapesQosLib::RsShapesQosProfile"/>
    </input>
```

(continues on next page)

(continued from previous page)

```
    <output name="Triangle" participant="domain1">
        <topic_name>Triangle</topic_name>
        <registered_type_name>ShapeType</registered_type_name>
    </output>
</topic_route>
```

There are three important aspects in this *TopicRoute* configuration:

- The custom *Processor* is specified with the `<processor>` tag. The `plugin_name` attribute must contain the qualified name of an existing processor plugin within a plugin library. The qualified name is built using the values from the `name` attributes of the plugin library and plugin element. Although our example does not make use of it, you could provide run-time configuration properties to our plugin through an optional `<property>` tag. This element represents a set of name-value string pairs that are passed to the `create_processor` call of the plugin.
- *Input* and *Output* elements have all a `name` attribute. This is the configuration name for these elements can be used within the Processor to look up and individual *Input* or *Output* by its name, such as we do in our example. Also notice how the names match the *Topic* names for which they are associated. Because we are not specifying `<topic_name>` element, *Routing Service* uses the *Input* and *Output* names as *Topic* names. In our example this makes it convenient to identify 1:1 inputs and outputs with their topics.
- The input *DataReaders* are configured with a QoS that sets a `KEEP_LAST` history of just one sample. This allows our processor to just read and aggregate the latest available sample from each input.

3.3 Splitting Data From a single Topic

Another common pattern consists of defining routing paths to divide or *split* data from a input *Topic* into several output *Topics*. This mechanism represents the reverse equivalent to aggregation and is known as *Topic* splitting. You can leverage the *Processor* component to perform the *Topic* splitting that best suits the needs of your system.

An example of *Topic* splitting is shown in Figure 3.5. There is a single input *Topic*, `Squares`, and two output *Topics*, `Circles` and `Triangles`. All *Topics* have the same type `ShapeType`. The goal is to produce output samples by splitting the content of data samples from the input.

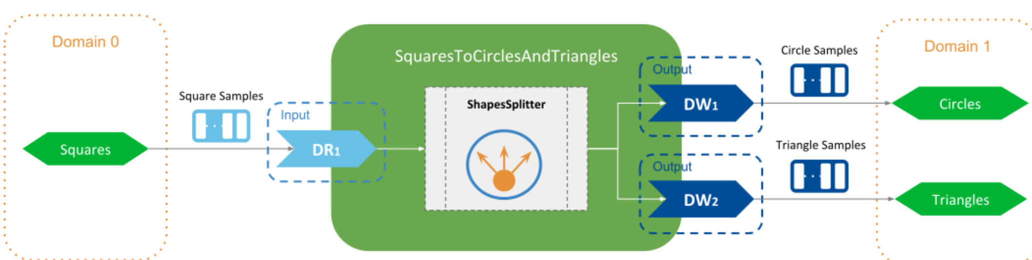


Figure 3.5: Splitting example of a *Topic*

The steps required to create a custom splitting *Processor* are the same as described in the previous section *Aggregating Data From Different Topics*. For this example we focus only in the aspects that are different.

3.3.1 Custom *Processor* implementation

In this example, we create a `ShapesSplitter` class to be our *Processor* implementation. Similar to `ShapesAggregator`, this class reacts only to two event notifications:

- `on_input_enabled`: Creates a sample buffer that will be used to contain the split content from the inputs. Because all the inputs and outputs have the same type (`ShapeType`), we can obtain use the input type to create the output sample.
- `on_data_available`: This is where the splitting logic takes place and it will simply generate split output samples that contain the same values as the `Square` samples for all fields except `x` and `y`, which are set as follows:
  - `Circle` output: the `x` field has the same value than the input `Square` and sets `y` to zero.
  - `Triangle` output: the `y` field has the same value than the input `Square` and sets `x` to zero.

3.3.2 Define a Configuration with the Splitting *TopicRoute*

In this example we need to define a *TopicRoute* that contains the single *Input* to receive the data streams from the `Square` *Topic*, and the two *Outputs* to write the data streams fro the `Circle` and `Triangle` *Topics*. The *TopicRoute* specifies a custom processor to be created from our plugin library, and it's configured to create the `SplitterProcessor`

```
<topic_route name="SquaresToCirclesAndTriangles">
    <processor plugin_name="ShapesPluginLib::ShapesSplitter"/>
    <input name="Square" participant="domain0">
        <registered_type_name>ShapeType</registered_type_name>
    </input>
    <output name="Circle" participant="domain1">
        <registered_type_name>ShapeType</registered_type_name>
    </output>
    <output name="Triangle" participant="domain1">
        <registered_type_name>ShapeType</registered_type_name>
    </output>
</topic_route>
```

In this *TopicRoute* configuration, the input *DataReader* and output *DataWriters* are create with default QoS. This is an important difference with regards to the configuration of aggregation example. The splitting pattern in this case is simpler since there's a single input and each received sample can hence be split individually.

Note that the splitting pattern can include multiple inputs if needed, and generate output samples based on more complex algorithms in which different content from different inputs is spread across the outputs.

3.4 Periodic and Delayed Action

Processors can react to certain events affecting *TopicRoutes*. One special event that requires attention is the *periodic* event. In addition to events of asynchronous nature such as data available or route running, a *TopicRoute* can be configured to also provide notifications occurring at a specified periodic rate.

Example below shows the XML that enables the periodic event notification at a rate of one second:

```
<topic_route>
    <periodic_action>
        <sec>1</sec>
        <nanosec>0</nanosec>
    </periodic_action>

    ...
</topic_route>
```

If a *TopicRoute* enables the periodic event, then your *Processor* can implement the `on_periodic_action` notification and perform any operation of interest, including reading and writing data. For details on the XML configuration for periodic action and *TopicRoutes* in general, see Section 8.2.6.

Note that each *TopicRoute* can specify a different period, allowing you to have different event timing for different routing paths. Similarly, the event period that can be modified at *runtime* through the `Route::set_period` operation that is available as part of the *Processor* API.

The configuration above will generate periodic action events **in addition to data available events coming from the inputs**. You could disable the notification of data available using the tag `<enable_data_on_inputs>`, causing the *TopicRoute* to be periodic-based only.

3.5 Simple data transformation: introduction to Transformation

There are cases involving basic manipulation of data streams that can be performed independently in a per-sample basis. That is, for a given input sample (or set of them) there's a *transformed* output sample (or set of them). For this particular use case, *Routing Service* defines the concept of *Transformation*, shown in Figure 3.6.

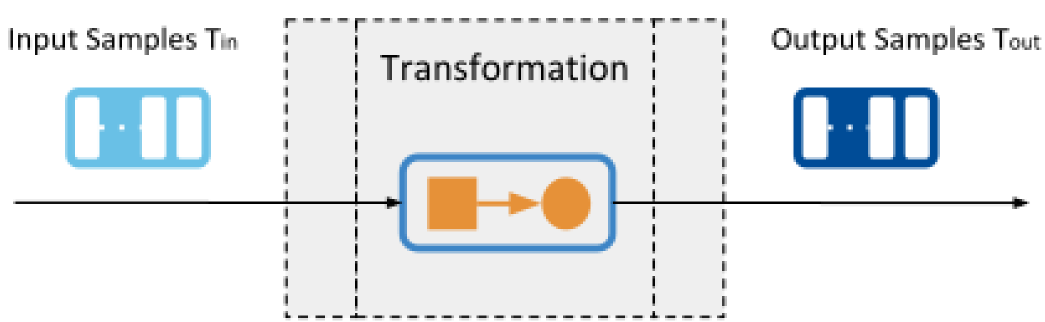


Figure 3.6: Transformation concept

A *Transformation* is a pluggable component that receives an input data stream of type T_{in} and produces an output data stream of type T_{out}. The relation between the number of input samples and output samples can also be different.

This component can be installed in two different entities in *Routing Service*. A *Transformation* can appear to process the data stream after is produced by an *Input DataReader* and/or to process a data stream before is passed to an *Output DataWriter*. Figure 3.7 shows the complete model and context of this component.

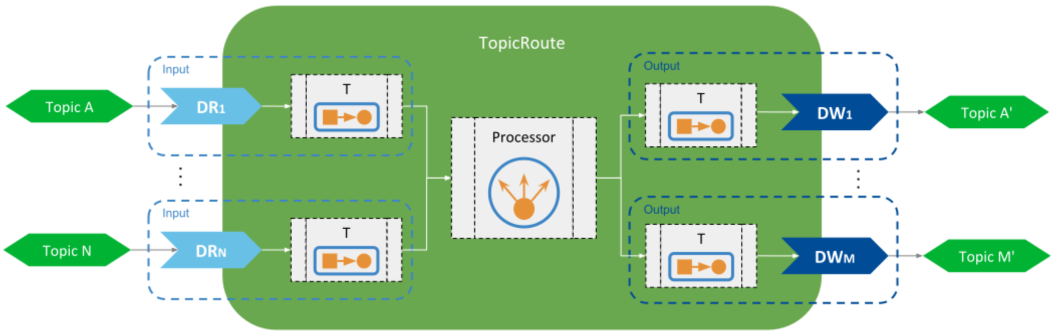


Figure 3.7: Transformation model and context

You can observe that a each *Input* and *Output* can contain a transformation. On the input side, the *Transformation* is applied to the data stream generated by the input *DataReader* and the result is fed to the *Processor*. Alternatively, on the output side the *Transformation* is applied to the data stream produced by the *Processor* and the result is passed to the output *DataWriter*.

When transformations are used it's a requirement that the type of the samples provided by the input *DataReader* is the same type T_{in} expected by the input *Transformation*. Similarly, the type $T_{out}$of the samples produced by the output *Transformation* must be the same than the type of the samples expected by the output *DataWriter*. As in with a *Processor*, a *Transformation* is expected to work with *DynamicData* (*DynamicData as a Data Representation Model*).

You can run *Example: Transforming the Data with a Custom Transformation* to see how a transformation can be used. In this example, the transformation implementation receives and generates samples of type `ShapeType`. The output samples are equal to the input samples except for the field `x`, which is adjusted to produce only two possible constant values.

3.5.1 Transformations vs Processors

A *Transformation* is fundamentally different than a *Processor*. Moreover, they complement each other. A *Transformation* can be seen a very simplified version of a *Processor* that has a single input and a single output and in which the input data stream is processed as it is read.

In general you will find yourself implementing a *Processor* to perform the data stream processing required by your system. Nevertheless there are cases where a *Transformation* is more suitable for certain patterns such as format conversion, content reduction, or normalization (see *What stream processing patterns can I perform?*).

3.6 What stream processing patterns can I perform?

With *Routing Service* you have the ability to define routing paths with multiple inputs and outputs, and provide custom processing logic for the data streams of those paths. This provides a great degree of flexibility that allows to perform pretty much any processing pattern.

In addition to the presented patterns of aggregation, splitting, and periodic action, there are other well-known patterns:

- **Data format conversion**: this is the case where the input samples are represented in one format and are converted to produce output samples of a different format. For example, input samples could be represented as a byte array and converted to a JSON string.
- **Content Enrichment**: an output sample is the result of amplifying or adding content to an input sample. For example, output samples can be enhanced to contain additional fields that represents result of computations performed on input sample fields (e.g., statistic metrics).
- **Content Reduction**: an output sample is the result of attenuating or removing content from an input sample. For example, output samples can have some fields removed that are not relevant to the final destination, to improve bandwidth usage.
- **Normalizer**: output samples are semantically equivalent to the input samples except the values are adjusted or normalized according to a specific convention. For example, input samples may contain a field indicating the temperature in Fahrenheit and the output samples provide the same temperature in Celsius.
- **Event-based or Triggered Forwarding**: output samples are generated based on the reception and content of concrete input samples in which some of them act as events indicating which, how, and when data is forwarded.

3.7 Key Terms

Data Stream The collection of samples received by a *TopicRoute*'s *Input* or written to a *TopicRoute*'s *Output*.

DynamicData A general purpose structure that contains data of any type.

Processor Pluggable component to control the forwarding process of a *TopicRoute*

Shared Library or Module An output artifact that contains the implementation of pluggable components that *Routing Service* can load at run-time.

Entry Point External symbol in a shared library that *Routing Service* calls to instantiate a custom plugin instance.

Stream Processing Patterns Processing algorithms applied to the data streams of a *TopicRoute*.

Periodic action *TopicRoute* event notification occurring at a configurable period.

Transformation Pluggable component perform modifications of a forwarded data stream.

Chapter 4

Data Integration: Combining Different Data Domains

In chapters *Routing Data: Connecting and Scaling Systems* and *Controlling Data: Processing Data Streams* we showed how *Routing Service* is a powerful solution to scale and aggregate DDS systems. You can define data flows between publication and subscription *Topics*, and also perform stream processing using a custom *Processor*.

Up to this point we have shown these capabilities only in the presence of DDS data sources and destinations. However, *Routing Service* can provide the same capabilities for any other data technology and protocol through the concept of an *Adapter*, which makes *Routing Service* a suitable framework for data integration.

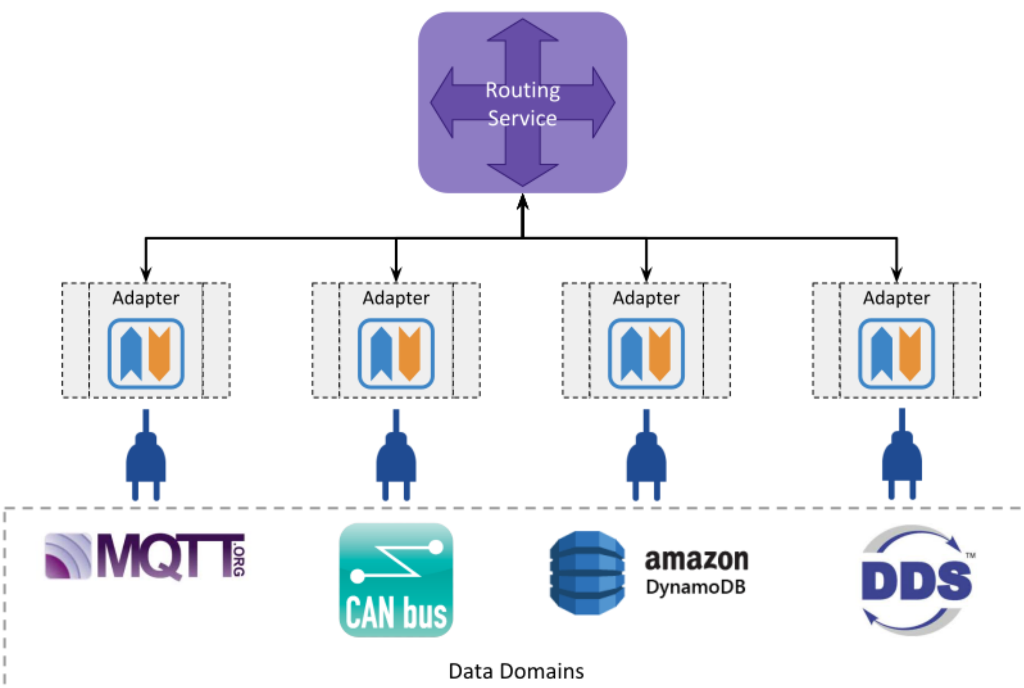


Figure 4.1: Data Integration in *Routing Service*

An *Adapter* is a *pluggable* component that allows you to access any data domain pertaining to any technology. *Adapters* provide a connection point to data domains so the information can flow back and forth to *Routing Service*. The main *Adapter* interfaces are:

- *Plugin*: Entry point to the custom implementation. It consists of a creation method that *Routing Service* can call to instantiate the *Adapter* implementation. (see *Plugin Management*).
- *Connection*: Entity responsible for *accessing* a concrete data domain. (see *Connection*). For example, a socket connection, database connection, or *DomainParticipant*. The *Connection* is the factory of *StreamReader* and *StreamWriter*.
- *StreamReader*: Entity responsible for *reading* data streams from a concrete data domain and with a single *Input*.
- *StreamWriter*: Entity responsible for *writing* data streams to a concrete data domain and associated with a single *Output*.

Figure 4.2 illustrates the concept of the *Adapter* and how it fits within the *Routing Service* entity model.

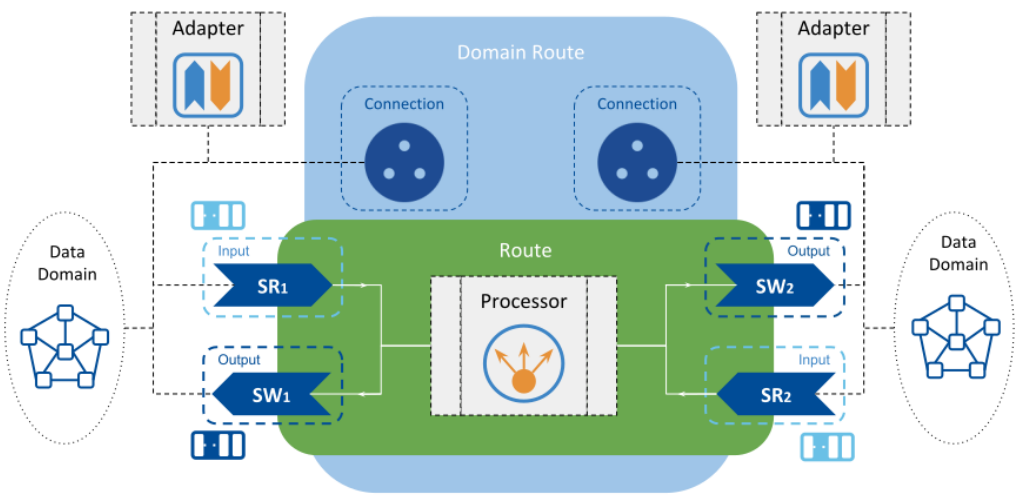


Figure 4.2: *Adapter* concept

Routing Service relies on concrete *Adapter* implementations to *read and write* data streams as part of the configured data flows. Similar to the *TopicRoute* object presented in *Routing a Topic between two different domains*, a *Route* represents a generalization of a *TopicRoute* whose *Inputs* and *Outputs* can interact with any data domain.

Each *Input* and *Output* are attached to a *Connection*, which through the underlying *Adapter* connection entity creates appropriate *StreamReaders* and *StreamWriters*, respectively. These *StreamReaders* and *StreamWriters* provide read and write access to data streams, respectively.

Note: All the following sections require you to be familiar with the routing concepts explained in *Routing Data: Connecting and Scaling Systems*. We also recommended becoming familiar with *Controlling Data: Processing Data Streams*. This section requires software programming knowledge in C/C++, understanding of CMake, and shared library management.

4.1 Unified Data Representation

Routing Service architecture allows all the data-related components such as *Adapter*, *Processor*, and *Transformation* to interoperate and coexist without knowing details of each other. *Routing Service* achieves this by defining a unified data representation that all components are required to use.

The unified data representation model is provided by *DynamicData*, a concept presented in *DynamicData as a Data Representation Model*. *Routing Service* imposes *DynamicData* as the data interface for all the components that have to deal with data streams. This contract for the unified data representation is the key element that enables data integration in *Routing Service*. Therefore, the main responsibility of an *Adapter* implementation is to provide a translation between the domain-specific data representation to *DynamicData* and vice versa.

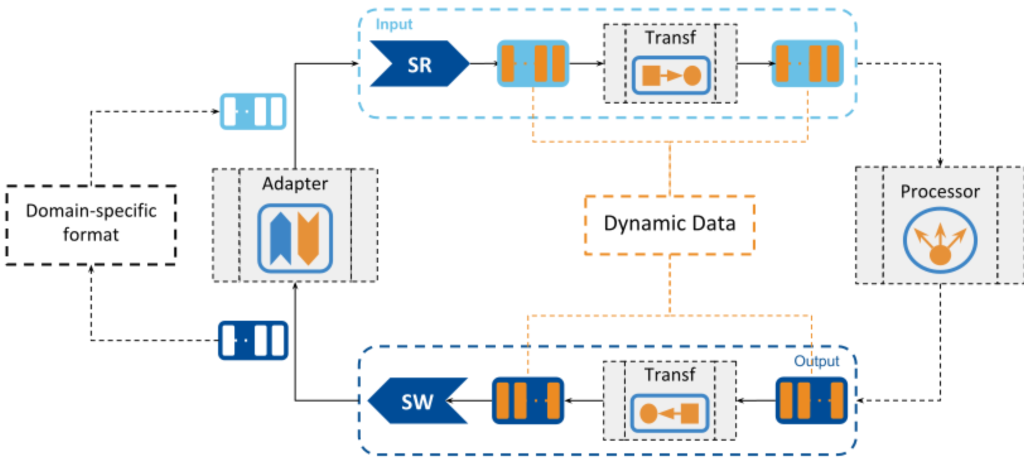


Figure 4.3: Unified Data Representation Model

In Figure 4.3, you can see all the data-related components interacting with each other independently of the domain-specific format of the data. All the data streams that flow across different components are presented as streams of *DynamicData* objects.

The following sections will guide you through an example that implements an *Adapter* that manipulates data from a file system. We will cover each step necessary to implement a custom *Adapter* and explain the purpose of each entity.

4.2 Integrating a File-Based Domain

This section will guide you through an example of how to implement a custom *Adapter* to integrate with a non-DDS technology. The example shows how to feed data stored in a set of `CSV` files back and forth between a DDS domain. The file integration example is shown in Figure 4.4.

The example requires the implementation of a custom *File Adapter*, which provides the ability to read and write from a set files and convert their content into a stream of *DynamicData* samples.

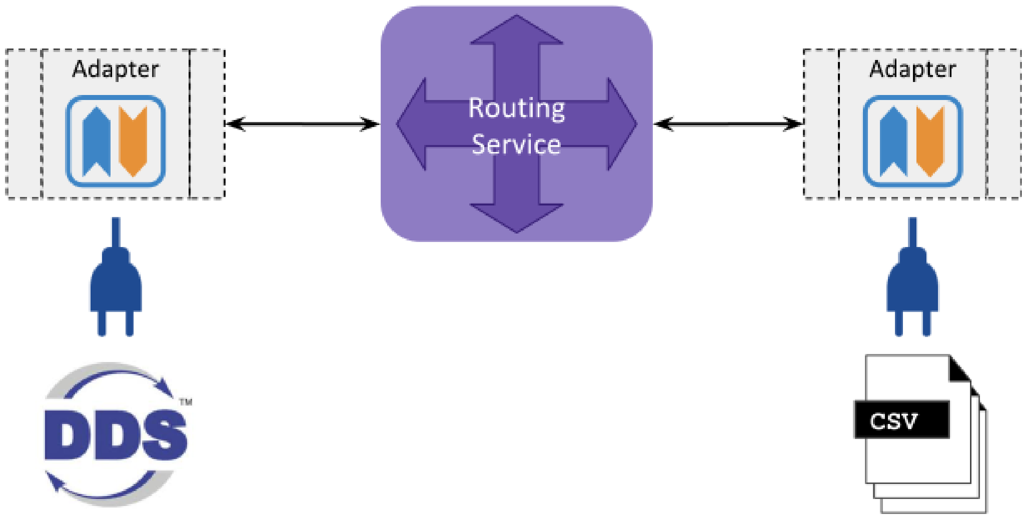


Figure 4.4: Example of data integration with a simple CSV file adapter

Let's review all the tasks you need to do to create a custom *Adapter* using the *Example: Using a File Adapter*. You can run it first to see it in action, but you can also run one step at a time. We explain each method.

4.2.1 Develop a Custom Adapter

As mentioned earlier, there are three main *Adapter* interfaces that must be implemented in order to provide access to, read, and write in a data domain. The most important step in designing a custom *Adapter* is to properly define the **mapping** between the adapter interfaces and specific entities or agents involved in the adapted data domain.

For this example, the mapping is very simple and consists of the following:

- `FileConnection` A simple factory class for *FileStreamReader* and *FileStreamWriter*.
- `FileStreamReader` Reads data from a single file and converts it to *DynamicData*.
- `FileStreamWriter` Writes data to a single file after being converted.

Both the `FileStreamReader` and `FileStreamWriter` process files in a custom and consistent `CSV` format. For simplicity, they also expect and understand the `ShapeType` only.

To better understand how these implementations work, we will split the focus into two separate concepts: reading and writing.

Implement a StreamReader for Reading Data

Reading from a data domain is the responsibility of the *StreamReader*. If you need to provide read access from your integrated data domain, you will need to implement this part of the *Adapter*, although it's optional.

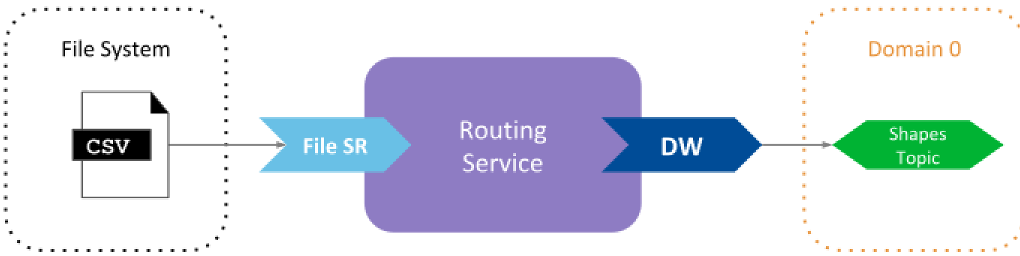


Figure 4.5: Routing from the file adapter to DDS

StreamReader Creation

Creating *StreamReaders* is the responsibility of the *Connection*. Hence the *Adapter* connection interface has an abstract method to implement the creation of a *StreamReader*. In this method you will find, among others, two important parameters:

- Information about the *Stream* for which the *StreamReader* is created. This parameter has type `rti::routing::StreamInfo` and contains:
  - *Stream* name: This is the name provided as part of the *Input* configuration in the <stream_name> tag.
  - Type information: The registered name and `TypeCode` of the type of the input data stream. This information is encapsulated in a `TypeInfo` structure that contains:
    - `type_name`` is the registered type name, as specified in the *Input* configuration in the <registered_type_name> tag.
    - `type_representation` is the type definition as `TypeCode`, obtained either from XML or from *Stream discovery*. You can learn more details about type registration in *Specifying Types*.
- A `StreamReaderListener` object to provide asynchronous notifications about data available to read. This is an object provided by the *Routing Service* engine and the implementation can use it to signal the availability of input stream data and generate an event that's notified to the owner *Route*.

Read Operation

`FileStreamReader` inherits from the `rti::routing::adapter::DynamicDataStreamReader` interface, which has different abstract method overload to read data. Which read operation version is called depends on the behavior of the `Processor` set in the parent *Route*. The default forwarding *Processor* only calls the basic `take()` and is the one our example implements.

When implementing a *StreamReader*, there are two main tasks that require special attention:

- **Providing an input stream of loaned *DynamicData* samples**: All of the abstract read operations have two output parameters that shall hold the returned samples: list of user-data objects, and a list for info-data objects.

  The `FileStreamReader::take()` implementation reads one `CSV` text line at a time, parses each member, and converts it to a `DynamicData` object. In this case, the take operation can only read one sample at a time, and a heap-allocated `DynamicData` is provided as part of the output sample list. Note that `FileStreamReader::return_loan()` frees this heap-allocated object. The `return_loan()` operation is called automatically by the processor implementation when the sample loan from the take operation is no longer needed.

  Note that the take operation may also return a list of info-objects. These objects are meant to provide metadata associated with the user-data objects, such as reception timestamps or sequence numbers (which metadata is available depends on the data domain being adapted). Our example does not provide any metadata and hence the list is returned empty.

- **Notifying *Routing Service* about available data**: This is an important yet subtle step involved in the data processing pipeline. If you look at the `Connection::create_stream_reader` operation you will notice that one of the input parameters is an object of `rti::routing::adapter::StreamReaderListener`. This object is provided by the *Routing Service* engine and you can use it to indicate to *Routing Service* about the existence of data available from the *StreamReader*. When `StreamReaderListener::on_data_available` is called, it will trigger the generation of a `DATA_ON_INPUTS` event that will be dispatched to the *Processor* installed in the parent *Route*.

  In our example implementation, the `FileStreamReader` *spawns a thread* that reads a text line from the file and notifies the `StreamReaderListener` right after, repeating this sequence in a loop until the whole file is read. Note that if we didn't notify the `StreamReaderListener`, then the only way for *Routing Service* to read data would be through a *periodic event* (see *Periodic and Delayed Action*).

Read vs. Take

In the *StreamReader* you will find that there are always two parallel operations with the same signature but different names: one called `read()` and one called `take()`. Their behavior should be the same except for one main difference: `take()` will return samples from a *StreamReader* only once, while `read()` allows the same samples to be returned more than once.

In the DDS world, this is similar to the read and take operations of a *DataReader*. While the behavior is the same in both of them, the take operation will remove the samples from the *DataReader*'s cache (freeing space and preventing them from being read again), while the read will leave the cache intact, simply marking the samples with `READ` status.

Implement a StreamWriter for Writing Data

StreamWriter Creation

Creating *StreamWriters* is responsibility of of the *Connection*. Hence the *Adapter* connection interface has an abstract method to implement the creation of a *StreamWriter*. In this method you will find, among others, an important parameter that identifies the *Stream* for which the *StreamReader* is created. This parameter has type `rti::routing::StreamInfo` and its content and purpose are the same as explained in the reading section above.

Write Implementation

Writing to a data domain is the responsibility of the *StreamWriter*. In our example, `FileStreamWriter` inherits from the `rti::routing::adapter::DynamicDataStreamWriter` interface, which has abstract methods to write data. Similar to the reading part, the write operation is called by the installed *Processor* of the parent *Route*. The default *Processor* calls the write operation, passing the same samples read from the *Inputs* belonging to the same parent *Route*.

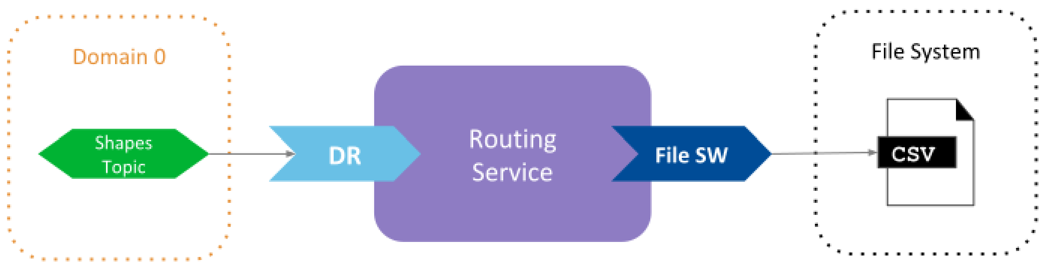


Figure 4.6: Routing from DDS to the file adapter

The abstract write operation receives two input parameters: a list of user-data *DynamicData* objects, and a list of info-data objects of type `SampleInfo`. The info-data list may be empty if no such information is available, though if it's not, then it has the same size as the user-data objects (a 1:1 mapping between user-data and info-data objects).

Our `FileStreamWrite::write()` implementation is as simple as iterating over the list of user-data objects and storing each of them in a file as a separate `CSV` text line. However, our example does not use the info-data list, though it could have used it to store, for example, the timestamps of the samples.

Note: Implementing either the *StreamReader* and *StreamWriter* is optional. You can implement only the side that you need, that is, reading or writing.

See also:

Adapter C++ API reference

Processor Events Overview for the Processor API.

Forwarding Processor Details on the default forwarding *Processor* of the *TopicRoutes*.

4.2.2 Create a Shared Library

Once the *Adapter* implementation is finished, we need to create a *shared library* that *Routing Service* can load. In this example we use `CMake` as the build system to create the shared library. We specify the generation of a library named `fileadapter`:

```
...

add_library(
    fileadapter
        "${CMAKE_CURRENT_SOURCE_DIR}/FileAdapter.cxx"
        "${CMAKE_CURRENT_SOURCE_DIR}/FileConnection.cxx"
        "${CMAKE_CURRENT_SOURCE_DIR}/FileInputDiscoveryStreamReader.cxx"
        "${CMAKE_CURRENT_SOURCE_DIR}/FileStreamReader.cxx"
        "${CMAKE_CURRENT_SOURCE_DIR}/FileStreamWriter.cxx"
)

...
```

The generated library contains the compiled code of our implementation, contained in multiple `.cxx` files. A key aspect of the generated library is that it must export an external function that instantiates the `FileAdapter`, and it is the function that *Routing Service* will call to instantiate the *Adapter*. This external symbol is denoted *entry point* and you can declare it as follows:

```
RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DECL(FileAdapter);
```

The macro declares an external exported function with the following signature:

```
struct RTI_RoutingServicAdapterPlugin*
FileAdapter_create_adapter_plugin(
        const struct RTI_RoutingServiceProperties *,
        RTI_RoutingServiceEnvironment *);
```

which is the signature *Routing Service* requires and will assume for the entry point to create a custom *Adapter*. Note that the implementation of this function requires using the macro `RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DEF` in the source file.

4.2.3 Define a Configuration that Integrates DDS with the File Adapter

This is similar to the process explained in *Routing a Topic between two different domains*, except that we will use a *Connection* from the file adapter and only one *DomainParticipant*.

The example configuration file contains three different configurations that perform the integration in multiple combinations: **file to DDS, DDS to file, and file to file**. Note that all combinations could fit in a single *Routing Service* configuration, but we chose this model to better explain the adapter capabilities.

Below are the steps you need to follow.

Configure a Plugin Library

Within the root element of the XML configuration, you can define a *plugin library* element that contains the description of all the plugins that can be loaded by *Routing Service*. In our case, we define a plugin library with a single entry for our *File Adapter* plugin:

```
<plugin_library name="AdapterLib">
    <processor_plugin name="FileAdapter">
        <dll>fileadapter</dll>
        <create_function>
            FileAdapter_create_adapter_plugin
        </create_function>
    </processor_plugin>
</plugin_library>
```

The values specified for the `name` attributes can be set at your convenience and they shall uniquely identify a plugin instance. We will use these names later within the *Connection* to refer to this plugin. For the definition of our ADAPTER plugin, we have specified two elements:

- `dll` is the name of the shared library we specified in the build step. We just provide the library name so *Routing Service* will try to load it from the working directory, or assume that the library path is set accordingly.
- `<create_function>` is the name of the entry point (external function) that creates the plugin object, exactly as we defined in code with the `RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DECL` macro.

Once we have the plugin defined in the library, we can move to the next step and define a *Connection* to the data domain of this plugin and the *Route* for the data flows for reading and writing.

> **Warning:** When a name is specified in the <dll> element, *Routing Service* will automatically append a **d** suffix when running the debug version of *Routing Service*.

See also:

Plugins Documentation about the `<plugin_library>` element.

Plugin Management For in-depth understanding of plugins.

Define a Connection Linked to the Adapter

As mentioned before, the *Connection* is the entity that enables access to a specific domain. To do so, the connection configuration shall refer to the *Adapter* plugin from which the underlying domain connection shall be created.

In this example, the connection configuration is defined as follows:

```
<connection name="FileConnection" plugin_name="AdapterLib::FileAdapter">
    <registered_type name="ShapeType" type_name="ShapeType"/>
</connection>
```

There are three key elements that shall be set in a *Connection* configuration:

- `name` is the attribute that represents the name of the *Connection* entity. You can choose any name you like that helps you identify the data domain. This name will be used later by the *Input* and *Output* configurations to indicate from which *Connection* their underlying *StreamReader* and *StreamWriter*, respectively, are created. In our case, we named it `FileConnection`.
- `plugin_name` is the attribute that must refer to the *Adapter* plugin from which the adapter connection is created. The value of this attribute must be the fully qualified name of the adapter plugin within the plugin library. The fully qualified name of the plugin is built using the values from the `name` attributes of the plugin library and plugin element. In our case, the fully qualified name of the file adapter plugin is given by `AdapterLib::FileAdapter`.
- `register_type` is an element tag that refers to a type definition (`TypeCode`) described in XML. This element has two attributes: `name` to uniquely identify and register a type, and `type_ref` to point to an existing type in XML providing its fully qualified name. This element can optionally appear as many times as needed. You will need to use this element if your adapter does not support discovery and *Routing Service* cannot provide it through means of others adapters.

  Our file adapter example is quite basic. It only works with the `ShapeType` and it requires the definition to be available in XML (you can find it under the `<types>` section).

Define the Data Flows that Read and Write from Your Adapter

Once a *Connection* to the adapted data domain is available, we need to define the *Routes* (or *AutoRoutes*) that will indicate how data *streams* flow from inputs to outputs. *Inputs* and *Outputs* are ultimately the entities that hold *StreamReaders* and *StreamWriters* that perform the reading and writing.

The file adapter example maps a separate `CSV` file for each *stream*. This allows us to nicely perform a 1:1 mapping between a DDS *Topic* and a file stream. In general, the expectation is that data that is read from an input's *StreamReader* shall originate from a single input stream. Likewise, the data written to an output's *StreamWriter* shall be sent to a single stream.

As mentioned at the beginning, this example provides three different *Routing Service* configurations, each with a single *Route* that defines the data flow for a specific combination. We will review each separately.

Routing from a File Stream to a DDS Topic

For this case we define a *Route* with:

- An input attached to the file adapter (`<input>`). This requires setting the following elements:
  - `connection` is the attribute that specifies from which *Connection* the underlying *StreamReader* is created. This attribute shall refer to the name of the *Connection* configuration exactly as it was set in its `name` attribute.
  - `<stream_name>` is the *stream* name associated with this input. The impact of this value is specific to each *Adapter* implementation.
  - `<registered_type_name>` indicates the associated type to the input stream. This ultimately translates into finding a `TypeCode` that matches this name and providing it on the *StreamReader*

creation as part of the `StreamInfo`. In our case, this name matches the value in the `name` attribute of the `<register_type>` element in the connection configuration, so the type is the one defined in XML.

- `<property>` is the adapter-specific configuration in the form of name-value pairs. This content is passed directly as a set of name-value string pairs on the creation of the *StreamReader*. Our file *StreamReader* receives the name of the `CSV` file from where data is read and a period at which the file is read.

- An output attached to the built-in DDS adapter (`<dds_output>`). This requires setting the following elements:
  - `participant` is the attribute that specifies from which *DomainParticipant* the underlying *StreamWriter* is created. This attribute shall refer to the name of the *DomainParticipant* configuration exactly as it was set in its `name` attribute.
  - `<topic_name>` is the name of the *Topic* the underlying *DataWriter* writes to.
  - `<registered_type_name>` indicates the type associated with the *Topic*. This has the same behavior as for the input.

The XML is shown below.

```xml
<route>
    <input connection="FileConnection">
        <creation_mode>ON_DOMAIN_MATCH</creation_mode>
        <stream_name>$(SHAPE_TOPIC)</stream_name>
        <registered_type_name>ShapeType</registered_type_name>
        <property>
            <value>
                <element>
                    <name>example.adapter.input_file</name>
                    <value>Input_$(SHAPE_TOPIC).csv</value>
                </element>
                <element>
                    <name>example.adapter.sample_period_sec</name>
                    <value>1</value>
                </element>
            </value>
        </property>
    </input>
    <dds_output participant="DDSConnection">
        <creation_mode>ON_ROUTE_MATCH</creation_mode>
        <registered_type_name>ShapeType</registered_type_name>
        <topic_name>$(SHAPE_TOPIC)</topic_name>
    </dds_output>
</route>
```

Routing from a DDS Topic to a File Stream

For this case we define a *Route* with:

- An input attached to the built-in DDS adapter (`<dds_input>`). This requires setting the following elements:
  - `participant` is the attribute that specifies from which *DomainParticipant* the underlying *StreamReader* is created. This attribute shall refer to the name of the *DomainParticipant* configuration exactly as it was set in its `name` attribute.
  - `<topic_name>` is the name of the *Topic* the underlying *DataReader* reads data from.
  - `<registered_type_name>` indicates the type associated with the *Topic*. This has the same behavior as for the input.
- An output attached to the file adapter (`<output>`). This requires setting the following elements:
  - `connection` is the attribute that specifies from which *Connection* the underlying *StreamWriter* is created. This attribute shall refer to the name of the *Connection* configuration exactly as it was set in its `name` attribute.
  - `<stream_name>` is the *stream* name associated with this output. The impact of this value is specific to each *Adapter* implementation.
  - `<registered_type_name>`: indicates the associated type to the output stream. This ultimately translates into finding a `TypeCode` that matches this name and providing it on the *StreamWriter* creation as part of the `StreamInfo`. In our case, this name matches the value in the `name` attribute of the `<register_type>` element in the connection configuration, so the type is the one defined in XML.
  - `<property>` is the adapter-specific configuration in the form of name-value pairs. This content is passed directly as a set of name-value string pairs on the creation of the *StreamWriter*. Our file *StreamWriter* receives the name of the `CSV` file where data is written.

The XML is shown below.

```xml
<route>
    <dds_input participant="DDSConnection">
        <creation_mode>ON_ROUTE_MATCH</creation_mode>
        <registered_type_name>ShapeType</registered_type_name>
        <topic_name>$(SHAPE_TOPIC)</topic_name>
    </dds_input>
    <output connection="FileConnection">
        <creation_mode>ON_ROUTE_MATCH</creation_mode>
        <registered_type_name>ShapeType</registered_type_name>
        <stream_name>$(SHAPE_TOPIC)</stream_name>
        <property>
            <value>
                <element>
                    <name>example.adapter.output_file</name>
                    <value>Output_$(SHAPE_TOPIC).csv</value>
                </element>
            </value>
```

(continues on next page)

(continued from previous page)

```
        </property>
    </output>
</route>
```

Routing from a File Stream to Another File Stream

This scenario represents a case where both the input and output are attached to the file *Adapter*. Hence, the routing path of this configuration generates a flow from file to file. This scenario demonstrates the flexibility and abstraction of *Routing Service* working agnostically with data domains.

For this case, the *Route* configuration is defined with the same input configuration from *Routing from a File Stream to a DDS Topic* and the same output configuration from *Routing from a DDS Topic to a File Stream*.

The XML is shown below.

```
<route>
     <input connection="FileConnection">
        <creation_mode>ON_DOMAIN_MATCH</creation_mode>
        <stream_name>$(SHAPE_TOPIC)</stream_name>
        <registered_type_name>ShapeType</registered_type_name>
        <property>
            <value>
                <element>
                    <name>example.adapter.input_file</name>
                    <value>Input_$(SHAPE_TOPIC).csv</value>
                </element>
                <element>
                    <name>example.adapter.sample_period_sec</name>
                    <value>1</value>
                </element>
            </value>
        </property>
    </input>
    <output connection="FileConnection">
        <creation_mode>ON_ROUTE_MATCH</creation_mode>
        <registered_type_name>ShapeType</registered_type_name>
        <stream_name>$(SHAPE_TOPIC)</stream_name>
        <property>
            <value>
                <element>
                    <name>example.adapter.output_file</name>
                    <value>Output_$(SHAPE_TOPIC).csv</value>
                </element>
            </value>
        </property>
    </output>
</route>
```

Note: In all configurations, the Stream and *Topic* names are set using the XML variable `SHAPES_TOPIC`.

Its purpose is to allow reusing the same configuration providing the actual desired name at run time. Another alternative is to use an *AutoRoute* instead (see *Routing a group of Topics*).

4.3 Discovery Capabilities

Besides allowing integration with application or user data, the *Adapter* interface also provides *data stream discovery* capabilities.

Data communication frameworks may offer the ability to detect at run time which streams of user-data information are available and which endpoints (producers and consumers) are involved in the communication. Such is the case with DDS for example, which has a builtin discovery protocol to detect and notify applications of the presence of *Topics*.

Discovery is very useful because it eliminates deployment configuration complexity and allows dynamic systems where endpoints come and go to function autonomously. *Routing Service* can interoperate with discovery streams from any data domain through the *Adapter* by defining the concept of *Stream discovery*.

Stream discovery refers to the ability to detect the presence of streams of information that carry user data. User-data streams are categorized as:

- *publication or input streams*: These are data streams that originate from producer endpoints and from which *Routing Service* receives data using *StreamReaders*. An input stream is read-only.
- *subscription or output streams*: These are data streams that originate from the consumer endpoints and to which *Routing Service* sends data using *StreamWriters*. An output stream is write-only.

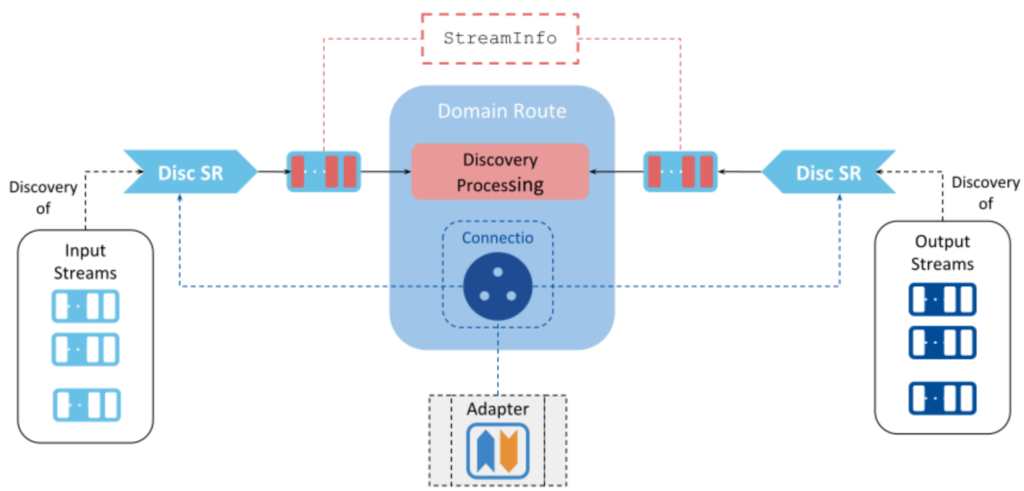


Figure 4.7: Integration with discovery capabilities of data domains

Routing Service uses stream discovery for two main activities:

- Detecting the **generation or disposal of streams** that trigger the filter matching with *AutoRoutes* (see *Routing a group of Topics*) and the creation of *StreamReaders* and *StreamWriters* based on the input and output *creation modes* (see *Creation Modes*).

- Receiving **information about the type of the samples** carried on the user-data streams. *Routing Service* needs to obtain the *Stream* type information (`TypeCode`) beforehand in order to create the *StreamReaders* and *StreamWriters*. *Stream* discovery provides a channel for the reception of types.

The discovery information in *Routing Service* is represented in a unified way by defining a common type to describe *Stream* information: *StreamInfo*. Key information that a *StreamInfo* object provides:

- *Stream* name: A unique identifier of a *stream* within a particular data domain connection (e.g., a *Topic* name in a DDS domain).
- Alive or dispose: Whether or not the stream has any alive endpoints associated with it.
- *TypeInfo*: Contains the unique identifier for the registration name of the type, as well as the type description as a `TypeCode`.

Routing Service receives *StreamInfo* objects through the *Discovery StreamReader* interfaces from the *Adapter*. Namely, there are two discovery *StreamReaders* to read *StreamInfo* samples, one for each input and output stream.

Implementation of discovery in the *Adapter* is optional. The *Connection* is responsible for the provision of the *Discovery StreamReaders* and its interface has two abstract methods to retrieve them.

Routing Service calls these operations upon enabling the parent *DataReader* (typically at startup) and will use the returned *StreamReaders* (if any) to obtain the *StreamInfo* objects from them. *Routing Service* has a dedicated *discovery thread* to call the read and return loan operations from all discovery *StreamReaders*.

The next section shows an example of how to provide discovery information using the file *Adapter*.

4.3.1 Discovery in a File-Based Domain

When working with files on a file system, there are many ways in which discovery information can be useful. One of them is to provide notification about the creation and removal of files. Our file adapter example shows a basic way to recreate this.

The file adapter example implements only the input stream *Discovery StreamReader*. It provides information about which files are available to read and when the user *StreamReaders* are done reading them.

The class `FileInputDiscoveryStreamReader` inherits from the abstract class `rti::routing::adapter::DiscoveryStreamReader` and represents the implementation of the *StreamReader* that provides discovery information about input streams.

The implementation of this class is similar to the user-data *StreamReader*. You will find that the abstract take operation is implemented by returning a list of `rti::routing::StreamInfo` objects. The implementation also uses an `rti::routing::adapter::StreamReaderListener` object to notify *Routing Service* about discovery information that is available to read.

The file `FileInputDiscoveryStreamReader` has two ways to generate `StreamInfo` objects:

- On class instantiation, which in this case occurs when *Routing Service* calls the `FileConnection::get_input_stream_discovery_reader`. The constructors checks for the existence of `CSV` files containing the user data in hard-coded locations.

- When user *StreamReaders* obtain an end-of-file token, they call `FileInputDiscoveryStreamReader::dispose`, which will generate a `StreamInfo` object marked as disposed for each finished file.

The file adapter has basic code to illustrate how to implement the discovery functionality. More useful behavior could include providing continuous notifications about new files (hence new streams) to be read. It could also implement the *output Discovery StreamReader* by also detecting when a file is placed in a directory as a signal to write data obtained from a peer input stream.

4.4 Key Terms

Data Integration The process of combining data from multiple and different sources for analysis, processing, or system integration purposes.

Adapter Pluggable component that allows access to a custom data domain.

Info Object A structure that contains metadata associated with the user-data object. In DDS, this is defined as `SampleInfo`.

Sample A structure composed of a data object and its associated info object.

Loaned samples A list of samples returned by a *StreamReader* for which a return loan operation is perform.

Stream Discovery A mechanism through which *Routing Service* detects the presence of user-data streams.

StreamInfo A common data structure to represent discovery information across all data domains.

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 *Remote Administration Platform*. 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 *Routing Service Tag*) or the `-remoteAdministrationDomainId` command-line parameter, which enables remote administration and sets the domain ID for remote communication (see *Command-Line Executable*).

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 *Resource Identifiers* 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 *Example: Configuration Reference* as a reference. (For a complete reference of the available configuration tags used in *Routing Service*, see *XML Tags for Configuring RTI Routing Service*.)

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
```

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
```

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"/>`
inputObjectRepresentation	`<xs:element name="input" type="routeStreamPort"/>`

continues 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)
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.
	UPDATE /routing_services/(rs):save	Saves the *Service* loaded configuration.

continues on next page

Table 5.3 – continued from previous page

Resource	Operation	Description
	DELETE /routing_services/(rs)/domain_routes/(dr)	Deletes a *DomainRoute* object.
	DELETE /routing_services/(rs)/config	Deletes 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.
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* and non-*Connext*).
Output	*UPDATE <SERVICE>/domain_route/(dr)/sessions/(s)/routes(r)/outputs/(o)*	Updates an *Output* (*Connext* and non-*Connext*).

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* (*Create Resource*).

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* (*Get Resource*).

Example reply body:

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

UPDATE /routing_services/(rs)

Operation: `update`

Updates the specified *Service* object.

See *Update Resource* (*Update Resource*).

The expected XML configuration is a subset of *routerObjectRepresentation* and only contains the properties that are mutable and whose values have changed.

Example Update a *Service* with the name “MyRouter”.

Request Field	Value
action	UPDATE
resource_identifier	/routing_services/MyRouter
string_body	`str://\"<routing_service> <save_path>./service_snapshot.xml</save_path> </routing_service>"`

UPDATE /routing_services/(rs)/state

Operation: set_state

Sets the state of a *Service* object.

See *Set Resource State* (*Set Resource State*).

Valid requested states:

- ENABLED
- DISABLED
- 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* (*Delete Resource*).

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* 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* (*Create Resource*).

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* (*Update Resource*).

The expected XML configuration is a subset of *domainRouteObjectRepresentation* and only contains the properties that are mutable and whose values have changed.

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

Request Field	Value
action	UPDATE
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* (*Set Resource State*).

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* (*Delete Resource*).

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* (*Update Resource*).

The expected XML configuration is a subset of *participantObjectRepresentation* or *connectionObjectRepresentation* and only contains the properties that are mutable and whose value is changed.

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

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/connections/MyParticipant

`string_body`	`str://"<participant>`
	`<domain_participant_qos>`
	`<property>`
	`<value>`
	`<element>`
	`<name>property_name</name>`
	`<value>property_new_value</value>`
	`</element>`
	`</value>`
	`</property>`
	`</domain_participant_qos>`
	`</participant>"`

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

Request Field	Value
action	UPDATE
resource_identifier	/routing_services/MyRouter/domain_routes/MyDomainRoute/connections/MyConnection
string_body	`str://"<connection> <property> <value> <element> <name>property_name</name> <value>property_new_value</value> </element> </value> </property> </connection>`

DELETE \<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 peers 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* (*Create Resource*).

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* (*Create Resource*).

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* (*Update Resource*).

The expected XML configuration is a subset of *sessionObjectRepresentation* and only contains the properties that are mutable and whose values have changed.

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

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession
`string_body`	`str://"<session> <publisher_qos> <partition> <name> <element>MyNewPartition</element> </name> </partition> </publisher_qos> </session>"`

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

Operation: `set_state`

Sets the state of a *Session* object.

See *Set Resource State* (*Set Resource State*).

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* (*Delete Resource*).

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

Operation `delete_route`

Deletes the specified *Route*.

See *Delete Resource* (*Delete Resource*).

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* (*Update Resource*).

The expected XML configuration is a subset of *autoRouteObjectRepresentation* or *autoTopicRouteObjectRepresentation* and only contains the properties that are mutable and whose value is changed.

Note that *AutoRoute* or *AutoTopicRoute* don't have any children resources. All the properties defined for the XML representation can be used for the update operation. Also the *Route* or *TopicRoute* created as part of an *AutoRoute* or *AutoTopicRoute* can be updated independently.

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

Request Field	Value
action	UPDATE
resource_identifier	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/auto_routes/MyAutoRoute
string_body	str://"<auto_route> <dds_input> <datareader_qos> <period> <sec>1</sec> <nanosec>0</nanosec> </period> </datareader_qos> </dds_input> </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* (*Set Resource State*).

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* (*Update Resource*).

The expected XML configuration is a subset of *routeObjectRepresentation* or *topicRouteObjectRepresentation* and only contains the properties that are mutable and whose value is changed.

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

Request Field	Value
action	UPDATE
resource_identifier	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/routes/MyRoute
string_body	`str://"<route> <processor> <property> <value> <element> <name>property_name</name> <value>property_new_value </value> </element> </value> </property> </processor> </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* (*Set Resource State*).

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* (*Update Resource*).

The expected XML configuration is a subset of *routeInputObjectRepresentation* or *topicRouteInputObjectRepresentation* and only contains the properties that are mutable and whose value is changed.

Example Update *Input* with the name “MyInput” under the *TopicRoute* “MyRoute”, with its configuration provided as a `str://` scheme.

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/routes/MyRoute/inputs/MyInput
`string_body`	str://"<input> <datareader_qos> <period> <sec>1</sec> <nanosec>0</nanosec> </period> </datareader_qos> </input>"

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

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/routes/MyRoute/inputs/MyInput
`string_body`	str://"<input> <property> <value> <element> <name>property_name</name> <value>property_new_value</value> </element> </value> </property> </input>"

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

Operation: `update`

See *Update Resource* (*Update Resource*).

The expected XML configuration is a subset of *routeOutputObjectRepresentation* or *topicRouteOutputObjectRepresentation* and only contains the properties that are mutable and whose value is changed.

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

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/routes/MyRoute/inputs/MyInput
`string_body`	`str://"<output> <datawriter_qos> <period> <sec>1</sec> <nanosec>0</nanosec> </period> </datawriter_qos> </output>"`

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

Request Field	Value
`action`	UPDATE
`resource_identifier`	/routing_services/MyRouter/domain_routes/MyDomainRoute/sessions/MySession/routes/MyRoute/outputs/MyOutput
`string_body`	`str://"<output> <property> <value> <element> <name>property_name</name> <value>property_new_value</value> </element> </value> </output> </input>"`

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 -->
```

(continues on next page)

(continued from previous page)

```
                <route name="MyRoute">
                    <route_types>true</route_types>
                    <input name="MyInput">
                        ...
                        <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>
```

5.4 The Remote Administration Shell

Any *Connext* application can be implemented to send remote administration commands and receive the corresponding responses. A shell application that sends/receives these commands is provided with *Routing Service*.

The script for the shell application is in `<NDDSHOME>/bin/rtirssh`.

Entering `rtirssh -help` will show you the command-line options:

```
RTI Routing Service Shell
Usage: rtirssh [options]...
Options:
 -domainId  <integer>   Domain ID for the remote configuration
 -timeout   <seconds>   Max time to wait for a remote response
 -cmdFile   <file>      Run commands in this file
 -help                  Displays this information
```

5.4.1 Remote Shell Commands

This section describes the remote commands using the shell interface. The available remote commands are:

Command	Parameters
add_peer	`<target_routing_service> <domain_route_name> p1|p2 <peer_list>`
create	`<target_routing_service> domain_route|session|topic_route|auto_route [<parent_entity_name>] <xml_url> [remote|local]`
delete	`<target_routing_service> [<entity_name>]`
disable	`<target_routing_service> [<entity_name>]`
enable	`<target_routing_service> [<entity_name>]`
get	`<target_routing_service>`
load	`<target_routing_service> <cfg_name><xml_url> [remote|local]`
pause	`<target_routing_service> [<entity_name>]`
resume	`<target_routing_service> [<entity_name>]`
save	`<target_routing_service>`
shutdown	`<target_routing_service>`
unload	`<target_routing_service>`
update	`<target_routing_service> [<entity_name>] [<xml_url>|<assignment_expr>] [remote|local]`

5.4.2 Command: add_peer

```
add_peer <target_routing_service> <domain_route_name> p1|p2 <peer_list>
```

The `add_peer` command passes the `peer_list` to the underlying DomainParticipant's `add_peer()` function. It is only valid for DomainParticipants in a Domain Route. Parameter `<domain_route_name>` is like `<entity_name>`, but must be a Domain Route entity. Parameter `p1|p2` specifies if the DomainParticipant associated with `<participant_1>` or `<participant_2>` configuration is selected. Parameter `<peer_list>` is a comma-separated list of peers.

5.4.3 Command: create

```
create <target_routing_service> domain_route|session|topic_route|auto_route
        [<parent_entity_name>] <xml_url> [remote|local]
```

The create command is similar to *update*, but the configuration is applied to a newly created entity instead of an existing one. The second parameter `(domain_route|session|topic_route|auto_route)` is the kind of entity to be created. If the kind is a `domain_route`, there will be no parent. For the other kinds (`session`, `topic_route`, or `auto_route`), a `<parent_entity_name>` must be specified. Parameters `<xml_url>` and `[remote|local]` are the same as those used in *update*, except that only XML snippets matching the entity kind are allowed. A full file (starting with `<dds>...`) is not valid.

For example (this would be entered as a single command, with no line-breaks):

```
create example topic_route DomainRoute::Session
    str://"<topic_route name="TrianglesToTriangles">
    <input participant="1"><registered_type_name>ShapeType
    </registered_type_name><topic_name>Triangle</topic_name></input>
    <output><registered_type_name>ShapeType</registered_type_name>
    <topic_name>Triangle</topic_name></output></topic_route>"
```

5.4.4 Command: delete

```
delete <target_routing_service> [<entity_name>]
```

You can invoke the `delete` command on Domain Routes, Routes and Auto Routes. It acts like the *disable* command, but also purges the configuration data for the target entity.

For example:

```
delete example DomainRoute::Session::CirclesToCircles
```

A deleted entity cannot be re-enabled, but a new one can be created. `

5.4.5 Command: disable

```
disable <target_routing_service> [<entity_name>]
```

The disable command disables a *Routing Service* entity by destroying its sub-entities and corresponding DDS objects:

- Routing service: When a *Routing Service* is disabled, all of its Domain Routes are destroyed. You do not need to specify the entity_name to disable a *Routing Service*.
- Domain Route: When a Domain Route is disabled, all its Routes, Topic Routes, Auto Routes, and Auto Topic Routes are destroyed, as well as both Connections (DomainParticipants for DDS). All the session threads are stopped and their corresponding adapter sessions (*Publisher* and *Subscriber* for DDS) are also deleted.
- Route, Topic Route, Auto Route and Auto Topic Route: When a Route, Topic Route, Auto Route, or Auto Topic Route is disabled, its StreamReaders and StreamWriters are destroyed, so data will no longer be routed.

5.4.6 Command: enable

```
enable <target_routing_service> [<entity_name>]
```

The enable command enables an entity that has been disabled or marked as ‘enabled=false’ in the configuration file.

This command can be used to enable the following entities:

- Routing service: When a *Routing Service* is enabled, it uses the currently loaded configuration and starts. You don’t need to specify the entity_name to enable a *Routing Service*.
- Domain Route: When a Domain Route is enabled, it creates the Participants, Routes, Topic Routes, Auto Routes, and Auto Topic Routes that it contains. The Routes, Topic Routes, Auto Routes, and Auto Topic Routes will be created enabled or disabled depending on their current configuration. Enabling a Domain Route is required to start routing data from the input domain to the output domain.
- Route, Topic Route, Auto Route, and Auto Topic Route: Enabling a Route, Topic Route, Auto Route or Auto Topic Route is a necessary condition to start routing data between input and output streams. However, data routing will not start until the StreamWriter and StreamReader associated with a Route are created (see *Creation Modes* for additional information).

5.4.7 Command: get

```
get <target_routing_service>
```

The get command retrieves the current configuration.

The retrieved configuration, provided in an XML string format, is functionally equivalent to the loaded XML file, plus any updates (either from an update command or other remote commands that change the configuration, such as `add_peer`). However, the retrieved configuration may not be textually equivalent. For example, the retrieved configuration may explicitly contain default values that were not in the initial XML.

5.4.8 Command: load

```
load <target_routing_service> <cfg_name> <xml_url> [remote|local]
```

The load command loads specific XML configuration code. The `target_routing_service` must be disabled. For more information, see How to Load the XML Configuration *here*.

The XML code received must represent a valid *Routing Service* configuration file. The name of the `<routing_service>` tag to load is identified with `<cfg_name>`.

5.4.9 Command: pause

```
pause <target_routing_service> <entity_name>
```

When the pause command is called for a Route, the session thread containing this Route will stop reading data from the Route's StreamReader.

For *Routing Service*, Domain Routes, Auto Routes, and Auto Topic Routes, the execution of this command will pause the contained Topic Routes and Routes.

5.4.10 Command: resume

```
resume <target_routing_service> <entity_name>
```

When the resume command is called for a Route, the session thread containing this Route will continue reading data from the Route's StreamReader.

For *Routing Service*, Domain Routes, Auto Routes and Auto Topic Routes, the execution of this command will resume the contained Topic Routes and Routes.

5.4.11 Command: save

```
save <target_routing_service>
```

This command writes the current configuration to a file. The file itself is specified with `<save_path>` (see tag within the *Administration Tag* table). If `<save_path>` has not been specified, the save command will fail. If the file specified by `<save_path>` already exists, the file will be overwritten.

The saved configuration is functionally equivalent to the loaded XML file plus any updates (either from an `update` command or other remote commands that change the configuration, such as `add_peer`). However it may not be textually equivalent. For example, the saved XML configuration may explicitly contain default values that were not in the initial XML.

Note: If the `<autosave_on_update>` tag (see tag within the *Administration Tag* table) is set to `TRUE`, this will automatically trigger a save command when configuration updates are received.

5.4.12 Command: shutdown

```
shutdown <target_routing_service>
```

The shutdown command initiates the shutdown sequence on the process where the `target_routing_service` runs. The result of the remote shutdown command depends on how *Routing Service* is instantiated:

- If Routing Service runs as a process executed by the shipped executable in your *RTI Connext* installation, the process will exit upon command reception.
- If *Routing Service* is instantiated as a library in your application, the service instance will notify the installed remote shutdown hook. In this case, the application creating the Routing Service instance is responsible to handle the shutdown sequence. If the shutdown hook is not set, the command request will fail with a response indicating an error.

On a successful shutdown request, *Routing Service* will send a reply with `RTI_ROUTING_SERVICE_COMMAND_RESPONSE_OK`, or `RTI_ROUTING_SERVICE_COMMAND_RESPONSE_ERROR` and an error message indicating the problem.

This command will take effect regardless of the `target_routing_service`'s enabled state.

5.4.13 Command: unload

```
unload <target_routing_service>
```

The unload command unloads the current configuration that the target_routing_service is using, so you can change it with a subsequent *load* command.

The `target_routing_service` must be disabled for this command to succeed.

5.4.14 Command: update

```
update <target_routing_service> [<entity_name>] [<xml_url>|<assignment_expr>␣
↪[remote|local]
```

The update command changes the configuration of a specific entity. The following table shows the parameters that can be changed for each entity:

Entity	Mutable (can be changed at any time)	Immutable (can only be changed when disabled)
Routing Service	• <monitoring><enabled> • <monitoring><status_publication_period> • <entity_monitoring><enabled> • <entity_monitoring><status_publication_period> • <administration><save_path> • <administration><autosave_on_update>	• <monitoring><statistics_sampling_period> • <monitoring><historical_statistics> • <monitoring><domain_id> • <entity_monitoring><statistics_sampling_period> • <entity_monitoring><historical_statistics> • <administration>
Domain Route	• <connection>: Mutable properties in <property> (adapter-specific) • <participant>: Mutable QoS policies in <domain_participant_qos> • <entity_monitoring><enabled> • <entity_monitoring><status_publication_period>	• <connection>: Immutable properties in <property> (adapter-specific) • <domain_participant_qos>: Immutable QoS policies in <domain_participant_qos> • <entity_monitoring><statistics_sampling_period> • <entity_monitoring><historical_statistics>
Session	• (Non-DDS) Mutable properties in <property> (adapter-specific) • (DDS) Mutable QoS policies in <publisher_qos> and <subscriber_qos> • <entity_monitoring><enabled> • <entity_monitoring><status_publication_period>	• (Non-DDS) Immutable properties in <property> (adapter-specific) • (DDS) Immutable QoS policies in <publisher_qos> and <subscriber_qos> • <entity_monitoring><statistics_sampling_period> • <entity_monitoring><historical_statistics>
Route	• Mutable properties in <property> (adapter-specific) • Mutable properties in <transformation><property> (transformation-specific)	• Immutable properties in <property> (adapter-specific) • Immutable properties in <transformation><property> (transformation-specific)
Auto Route	• Mutable properties in <property> (adapter-specific)	• Immutable properties in <property> (adapter-specific)
Topic Route	• Mutable QoS policies in <datawriter_qos> and <datareader_qos> • Mutable properties in	• Immutable properties in <datawriter_qos> and <datareader_qos> • <creation_mode>

If you try to change an immutable parameter in an entity that is enabled, you will receive an error message. To change an immutable parameter, you must disable the *Routing Service* entity, change the parameter, and then enable the *Routing Service* entity again.

You can send an XML snippet (or an assignment expression) that only contains the values you want to change for that entity, or you can send a whole well-formed configuration file:

- If you send an XML snippet (or an assignment expression), only the changes you specify will take effect. For example, suppose you send this command:

  ```
  update ShapeRouter DomainRoute1::Session1::SquareToCircles
      str://"<topic_route><input><datareader_qos><deadline><period>
      <sec>1</sec></period></deadline></datareader_qos></input>
      </topic_route>"
  ```

  or

  ```
  update ShapeRouter DomainRoute1::Session1::SquareToCircles
      topic_route.input.datareader_qos.deadline.period.sec = 1
  ```

  The Topic Route `DomainRoute1::Session1::SquareToCircles` will only change the period value in the Deadline QoS for that particular *DataReader*.

  Now suppose that later on you send this command:

  ```
  update ShapeRouter DomainRoute1::Session1::SquareToCircles
      str://"<topic_route><input><datareader_qos><property>
      <value><element><name>MyProp</name><value>MyValueRemote</value>
      </element></value></property><datareader_qos></input>
      </topic_route>"
  ```

  This would only change the Property QoS; the Deadline QoS would keep the setting from the prior command. In both cases, an update command can only reconfigure one entity at a time and *Routing Service* will ignore all contained entities. For example, a command to update a session will not modify the configuration of its contained Routes. If you need to reconfigure several entities at the same time, consider using the *load* command.

- If you send a well-formed configuration file (starting with `<dds><routing_service>`), the properties in the Route (QoS values in the Topic Route) will be completely replaced with the properties (QoS values) defined in the XML code. If a QoS value for a Topic Route is not defined in the XML code, *Routing Service* will use the *Connext* default.

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 *Monitoring Distribution Platform*. We recommend that you read *Monitoring Distribution Platform* 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 *Routing Service Tag*) or the `-remoteMonitoringDomainId` command-line parameter, which enables remote monitoring and sets the domain ID for data publication (see *Command-Line Executable*).

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`

continues on next page

Table 6.1 – continued from previous page

Keyed Resource Class	Config	Event	Periodic
AutoRoute/AutoTopicRoute	`AutoRouteConfig`	`AutoRouteEvent`	`AutoRoutePeriodic`
Route/TopicRoute	`RouteConfig`	`RouteEvent`	`RoutePeriodic`
Input	`InputConfig`	`InputEvent`	`InputPeriodic`
Output	`OutputConfig`	`OutputEvent`	`OutputPeriodic`

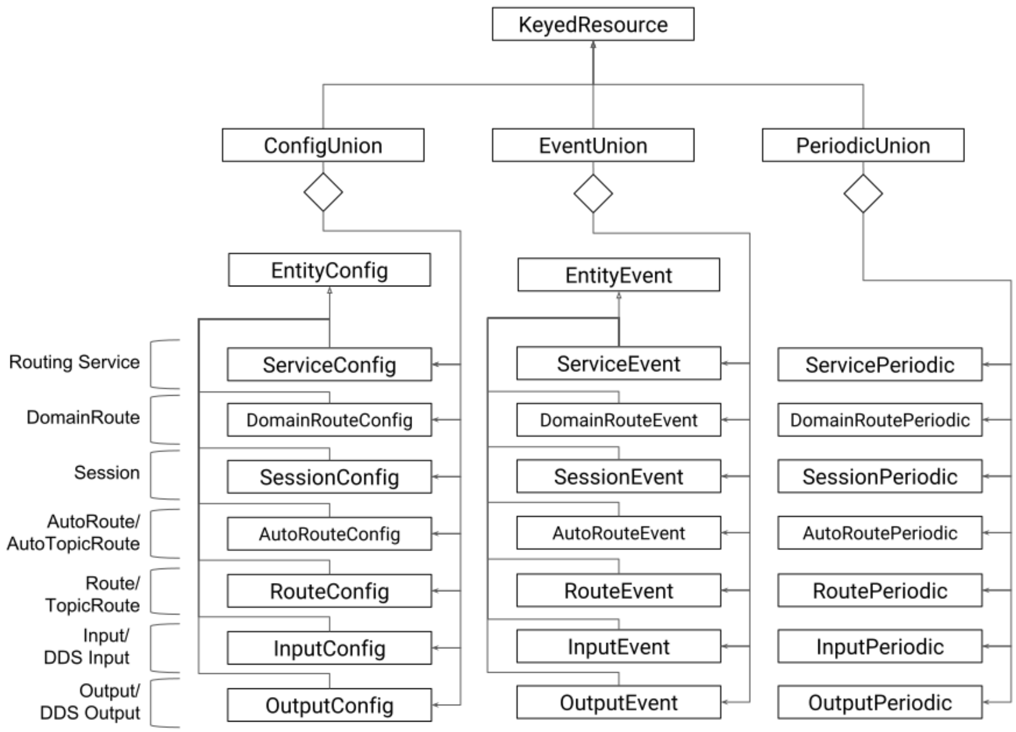


Figure 6.1: Keyed Resource Types for *Routing Service* monitoring

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

@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 12.14.
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 12.10.
process	See Table 12.12.

Table 6.3: `ServiceEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.15.

Table 6.4: `ServicePeriodic`

Field Name	Description
host	See Table 12.11.
process	See Table 12.13.

6.2.2 DomainRoute

Listing 6.2: *DomainRoute* Types

```
@mutable @nested
struct ConnectionConfigInfo {
    BoundedString name;
    AdapterClassKind class;
    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 12.14.
connections	Sequence of `ConnectionInfo` objects, one for each *Connection* inside the *DomainRoute*. 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 12.15.

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;
    @optional Service::Monitoring::ThreadPoolPeriodic thread_pool;
};
```

Table 6.9: `SessionConfig`

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

Table 6.10: `SessionEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 12.15.

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*.
thread_pool	Sequence of `ThreadPeriodic` objects, one for each thread of the *Session*'s thread pool. See Table 12.17.

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;
};
```

(continues on next page)

(continued from previous page)

Table 6.12: `AutoRouteConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 12.14.
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 12.15.

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 12.14.
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 12.15.

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 12.14.
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 12.15.

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

Usage

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

7.1 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* (*Starting Routing Service*).
- How to Stop *Routing Service* (*Stopping Routing Service*).
- *Routing Service* Command-line Parameters (*Routing Service Command-Line Parameters*).

7.1.1 Starting Routing Service

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

Linux/macOS

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

Windows

```
> %NDDSHOME%\bin\rtiroutingservice
```

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

Table 7.1 describes the command-line parameters.

Note: To run *Routing Service* on a *target* system (not your host development platform), you must first select the target architecture. To do so, either:

- Set the environment variable `CONNEXTDDS_ARCH` to the name of the target architecture. (Do this for each command shell you will be using.)
- Or set the variable `connextdds_architecture` in the file `rticommon_config.[sh/bat]` to the name of the target architecture. (The file is `resource/scripts/rticommon_config.sh` on Linux or macOS systems, `resource/scripts/rticommon_config.bat` on Windows systems.) If the `CONNEXTDDS_ARCH` environment variable is set, the architecture in this file will be ignored.

7.1.2 Stopping Routing Service

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

7.1.3 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 7.1: 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

continues on next page

Table 7.1 – continued from previous page

Parameter	Description
-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.
-licenseFile <path>	Specifies the path to the license file. See *How to use a License File with RTI Services*.
-listConfig	Prints the available configurations and exits.
-logFile <file>	Redirects logging to the specified file.
-logFormat <string>	A mask to configure the format of the log messages for both the service and DDS. • `DEFAULT` - Print message, method name, log level, activity context, and logging category • `VERBOSE` - Print `DEFAULT` information, plus the following: module, thread ID, and message location (and spread the message over two lines) • `TIMESTAMPED` - Print `VERBOSE` information, timestamped • `MINIMAL` - Print only message number and message location • `MAXIMAL` - Print all available fields
-maxObjectsPerThread <int>	Maximum number of thread-specific objects that can be created. **Default:** Same as the Connext DDS default for max_objects_per_thread
-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

continues on next page

Table 7.1 – continued from previous page

Parameter	Description
-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 <service_level>[:<dds_level>]	Controls what type of messages are logged. <service_level> is the verbosity level for the service logs and <dds_level> is the verbosity level for the DDS logs. Both can take any of the following values: • `SILENT` • `ERROR` • `WARN` • `LOCAL` • `REMOTE` • `ALL` **Default:** `ERROR:ERROR`
-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 *Configuration* for the set of XML elements that can be overridden with command-line parameters.

7.2 Routing Service Library

Routing Service can be deployed as a library linked into your application on selected architectures (see *Release Notes*). 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.

7.2.1 Example

C

```
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);
```

C++

```
using namespace rti::routing;

ServiceProperty property;
uint32_t running_seconds = 60;
property.cfg_file("my_routing_service_cfg.xml");
property.service_name("my_routing_service");
try {
    Service service(property);
    service.start();
    // Wait for 'running_seconds' seconds
    std::this_thread::sleep_for(std::chrono::seconds(running_seconds));
} catch (const std::exception &ex) {
    /* log error */
    ...
}
```

7.3 Operating System Daemon

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

Chapter 8

Configuration

8.1 Configuring Routing Service

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

8.2 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 8.1 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 8.1: Top-Level Tags in the Configuration File

Tags within <dds>	Description	Multi-plicity
<user_environment>	Assigns default values to XML variables.	0..*
<qos_library>	Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for a *Connext* 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 *Specifying Types*.	0..1
<plugin_library>	Specifies a library of *Routing Service* plugins. Available plug-ins are *Adapters*, *Transformations* and *Processors*. See *Plugins*.	0..*

continues on next page

Table 8.1 – continued from previous page

<table>
<tr><th>Tags within <dds></th><th>Description</th><th>Multiplicity</th></tr>
<tr><td><routing_service></td><td>Specifies a Routing Service configuration. See Routing Service Tag.
Attributes
<code>name</code>: Uniquely identifies a Routing Service configuration. Required.
<code>enabled</code>: 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.
<code>group_name</code>: 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_<code>[Host Name]</code>_<code>[Process ID]</code>
Example<pre><routing_service name="ExampleService">
    <!-- your service settings ... -->
</routing_service></pre></td><td>0..*</td></tr>
</table>

8.2.1 Routing Service Tag

The `<routing_service>` tag is used to configure an execution of *Routing Service*. Configurations may contain multiple `<routing_service>` tags, so you will need to select which *Service* configuration to run (for example with `-cfgName` command-line parameter).

Note that the `<routing_service>` tag is optional. This is allowed so that different aspects of the configurations can be separated in different parts. For example, you could have all the QoS profiles in one file, and all the *Service* configurations in another.

Table 8.2 describes the tags allowed within a `<routing_service>` tag.

Table 8.2: *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 *Administration* and *Remote Administration*.	0..1

continues on next page

Table 8.2 – continued from previous page

Tags within <routing_service>	Description	Multiplicity
<monitoring>	Enables and configures general remote monitoring. General monitoring settings are applicable to all the *Routing Service* entities unless they are explicitly overridden. See *Monitoring* and *Monitoring*.	0..1
<entity_monitoring>	Enables and configures remote monitoring for the service entity. See *Monitoring Configuration Inheritance* and *Monitoring*.	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 *Domain Route*. **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..*

Example: Specifying a configuration in XML

```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`.

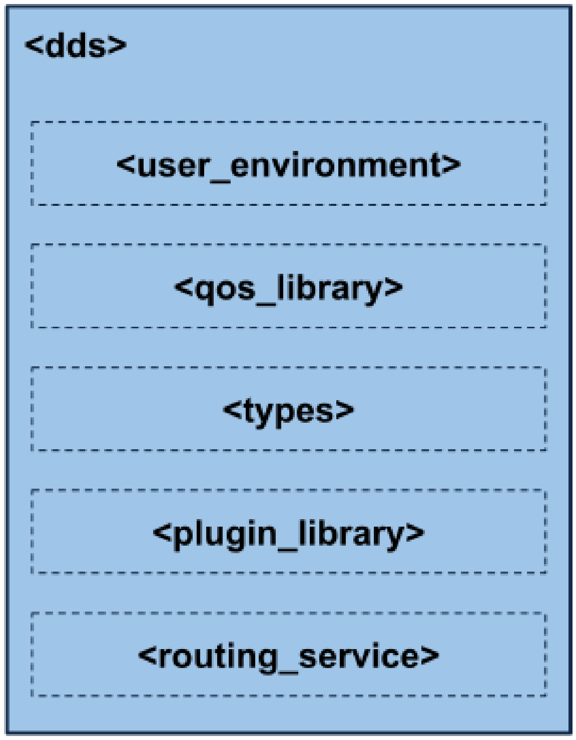


Figure 8.1: Top-level Tags in the Configuration File

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

8.2.2 Administration

You can create a *Connext* 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 *Remote Administration*.

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

Table 8.3: 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

<distributed_logger>	Configures *RTI Distributed Logger*. When you enable it, *Routing Service* will publish its log messages to *Connext*.
	Example:
	`<administration>`
	`    ...`
	`    <distributed_logger>`
	`        <enabled>true</enabled>`
	`    </distributed_logger>`
	`</administration>`
		0..1

<domain_participant_qos>	Configures the *DomainParticipant* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext* defaults.	0..1
<publisher_qos>	Configures the *Publisher* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext* defaults.	0..1
<subscriber_qos>	Configures the *Subscriber* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext* defaults.	0..1
<datareader_qos>	Configures the *DataReader* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext* 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
<datawriter_qos>	Configures the *DataWriter* QoS for remote administration. If the tag is not defined, *Routing Service* will use the *Connext* defaults with the following changes: • history.kind = DDS_KEEP_ALL_HISTORY_QOS • resource_limits.max_samples = 32	0..1

continues on next page

Table 8.3 – continued from previous page

Tags within <administration>	Description	Multiplicity

<memory_management>	Configures certain aspects of how *Connext* 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* allocates memory for samples is as follows: *Connext* 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

<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 (*API Reference*). If the specified file already exists, the file will be overwritten when save is executed. **Default:** [CURRENT DIRECTORY].	0..1
<save_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

continues on next page

Table 8.3 – continued from previous page

Tags within <administration>	Description	Multiplicity
<reuse_monitoring_participant>	Indicates whether the Monitoring participant is reused as the administration participant. If this tag is set to true *and* Monitoring is enabled, the tags `domain_id` and `domain_participant_qos` will be ignored if present. This tag has no effect if Monitoring is disabled or if the service is started in unloaded mode. **Default:** false.	0..1

8.2.3 Monitoring

You can create a *Connext* 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 *Monitoring*.

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

Table 8.4: 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

continues on next page

Table 8.4 – continued from previous page

<table>
<tr><th>Tags within <monitoring></th><th>Description</th><th>Multiplicity</th></tr>
<tr><td><ignore_initialization_failure></td><td>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</td><td>0..1</td></tr>
<tr><td><domain_participant_qos></td><td>Configures the DomainParticipant QoS for remote monitoring. If the tag is not defined, Routing Service will use the Connext defaults, with the following change:
resource_limits.type_code_max_serialized_length = 4096</td><td>0..1</td></tr>
<tr><td><publisher_qos></td><td>Configures the Publisher QoS for remote monitoring. If the tag is not defined, Routing Service will use the Connext defaults.</td><td>0..1</td></tr>
<tr><td><datawriter_qos></td><td>Configures the DataWriter QoS for remote monitoring. If the tag is not defined, Routing Service will use the Connext defaults with the following change:
durability.kind = DDS_TRANSIENT_LOCAL_DURABILITY_QOS</td><td>0..1</td></tr>
<tr><td><statistics_sampling_period></td><td>Specifies the frequency, in seconds, at which status statistics are gathered. Statistical variables such as latency are part of the entity status.
Example:<pre><statistics_sampling_period>
    <sec>1</sec>
    <nanosec>0</nanosec>
</statistics_sampling_period></pre>The statistics period for a given entity should be smaller than the publication period. The statistics sampling period defined in <code><routing_service></code> is inherited by all the entities. An entity can overwrite the period. Default: 1</td><td>0..1</td></tr>
<tr><td><status_publication_period></td><td>Specifies the frequency, in seconds, at which the status of an entity is published.
Example:<pre><status_publication_period>
    <sec>5</sec>
    <nanosec>0</nanosec>
</status_publication_period></pre>The statistics sampling period defined in <code><routing_service></code> is inherited by all the entities. An entity can overwrite the period. Default: 5</td><td>0..1</td></tr>
</table>

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 8.5: Entity Monitoring Tag

<table>
<tr><th>Tags within <entity_monitoring></th><th>Description</th><th>Multiplicity</th></tr>
<tr><td><enabled></td><td>Enables/disables remote monitoring for a given entity. If general monitoring is disabled, this value is ignored.
Default: true</td><td>0..1</td></tr>
<tr><td><statistics_sampling_period></td><td>Specifies the frequency at which status statistics are gathered. Statistical variables such as latency are part of the entity status.
Example:<pre>
<statistics_sampling_period>
    <sec>1</sec>
    <nanosec>0</nanosec>
</statistics_sampling_period>
</pre>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.</td><td>0..1</td></tr>
<tr><td><status_publication_period></td><td>Specifies the frequency at which the status of an entity is published.
Example:<pre>
<status_publication_period>
    <sec>5</sec>
    <nanosec>0</nanosec>
</status_publication_period>
</pre>If this tag is not defined, historical statistics are inherited from the general monitoring settings.
Default: 5 seconds.</td><td>0..1</td></tr>
</table>

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

8.2.4 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 8.6 describes the tags allowed within a `<domain_route>` tag.

Table 8.6: Domain Route Tag

Tags within <domain_route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *DomainRoute*. See *Monitoring*.	0..1

continues on next page

Table 8.6 – continued from previous page

Tags within <domain_route>	Description	Multiplicity
<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 8.7.	0..*
<participant>	Applicable to DDS domains. Configures a DDS adapter *DomainParticipant*. See Table 8.8.	0..*
<session>	Defines a multi-threaded context in which data is routed according to specified routes. See *Session*. **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 8.7: Connection Tag

<table>
<tr><th>Tags within <connection></th><th>Description</th><th>Multiplicity</th></tr>
<tr><td><property></td><td>A sequence of name-value string pairs that allows you to configure the Connection instance.
Example:
<pre>
<property>
    <value>
        <element>
            <name>jms.connection.
↪username</name>
            <value>myusername</value>
        </element>
    </value>
</property>
</pre></td><td>0..1</td></tr>
</table>

continues on next page

Table 8.7 – continued from previous page

Tags within <connection>	Description	Multiplicity
<register_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 *Route*.	0..*

Table 8.8: Participant Tag

<table>
<tr><th>Tags within <participant></th><th>Description</th><th>Multiplicity</th></tr>
<tr><td><domain_id></td><td>Sets the domain ID associated with the DomainParticipant. Default: 0</td><td>0..1</td></tr>
<tr><td><domain_participant_qos></td><td>Sets the participant QoS. The contents of this tag are specified in the same manner as a Connext QoS profile. If not specified, the DDS defaults are used, except for the participant name which takes the following value:

“RTI Routing Service: <app name>.<domain route name>#<participant name>”

where: - <code>app name</code>: The application name of the running Routing Service - <code>domain route</code>: the configuration name of the parent DomainRoute - <code>participant name</code>: the configuration name of the DomainParticipant
For example:

“RTI Routing Service: MyService.MyDomainRoute#domain1”

Note: Changing the default participant name may prevent Routing Service from being detected by Admin Console.

You can use a <code><domain_participant_qos></code> tag inside a <code><qos_library>/<qos_profile></code> previously defined in your configuration file by referring to it, and also override any value:
Example:
<pre>
<domain_participant_qos base_name=
↪"MyLibrary::MyProfile">
    <discovery>
        <initial_peers>
            <element>udpv4://192.168.1.12
↪</element>
            <element>shmem://</element>
        </initial_peers>
    </discovery>
</domain_participant_qos>
</pre>See Configuring QoS with XML, in the Connext DDS Core Libraries User’s Manual.</td><td>0..1</td></tr>
</table>

continues on next page

Table 8.8 – continued from previous page

<table>
<tr><th>Tags within <participant></th><th>Description</th><th>Multi-plicity</th></tr>
<tr><td><memory_management></td><td>Configures certain aspects of how Connext allocates internal memory. The configuration is per DomainParticipant and therefore affects all the contained DDS entities.
Example:<pre>
<memory_management>
    <sample_buffer_min_size>
       1024
    </sample_buffer_min_size>
    <sample_buffer_trim_to_size>
       true
    </sample_buffer_trim_to_size>
</memory_management>
</pre>This tag includes the following tags:
sample_buffer_min_size: For all DataReaders and DataWriters, the way Connext allocates memory for samples is as follows: Connext 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.</td><td>0..1</td></tr>
<tr><td><register_type></td><td>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 Route.</td><td>0..*</td></tr>
</table>

Example: Mapping between Two DDS Domains

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

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

    ...
</domain_route>
```

Example: Mapping between a DDS Domain and raw Sockets

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

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

    ...
</domain_route>
```

8.2.5 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 8.9 lists the tags allowed within a `<session>` tag.

Table 8.9: Session Tag

<table>
<tr><th>Tags within <session></th><th>Description</th><th>Multi-plicity</th></tr>
<tr><td><entity_monitoring></td><td>Enables and configures remote monitoring for the Session. See Monitoring.</td><td>0..1</td></tr>
<tr><td><thread_pool></td><td>Defines the number of threads to process Routes and sets the mask, priority, and stack size of each thread.
Example:
<pre>
<thread_pool>
    <mask>MASK_DEFAULT</mask>
    <priority>THREAD_PRIORITY_DEFAULT</
↪priority>
    <stack_size>
        THREAD_STACK_SIZE_DEFAULT
    </stack_size>
</thread_pool>
</pre>
Default values:

size: 1
mask: MASK_DEFAULT
priority: THREAD_PRIORITY_DEFAULT
stack_size: THREAD_STACK_SIZE_DEFAULT
</td><td>0..1</td></tr>
<tr><td><periodic_action></td><td>Specifies a period at which Processors will receive notifications of the periodic event. This setting represents a default value for all the Routes in this | SESSION|. Default: INFINITE (no periodic notification)
Example:
<pre>
<periodic_action>
    <sec>1</sec>
    <nanosec>0</nanosec>
</periodic_action>
</pre>
The example above indicates the installed Processor should be notified every one second.</td><td>0..1</td></tr>
<tr><td><property></td><td>A sequence of name-value string pairs that allows you to configure the Session instance.
Example:
<pre>
<property>
    <value>
        <element>
            <name>com.rti.socket.timeout
↪</name>
            <value>1</value>
        </element>
    </value>
</property>
</pre>
These properties are only used in non-DDS domains.</td><td>0..1</td></tr>
</table>

continues on next page

Table 8.9 – continued from previous page

Tags within <session>	Description	Multi-plicity
<subscriber_qos>	Only applicable to *Routes* that are *Connext 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* 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* defaults.	0..1
<publisher_qos>	Only applicable to *Routes* that are *Connext 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* 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* 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 *Route*.	0..*
<auto_topic_route> or <auto_route>	Defines a factory for *Route* based on type and stream filters. See *Auto Route*. **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..*

8.2.6 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 8.10 lists the tags allowed within a `<route>`. Table 8.11 lists the tags allowed within a `<topic_route>`.

Table 8.10: Route Tag

Tags within <route>	Description	Multi-plicity
<entity_monitoring>	Enables and configures remote monitoring for the *Route*. See *Monitoring*.	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 *Discovering Types*. **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

<periodic_action>	Specifies a period at which the installed Processor will receive notifications of the periodic event. The *Session* will wake up and notify the installed Processor every specified period. This tag overrides the value set, if any, in the parent *Session*. **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

<enable_data_on_inputs>	Indicates whether this route enables the dispatch of `DATA_ON_INPUTS` event. **Default**: True	0..1

continues on next page

Table 8.10 – continued from previous page

Tags within <route>	Description	Multi-plicity
<processor>	Sets a custom Processor for handling the data forwarding process. See *Software Development Kit*. **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 *Input/Output*. **Attributes** • `name`: uniquely identifies an input configuration. Optional.	0..*
<dds_output>	Only applicable to DDS outputs. Defines an output topic. See *Input/Output*. **Attributes** • `name`: uniquely identifies an output configuration. Optional.	0..*
<input>	Only applicable to non-DDS inputs. Defines an input stream. See *Input/Output*. **Attributes** • `name`: uniquely identifies an input configuration. Optional.	0..*
<output>	Only applicable to non-DDS outputs. Defines an output stream. See *Input/Output*. **Attributes** • `name`: uniquely identifies an output configuration. Optional.	0..*

Table 8.11: Topic Route Tag

Tags within <topic_route>	Description	Multi-plicity
<entity_monitoring>	Enables and configures remote monitoring for the *TopicRoute*. See *Monitoring*.	0..1

continues on next page

Table 8.11 – continued from previous page

Tags within <topic_route>	Description	Multi-plicity
<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 *Discovering Types*. **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

<topic_query_proxy>	Configures the forwarding of *TopicQueries*. See *Topic Query Support* for detailed information on how *Routing Service* processes *TopicQueries*.
	The following tags are used to configure this tag:
	• <enabled>: Whether topic query forwarding is enabled or not. By default, it is disabled.
	• <mode>: How the TopicRoute handles the TopicQueries received from the user DataReaders on the subscription side. There are two modes for handling topic queries: `DISPATCH` and `PROPAGATION`. See *Topic Query Support* for details on each mode. *Default*: `PROPAGATION`.
	The XML 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

continues on next page

Table 8.11 – continued from previous page

Tags within <topic_route>	Description	Multi-plicity

<filter_propagation>	Configures the propagation of content filters. Specifies whether the feature is enabled and when events are processed (*Propagating Content Filters*).
	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 *Propagating Content Filters*).
	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

<periodic_action>	Specifies a period at which the installed Processor will receive notifications of the periodic event. The *Session* will wake up and notify the installed Processor every specified period.|br| This tag overrides the value set, if any, in the parent *Session*. **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

<enable_data_on_inputs>	Indicates whether this route enables the dispatch of `DATA_ON_INPUTS` event. **Default**: True	0..1

continues on next page

Table 8.11 – continued from previous page

Tags within <topic_route>	Description	Multi-plicity
<processor>	Sets a custom Processor for handling the data forwarding process. See *Software Development Kit*. **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 *Input/Output*. **Attributes** • `name`: uniquely identifies an input configuration. Optional.	0..*
<output>	Defines an output topic. See *Input/Output*. **Attributes** • `name`: uniquely identifies an output configuration. Optional.	0..*

8.2.7 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 8.12: Route Input/Output Tags

Tags within <input> and <output> of <route>	Description	Multi-plicity
<entity_monitoring>	Enables and configures remote monitoring for the *Input*/*Output*. See *Monitoring*.	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 *Creation Modes*.	0..1

continues on next page

Table 8.12 – continued from previous page

<table>
<tr><th>Tags within <input> and <output> of <route></th><th>Description</th><th>Multiplicity</th></tr>
<tr><td><on_delete_wait_for_ack_timeout></td><td>Specifies a period for which the StreamWriter will wait for acknowledgment before its elimination. See Waiting for Acknowledgments in a DataWriter, in the Connext DDS Core Libraries User's Manual. Default: 0 (no wait for acknowledgment)
Example:
<pre>
<on_delete_wait_for_ack_timeout>
    <sec>1</sec>
    <nanosec>0</nanosec>
</on_delete_wait_for_ack_timeout>
</pre>
The example above indicates that StreamWriter will wait one second for acknowledgment of the samples.</td><td>0..1 (within <dds_output> only)</td></tr>
<tr><td><property></td><td>A sequence of name-value string pairs that allows you to configure the StreamReader/StreamWriter.
Example:
<pre>
<property>
    <value>
        <element>
            <name>com.rti.socket.port</
↪name>
            <value>16556</value>
        </element>
    </value>
</property>
</pre></td><td>0..1</td></tr>
<tr><td><transformation> (within <output> only)</td><td>Sets a data transformation to be applied for every data sample. See Data Transformation.
Attributes
<code>plugin_name</code>: Name of the plug-in that creates a Transformation object. This name shall refer to a transformation plug-in registered either in a <code><plugin_library></code> or with the service attach_transformation() operation.</td><td>0..1</td></tr>
</table>

Table 8.13: 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

continues on next page

Table 8.13 – continued from previous page

Tags within <input> and <output> (in <topic_route>) and <dds_input> and <dds_output> (in <route>)	Description	Multiplicity
<creation_mode>	Specifies when to create the StreamReader/StreamWriter. **Default:** IMMEDIATE See *Creation Modes*.	0..1

<on_delete_wait_for_ack_timeout>	Specifies a period for which the StreamWriter will wait for acknowledgment before its elimination. See Waiting for Acknowledgments in a DataWriter, in the Connext DDS Core Libraries User's Manual. **Default:** 0 (no wait for acknowledgment)
	Example:
	`<on_delete_wait_for_ack_timeout>`
	`    <sec>1</sec>`
	`    <nanosec>0</nanosec>`
	`</on_delete_wait_for_ack_timeout>`
	The example above indicates that StreamWriter will wait one second for acknowledgment of the samples.
		0..1 (within <output> only)

<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* 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* defaults.	0..1

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

<transformation>`	Sets a data transformation to be applied for every data sample. See *Data Transformation*. **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 8.14: 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.
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 any of the *DomainParticipants* or *Connections* in a *DomainRoute*.

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* XML format for type definitions. See Creating User Data Types with Extensible Markup Language (XML), in the 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">
                ...
                <register_type name="Position" type_ref="PointType"/>
            </connection>
            <participant name="MyParticipant">
                ...
                <register_type name="Position" type_ref="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.

See *Type Registration* for more details about type registration.

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>
        ...
    </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 Library API. See *Software Development Kit*.
2. Instantiate a *Transformation* object by specifying a `<transformation>` tag inside a `<output>` or `<dds_output>`.

Table 8.15 lists the tags allowed within a `<transformation>` tag.

Table 8.15: 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

<output_type_name>	Available only when the transformation is set in an <input>. Specifies the registered type name of the output samples. If not specified, this tag is set to the registered type name of the first output that has no transformation.	0..1
<output_connection_name>	Available only when the transformation is set in an <input>. Name of the <connection>/<participant> from which the registered type must be obtained. If not specified, the type will be obtained from the same connection of the parent Input or the first connection that the type is available.	0..1
<input_type_name>	Available only when the transformation is set in an <output>. Specifies the registered type name of the input samples. If not specified, this tag is set to the registered type name of the first input that has no transformation.	0..1
<input_connection_name>	Available only when the transformation is set in an <output>. Name of the <connection>/<participant> from which the registered type must be obtained. If not specified, the type will be obtained from the same connection of the parent Output or the first connection that the type is available.	0..1

8.2.8 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 8.16 lists the tags allowed within a <auto_route>.
- Table 8.17 lists the tags allowed within a <auto_topic_route>.

Table 8.16: AutoRoute Tag

Tags within <auto_route>	Description	Multiplicity
<entity_monitoring>	Enables and configures remote monitoring for the *AutoRoute*. See *Monitoring*.	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
<periodic_action>	Specifies a period at which the installed Processor will receive notifications of the periodic event. The *Session* will wake up and notify the installed Processor every specified period.|br| This tag overrides the value set, if any, in the parent *Session*. **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
<enable_data_on_inputs>	Indicates whether this route enables the dispatch of `DATA_ON_INPUTS` event. **Default**: True	0..1

continues on next page

Table 8.16 – continued from previous page

Tags within <auto_route>	Description	Multi-plicity
<processor>	Sets a custom Processor for handling the data forwarding process. See *Software Development Kit*. **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 8.17: AutoTopicRoute Tag

Tags within <auto_topic_route>	Description	Multi-plicity
<entity_monitoring>	Enables and configures remote monitoring for the *AutoTopicRoute*. See *Monitoring*.	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

continues on next page

Table 8.17 – continued from previous page

<table>
<tr><th>Tags within <auto_topic_route></th><th>Description</th><th>Multiplicity</th></tr>
<tr><td><topic_query_proxy></td><td>Configures the forwarding of TopicQueries. See Topic Query Support 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:<pre>
<topic_query_proxy>
    <enabled>true</enabled>
    <mode>PROPAGATION</mode>
</topic_query_proxy>
</pre></td><td>0..1</td></tr>
<tr><td><filter_propagation></td><td>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 Propagating Content Filters).
Example:<pre>
<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>
</pre></td><td>0..1</td></tr>
<tr><td><periodic_action></td><td>Specifies a period at which the installed Processor will receive notifications of the periodic event. The Session will wake up and notify the installed Processor every specified period.|br| This tag overrides the value set, if any, in the parent Session. Default: INFINITE (no periodic notification)
Example:<pre>
<periodic_action>
    <sec>1</sec>
    <nanosec>0</nanosec>
</periodic_action>
</pre>The example above indicates the installed Processor should be notified every one second.</td><td>0..1</td></tr>
<tr><td><enable_data_on_inputs></td><td>Indicates whether this route enables the dispatch of <code>DATA_ON_INPUTS</code> event. Default: True</td><td>0..1</td></tr>
</table>

continues on next page

Table 8.17 – continued from previous page

Tags within <auto_topic_route>	Description	Multiplicity
<processor>	Sets a custom Processor for handling the data forwarding process. See *Software Development Kit*. **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 8.18: 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 *Monitoring*.	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
<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

<on_delete_wait_for_ack_timeout>	Specifies a period for which the StreamWriter will wait for acknowledgment before its elimination. See Waiting for Acknowledgments in a DataWriter, in the Connext DDS Core Libraries User's Manual. **Default:** 0 (no wait for acknowledgment)
	Example:
	`<on_delete_wait_for_ack_timeout>`
	`    <sec>1</sec>`
	`    <nanosec>0</nanosec>`
	`</on_delete_wait_for_ack_timeout>`
	The example above indicates that StreamWriter will wait one second for acknowledgment of the samples.
		0..1 (within <dds_output> only)

<creation_mode>	Specifies when to create the StreamReader/StreamWriter. **Default:** IMMEDIATE See *Creation Modes*.	0..1

continues on next page

Table 8.18 – continued from previous page

Tags within <input> and <output> of <auto_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

Table 8.19: 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 *Monitoring*.	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
<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

continues on next page

Table 8.19 – continued from previous page

Tags within <input> and <output> (in <auto_topic_route>) <dds_input> and <dds_output> (in <auto_route>)	Description	Multiplicity

<on_delete_wait_for_ack_timeout>	Specifies a period for which the StreamWriter will wait for acknowledgment before its elimination. See Waiting for Acknowledgments in a DataWriter, in the Connext DDS Core Libraries User's Manual. **Default:** 0 (no wait for acknowledgment)
	Example:
	`<on_delete_wait_for_ack_timeout>`
	`    <sec>1</sec>`
	`    <nanosec>0</nanosec>`
	`</on_delete_wait_for_ack_timeout>`
	The example above indicates that StreamWriter will wait one second for acknowledgment of the samples.
		0..1 (within <output> only)

<creation_mode>	Specifies when to create the StreamReader/StreamWriter. **Default:** IMMEDIATE See *Creation Modes*.	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* 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* defaults.	0..1

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

8.2.9 Plugins

All the pluggable components specific to *Routing Service* are configured within the `<plugin_library>` tag. Table 8.20 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 *Software Development Kit* for further information on developing *Routing Service* plug-ins.

Table 8.20: Configuration tags for plug-in libraries

Tags within <plugin_library>	Description	Multiplicity
<adapter_plugin>	Specifies a C/C++ *Adapter* plug-in. See Table 12.18. **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 8.6.	0..*
<java_adapter_plugin>	Specifies a Java *Adapter* plug-in. See Table 12.19. **Attributes** (See `<adapter_plugin>`)	0..*
<transformation_plugin>	Specifies a C/C++ *Transformation* plug-in. See Table 12.18. **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 *Route*.	0..*
<processor_plugin>	Specifies a C/C++ *Processor* plug-in. See Table 12.18. **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 *Route*.	0..*

8.3 Enabling Distributed Logger

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

Distributed Logger is included in *Connext* 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*. 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 *Administration*. For more details about *Distributed Logger*, see Enabling Distributed Logger in RTI Services, in the Connext DDS Core Libraries User's Manual.

8.4 Support for Extensible Types

Routing Service includes partial support for the "Extensible and Dynamic Topic Types for DDS" specification from the Object Management Group (OMG). This section assumes that you are familiar with Extensible Types and you have read the *Connext Extensible Types Guide*.

- *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 Extensible Types Guide*.
- 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 returned in 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*.

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

```
struct A {
  long x;
};

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

Table 8.21: 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 8.22: 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]

8.5 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 Sending Large Data, in the Connext DDS User Libraries User's Manual.

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

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

(continues on next page)

(continued from previous page)

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

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

    <routing_service name="FlatDataWithZeroCopy">

        <domain_route>
            <participant name="InputDomain">
                <domain_id>0</domain_id>
                <domain_participant_qos base_name="MyQosLib::ShmemOnly"/>
                <register_type name="Point" type_ref="Point"/>
            </participant>
            <participant name="OutputDomain">
                <domain_id>1</domain_id>
                <register_type name="Point" type_ref="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>
```

8.6 Support for RTI Security Plugins

Routing Service supports configuring and using *RTI Security Plugins*. To configure *Routing Service* securely, you need to configure the appropriate QoS settings in the XML configuration. For more information, see the RTI Security Plugins User's Manual.

8.6.1 Example: Configuring a Routing Service Instance using Security

The following example in XML demonstrates how to configure Routing Service to load and use the *RTI Security Plugins*. The example assumes a path where the user has created the necessary security artifacts (such as permissions files, certificates, and certificate authorities). This path is represented by the SECURITY_ARTIFACTS_PATH environment variable.

```
<dds>
    <qos_library name="SecureQosLibrary">
        <qos_profile name="SecureParticipantQos">
            <domain_participant_qos>
                <property>
                    <value>
                        <element>
                            <name>com.rti.serv.load_plugin</name>
                            <value>com.rti.serv.secure</value>
                        </element>
                        <element>
                            <name>com.rti.serv.secure.library</name>
                            <value>nddssecurity</value>
                        </element>
                        <element>
                            <name>com.rti.serv.secure.create_function</name>
                            <value>RTI_Security_PluginSuite_create</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.identity_ca</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/ca/
↪ecdsa01RootCaCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.identity_certificate</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/
↪identities/ecdsa01RoutingServiceCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.private_key</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/
↪identities/ecdsa01RoutingServiceKey.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.access.permissions_ca</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/ca/
↪ecdsa01RootCaCert.pem</value>
```

(continues on next page)

(continued from previous page)

```
                        </element>
                        <element>
                            <name>dds.sec.access.governance</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/signed_
↪Governance.p7s</value>
                        </element>
                        <element>
                            <name>dds.sec.access.permissions</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/signed_
↪PermissionsA.p7s</value>
                        </element>
                    </value>
                </property>
            </domain_participant_qos>
        </qos_profile>
    </qos_library>

    ...

    <routing_service name="SecureToUnsecureCommunication">
        <domain_route name="DomainRoute1">
            <participant name="1">
                <domain_id>1</domain_id>
                <!-- Domain Participant in Domain 1 is secured -->
                <domain_participant_qos base_name="SecureQosLibrary::SecureParticipantQos
↪" />
            </participant>
            <participant name="2">
                <domain_id>2</domain_id>
                <!-- Domain Participant in Domain 2 is not secured -->
                <domain_participant_qos base_name="DefaultQosLibrary::DefaultQos
↪" />
            </participant>
            <session name="S1">
                <topic_route name="SecureToUnsecure">
                    <route_types>true</route_types>
                    <input participant="1">
                        <topic_name>Topic01_Secure</topic_name>
                        <registered_type_name>...</registered_type_name>
                        <datareader_qos base_name="DefaultQosLibrary::DefaultQos
↪"/>
                    </input>
                    <output>
                        <topic_name>Topic01_Unsecure</topic_name>
                        <registered_type_name>...</registered_type_name>
                        <datawriter_qos base_name="DefaultQosLibrary::DefaultQos
↪"/>
                    </output>
                </topic_route>
            </session>
        </domain_route>
    </routing_service>
```

(continues on next page)

(continued from previous page)

```
</dds>
```

The above XML example configures a *Domain Route* that moves data from a secured *DomainParticipant* into an unsecure *DomainParticipant*. The security settings are encapsulated in a QoS Profile called SecureParticipantQos. When secured data reaches the secured endpoint, the *Routing Service* instance performs all security operations that will be incorporated in the cleartext sample moving into the other end of the *Topic Route*. The data is then published into the unsecured domain.

8.6.2 Example: Configuring Routing Service to use a Certificate Revocation List (CRL)

Routing Service can remove a *DomainParticipant* from the system when its certificate has been revoked. Use *RTI Security Plugins* to specify a CRL (Certificate Revocation List) file to track via the `authentication.crl` property; when the `authentication.crl_file_poll_period.millisec` property is configured in *Security Plugins*, *Routing Service* can banish revoked participants. For more information, see Properties for Configuring Authentication in the *RTI Security Plugins User's Manual*. The following example XML configuration file uses a CRL file to enable *Routing Service* to remove participants with revoked certificates.

```
<dds>
    <qos_library name="SecureQosLibrary">
        <qos_profile name="SecureParticipantQos">
            <domain_participant_qos>
                <property>
                    <value>
                        <element>
                            <name>com.rti.serv.load_plugin</name>
                            <value>com.rti.serv.secure</value>
                        </element>
                        <element>
                            <name>com.rti.serv.secure.library</name>
                            <value>nddssecurity</value>
                        </element>
                        <element>
                            <name>com.rti.serv.secure.create_function</name>
                            <value>RTI_Security_PluginSuite_create</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.identity_ca</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/ca/
↪ecdsa01RootCaCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.identity_certificate</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/
↪identities/ecdsa01RoutingServiceCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.private_key</name>
```

(continues on next page)

(continued from previous page)

```
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/
↪identities/ecdsa01RoutingServiceKey.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.access.permissions_ca</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/ca/
↪ecdsa01RootCaCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.access.governance</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/signed_
↪Governance.p7s</value>
                        </element>
                        <element>
                            <name>dds.sec.access.permissions</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/signed_
↪PermissionsA.p7s</value>
                        </element>
                    </value>
                </property>
            </domain_participant_qos>
        </qos_profile>
        <qos_profile name="SecureParticipantQosWithCrl" base_name=
↪"SecureQosLibrary::SecureParticipantQos">
            <domain_participant_qos>
                <property>
                    <value>
                        <element>
                            <name>com.rti.serv.secure.authentication.crl</
↪name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/
↪RoutingServiceRevoked.crl</value>
                        </element>
                        <element>
                            <name>com.rti.serv.secure.authentication.crl_file_
↪poll_period.millisec</name>
                            <value>500</value>
                        </element>
                    </value>
                </property>
            </domain_participant_qos>
        </qos_profile>
    </qos_library>

    ...

    <routing_service name="SecureToUnsecureCommunication">
        <domain_route name="DomainRoute1">
            <participant name="1">
                <domain_id>1</domain_id>
                <!-- Domain Participant in Domain 1 is secured -->
                <domain_participant_qos base_name="SecureQosLibrary::SecureParticipantQosW
↪" />
```

(continues on next page)

(continued from previous page)

```
            </participant>
            <participant name="2">
                <domain_id>2</domain_id>
                <!-- Domain Participant in Domain 2 is not secured -->
                <domain_participant_qos base_name="DefaultQosLibrary::DefaultQos
↪" />
            </participant>
            <session name="S1">
                <topic_route name="SecureToUnsecure">
                    <route_types>true</route_types>
                    <input participant="1">
                        <topic_name>Topic01_Secure</topic_name>
                        <registered_type_name>...</registered_type_name>
                        <datareader_qos base_name="DefaultQosLibrary::DefaultQos
↪"/>
                    </input>
                    <output>
                        <topic_name>Topic01_Unsecure</topic_name>
                        <registered_type_name>...</registered_type_name>
                        <datawriter_qos base_name="DefaultQosLibrary::DefaultQos
↪"/>
                    </output>
                </topic_route>
            </session>
        </domain_route>
    </routing_service>

</dds>
```

The above configuration in *Routing Service* reads the CRL file `$SECURITY_ARTIFACTS_PATH/RoutingServiceRevoked.crl`. In addition, the `authentication.crl_file_poll_period.millisec` element instructs the service to track the file for changes so that participants can be removed dynamically. The polling of the file happens every 500ms.

Note: If you do not specify the poll period, or if the poll period is zero, *Routing Service* will not track the file continuously.

8.6.3 Example: Configuring Routing Service for Dynamic Certificate Renewal

Routing Service can dynamically renew its certificate if it was revoked or it expired. Use *RTI Security Plugins* to specify a periodic check of the certificate file; when this property is configured in *Security Plugins*, *Routing Service* reloads the certificate if the file changes. For more information, see the RTI Security Plugins User's Manual.

The following example XML configuration file defines a 500ms period for checking the certificate file for changes.

```
<dds>
    <qos_library name="SecureQosLibrary">
        <qos_profile name="SecureParticipantQos">
            <domain_participant_qos>
                <property>
                    <value>
                        <element>
                            <name>com.rti.serv.load_plugin</name>
                            <value>com.rti.serv.secure</value>
                        </element>
                        <element>
                            <name>com.rti.serv.secure.library</name>
                            <value>nddssecurity</value>
                        </element>
                        <element>
                            <name>com.rti.serv.secure.create_function</name>
                            <value>RTI_Security_PluginSuite_create</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.identity_ca</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/ca/
↪ecdsa01RootCaCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.identity_certificate</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/
↪identities/ecdsa01RoutingServiceCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.auth.private_key</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/
↪identities/ecdsa01RoutingServiceKey.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.access.permissions_ca</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/ecdsa01/ca/
↪ecdsa01RootCaCert.pem</value>
                        </element>
                        <element>
                            <name>dds.sec.access.governance</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/signed_
↪Governance.p7s</value>
                        </element>
                        <element>
                            <name>dds.sec.access.permissions</name>
                            <value>$(SECURITY_ARTIFACTS_PATH)/signed_
↪PermissionsA.p7s</value>
                        </element>
                    </value>
                </property>
            </domain_participant_qos>
        </qos_profile>
```

(continues on next page)

(continued from previous page)

```
        <qos_profile name="SecureParticipantQosDynamicCert" base_name=
↪"SecureQosLibrary::SecureParticipantQos">
            <domain_participant_qos>
                <property>
                    <value>
                        <element>
                            <name>com.rti.serv.secure.authentication.identity_
↪certificate_file_poll_period.millisec</name>
                            <value>500</value>
                        </element>
                    </value>
                </property>
            </domain_participant_qos>
        </qos_profile>
    </qos_library>

    ...

    <routing_service name="SecureToUnsecureCommunication">
        <domain_route name="DomainRoute1">
            <participant name="1">
                <domain_id>1</domain_id>
                <!-- Domain Participant in Domain 1 is secured -->
                <domain_participant_qos base_name="SecureQosLibrary::SecureParticipantQosDy
↪" />
            </participant>
            <participant name="2">
                <domain_id>2</domain_id>
                <!-- Domain Participant in Domain 2 is not secured -->
                <domain_participant_qos base_name="DefaultQosLibrary::DefaultQos
↪" />
            </participant>
            <session name="S1">
                <topic_route name="SecureToUnsecure">
                    <route_types>true</route_types>
                    <input participant="1">
                        <topic_name>Topic01_Secure</topic_name>
                        <registered_type_name>...</registered_type_name>
                        <datareader_qos base_name="DefaultQosLibrary::DefaultQos
↪"/>
                    </input>
                    <output>
                        <topic_name>Topic01_Unsecure</topic_name>
                        <registered_type_name>...</registered_type_name>
                        <datawriter_qos base_name="DefaultQosLibrary::DefaultQos
↪"/>
                    </output>
                </topic_route>
            </session>
        </domain_route>
    </routing_service>
```

(continues on next page)

(continued from previous page)

```
</dds>
```

The above configuration in *Routing Service* periodically (every 500ms) checks the *DomainParticipant* certificate file `$SECURITY_ARTIFACTS_PATH/ecdsa01/identities/ecdsa01RoutingServiceCert.pem` for changes.

Note: If you do not specify the poll period, or if the poll period is zero, *Routing Service* will not track the file continuously.

Chapter 9

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* Library 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 *Routing Service Library*.
- *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*, 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 *Library 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 *Library 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 Library API.

Table 9.1 shows which modules are available for each API, along with links to the API documentation.

Table 9.1: API Documentation for the SDK

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

Chapter 10

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.

10.1 Resource Model

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

Figure 10.1 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 metadata. A resource in this plane is also known as an *entity*. The data provision and processing is performed using *plugins* (see *Software Development Kit* 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 10.2.

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.

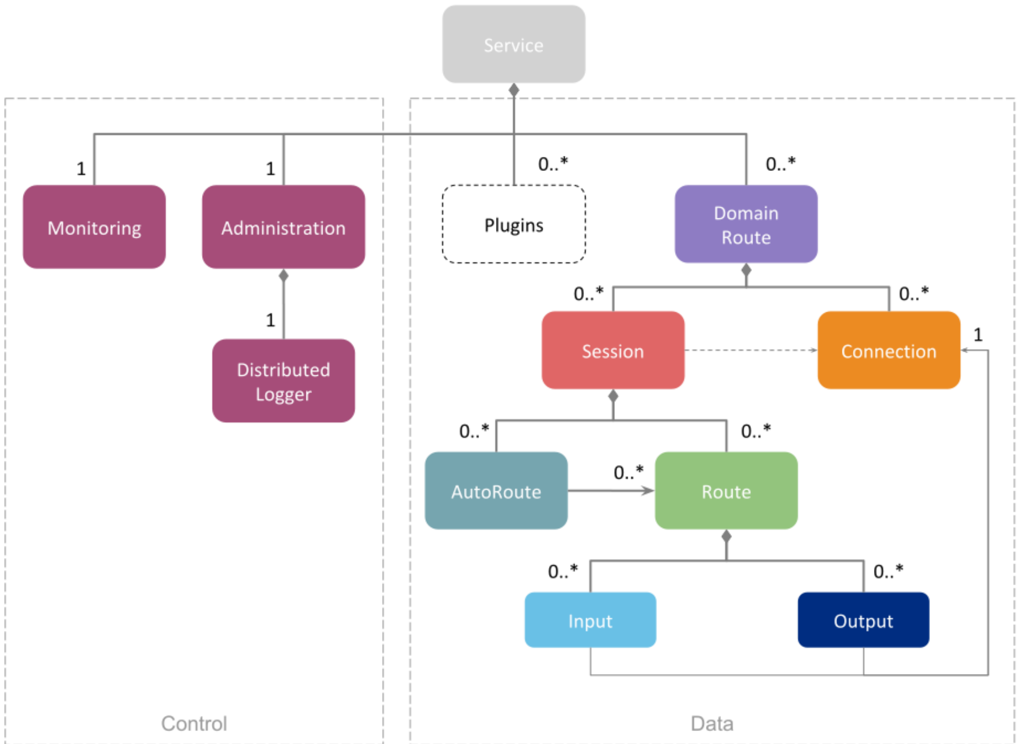


Figure 10.1: *Routing Service* Application Resource Model

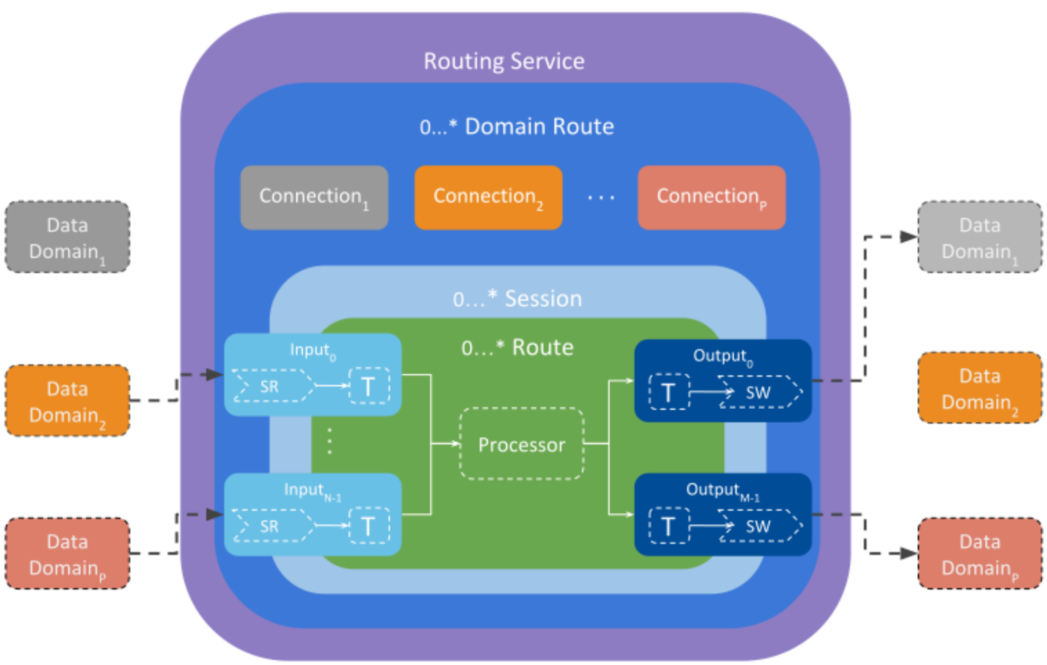


Figure 10.2: *Routing Service* Alternative representation of the Application Resource Model

- 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 (*DDS Adapter*). It's recommended though that you still review the general model for a solid understanding of *Routing Service*.

10.1.1 Directory

Table 10.1 provides a resource directory with quick links to access different types of information for each resource or entity.

Table 10.1: Resource Reference

Resource	Configuration	Administration	Monitoring
Service	*Routing Service Tag*	*Service*	*Service*
DomainRoute	*Domain Route*	*DomainRoute*	*DomainRoute*
Connection	*Domain Route*	*Connection*	*DomainRoute*
Session	*Session*	*Session*	*Session*
Route	*Route*	*Route*	*Route*
Input	*Input/Output*	*Input/Output*	*Input/Output*
Output	*Input/Output*	*Input/Output*	*Input/Output*

10.1.2 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 *Monitoring* and *Remote Administration*, 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 *Software Development Kit*). Figure 10.3 shows the relationship between the *Service* and the plugin objects.

See *Plugin Management* for more information about plugin management.

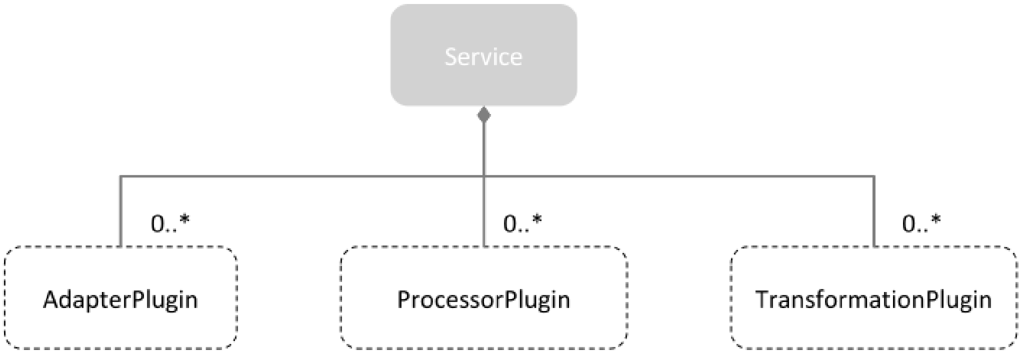


Figure 10.3: *Routing Service* composed of different plugins

Service States

A *Service* can be in one of the states listed in Table 10.2.

Table 10.2: 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 Library API (see *Usage*). • 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

continues on next page

Table 10.2 – 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

10.1.3 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*.

DomainRoute States

A *DomainRoute* can be in one of the states listed in Table 10.3.

Table 10.3: *DomainRoute* 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 (*Service States*) • Remote command	N/A

continues on next page

Table 10.3 – 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 (*Service States*) • 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 (*Service States*) • 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

10.1.4 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 8.7 and Table 8.8). 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 10.4 shows the relationship with the plugin objects. A *Connection* shall hold one, and only one, `adapter::Connection` object.

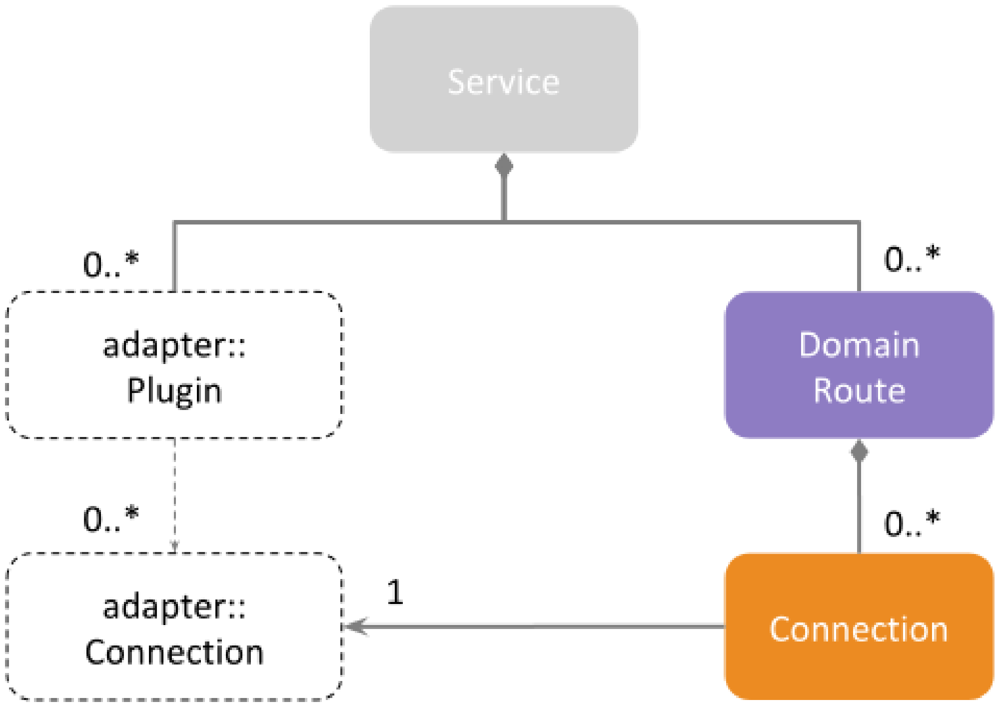


Figure 10.4: Relationship of plugins with a *Connection*

Connection States

A *Connection* can be in one of the states listed in Table 10.4.

Table 10.4: *Connection* states

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

continues on next page

Table 10.4 – continued from previous page

State	Description	Trigger	Plugin callback
`DISABLED`	A *Connection* object has deleted the *Adapter* connection object it holds.	• *DomainRoute* stops (*DomainRoute States*)	`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:

- **type registered 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 *Defining Types in the Configuration File*). 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.

10.1.5 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 10.5 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 *Route* for information about route processing). To prevent starvation, new events arriving will be deferred for the next selection cycle.

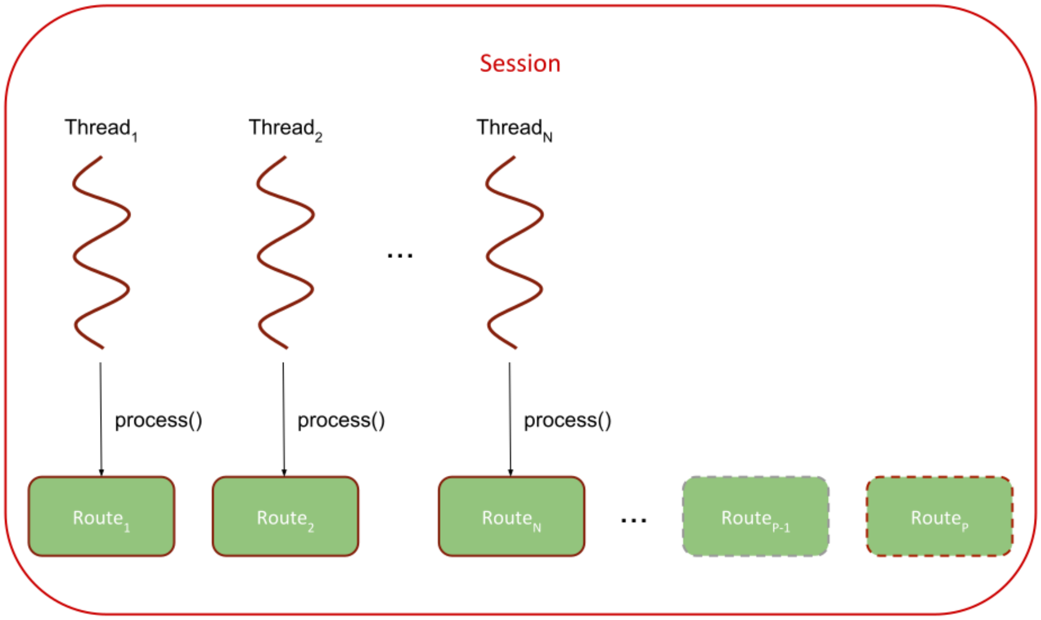


Figure 10.5: Processing mechanism of *Routes* within a *Session*

- 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 10.5 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; Route$_{P-1}$ is not active (no pending events).

Plugin Interaction

Figure 10.6 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 10.5.

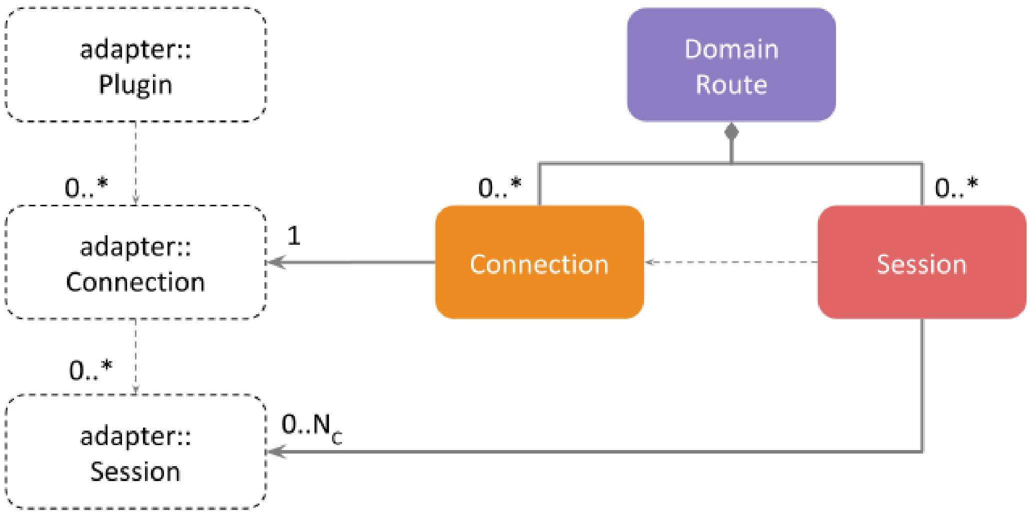


Figure 10.6: Relationship of plugins with a *Session*

Table 10.5: *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 (*DomainRoute States*) • 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 (*DomainRoute States*) • Remote command	N/A
`STOPPED`	A *Session* object has stopped the thread pool, and disabled all the underlying *AutoRoutes* and *Routes*.	• *DomainRoute* stops (*DomainRoute States*) • Remote command	N/A

continues on next page

Table 10.5 – continued from previous page

State	Description	Trigger	Plugin callback
DISABLED	A *Session* object has deleted all the adapter::Session objects it holds.	• *DomainRoute* stops (*DomainRoute States*) • Remote command	Connection::delete_session

10.1.6 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 10.7 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 (*Software Development Kit*).

Route States

A *Route* state machine is shown in Figure 10.8.

Table 10.6 shows all the states a *Route* can enter.

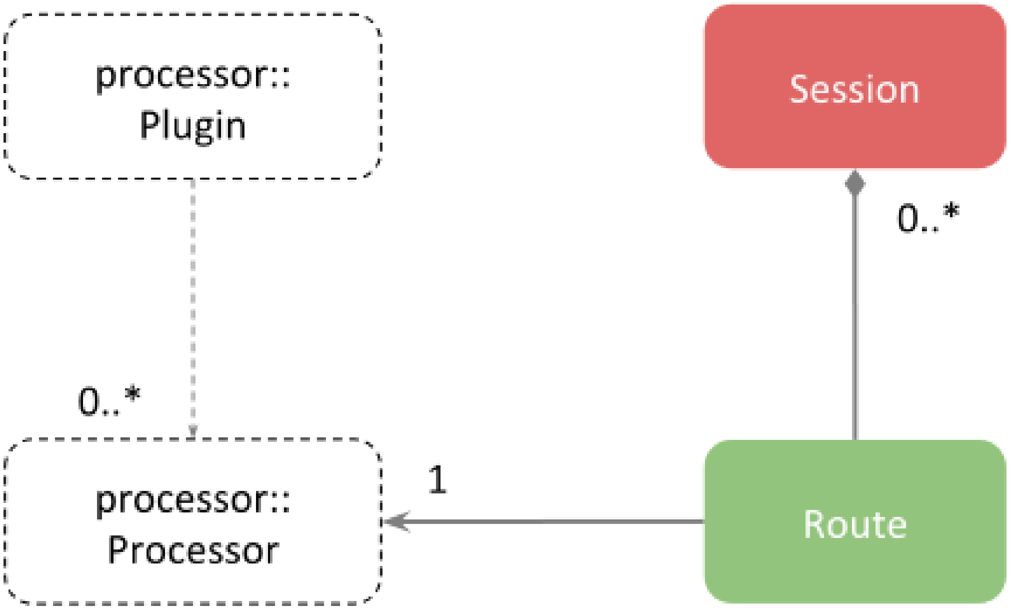


Figure 10.7: Relationship of plugins with a *Route*

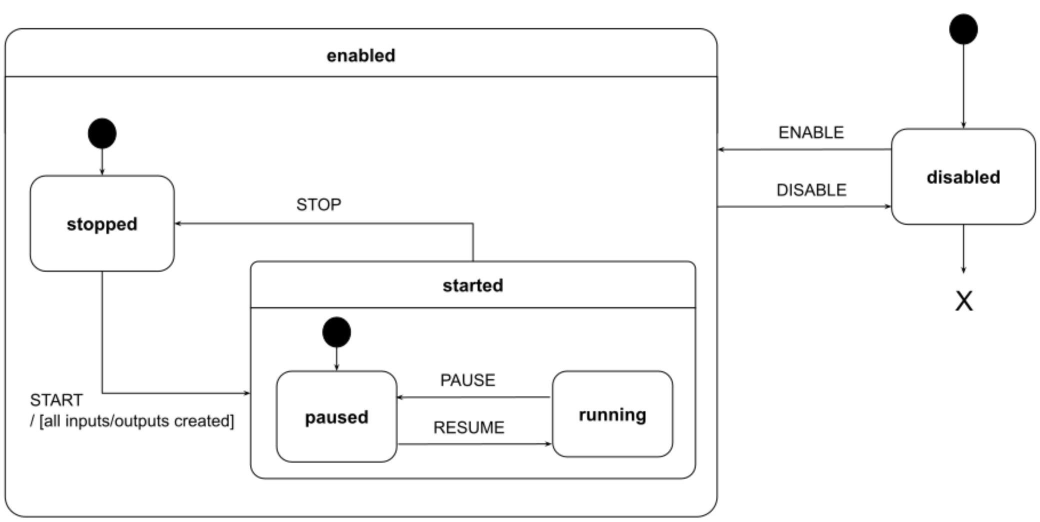


Figure 10.8: *Route* state machine

Table 10.6: *Route* states

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

continues on next page

Table 10.6 – continued from previous page

State	Description	Trigger	Plugin callback
STOPPED	A *Route* has disabled at least one of its *Inputs* and *Outputs*.	• *Session* stops (*Session States*) • Disable *Input* (*Input States*) or *Output* (*Output States*) • 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 (*Session States*) • Enable *Input* (*Input States*) or *Output* (*Output States*) • Remote command	• `Processor::on_route_event` • `StreamReader::read` • `StreamReader::return_loan` • `Transformation::transform` • `Transformation::return_loan` • `StreamWriter::write`
PAUSED	A *Route* is temporarily suspending the processing of data stream related events.	• *Session* stops • Disable *Input* (*Input States*) or *Output* (*Output States*) • Remote command	`Processor::on_route_event`

10.1.7 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* per stream name:

$$[S_m]$$

where S_m is the name for the *stream* `m`. The name of the type T_m, only plays a role while passing the filter criteria and while creating the generated *Route*'s *Input* and *Output*.

This means a *stream* name that is shared by multiple type names won't spawn a new *Route* from an *AutoRoute* per type name. Rather it will reuse the first one generated for the shared *stream* name.

Note: It is not advised to have *streams* share the same name but have different type names when using an *AutoRoute*. It can lead to a situation where the *Input* and *Output* discover *streams* with different type names, leading to an incompatible *Route* creation - especially when dealing with the builtin DDS *Adapter*. In such a situation it is better to bifurcate the *streams* based on their stream name.

In the case of the builtin DDS *Adapter*, if the two types under the same topic (stream) name are compatible as per the rules of Extensible Types, only then will data be successfully routed by the common generated *Route*.

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 10.9 illustrates this relationship.

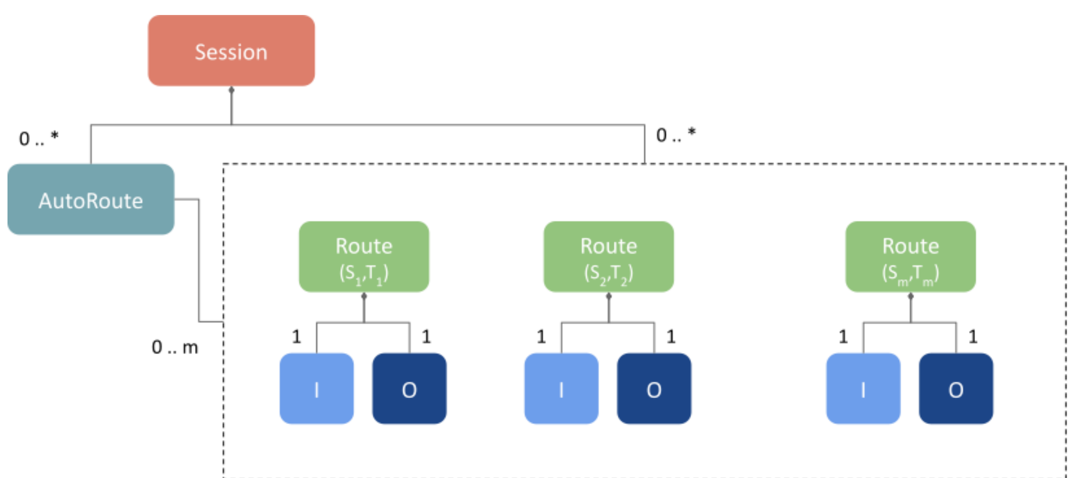


Figure 10.9: *AutoRoute* as a map of *Routes* keyed by stream name

The *AutoRoute* creates a *Route* only if it has not previously matched S_m. *AutoRoutes* never delete the created *Route*, regardless 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 10.7.

Table 10.7: *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 (*Session States*) • 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 (*Session States*)	N/A

10.1.8 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 10.10 shows the relationship with the plugin objects. An *Input* shall hold one, and only one, `adapter::StreamReader` object. Optionally, an *Input* may hold one and only `transformation::Transformation` instance, that is applied to the sample stream returned by the `adapter::StreamReader`.

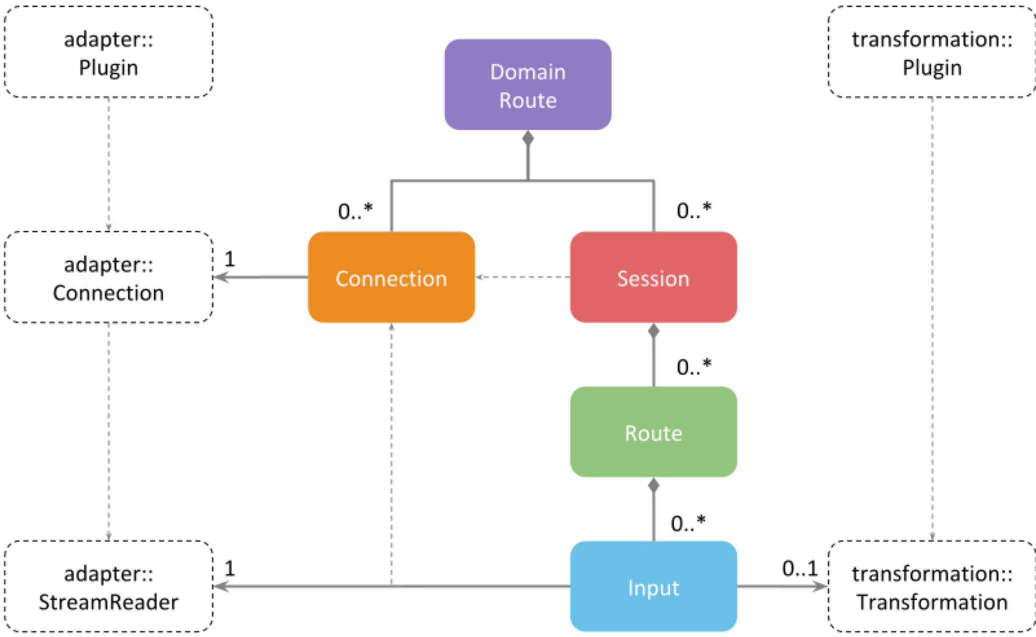


Figure 10.10: Relationship of plugins with an *Input*

The *Input* obtains data from a domain by calling the `StreamReader::read` operation. If a *Transformation* is present, the `Transformation::transform` operation is called right after reading from the *StreamReader*. The `Transformation::return_loan` is called when the obtained loaned samples are returned.

Input States

An *Input* can be in one of the states listed in Table 10.8.

Table 10.8: *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_reader` • `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_rea` • `Processor::on_route_event`

10.1.9 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 10.11 shows the relationship with the plugin objects. An *Output* shall hold one, and only one, `adapter::StreamWriter` object. Optionally, an *Input* may hold one and only `transformation::Transformation` instance, that is applied to a sample stream before is passed to the `adapter::StreamWriter`.

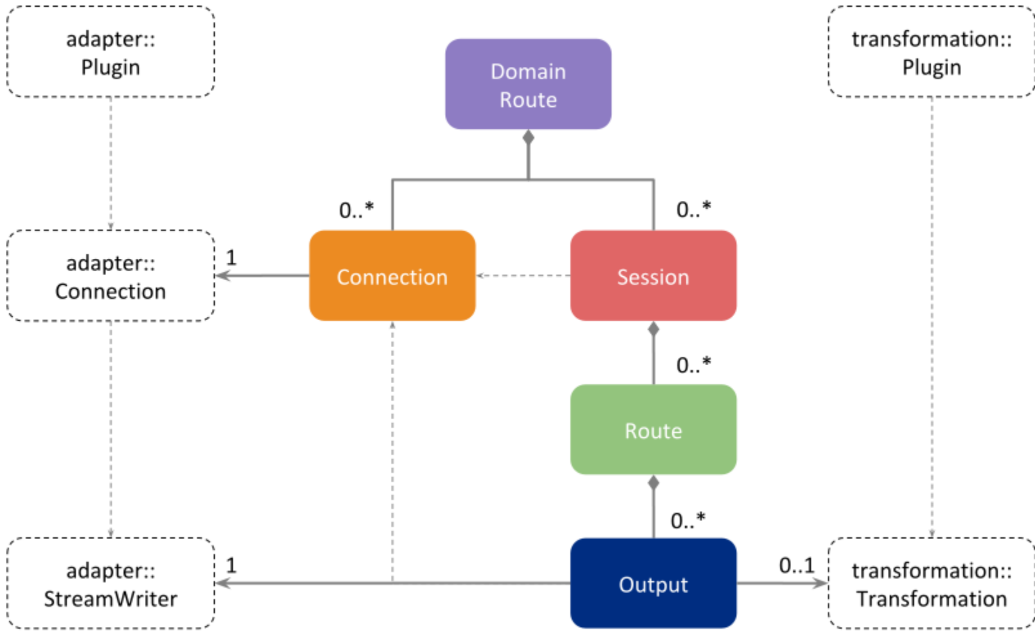


Figure 10.11: Relationship of plugins with an *Output*

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.

Output States

An *Output* can be in one of the states listed in Table 10.9.

Table 10.9: *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_writer` • `Processor::on_route_event` • `TransformationPlugin::new` (only once for each plugin class) • `TransformationPlugin::create_transformation`
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

continues on next page

Table 10.9 – continued from previous page

State	Description	Trigger	Plugin callback
`DISABLED`	*Output* has deleted its underlying *StreamWriter* and can no longer write data.	Creation mode condition becomes false	• `Connection::delete_stream_wri` • `TransformationPlugin::delete_transformation` • `Processor::on_route_event`

10.2 Builtin plugins

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

10.2.1 DDS Adapter

This is an *Adapter* implementation that provides access to DDS domains. Figure 10.12 shows the architecture of the DDS *Adapter*.

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*

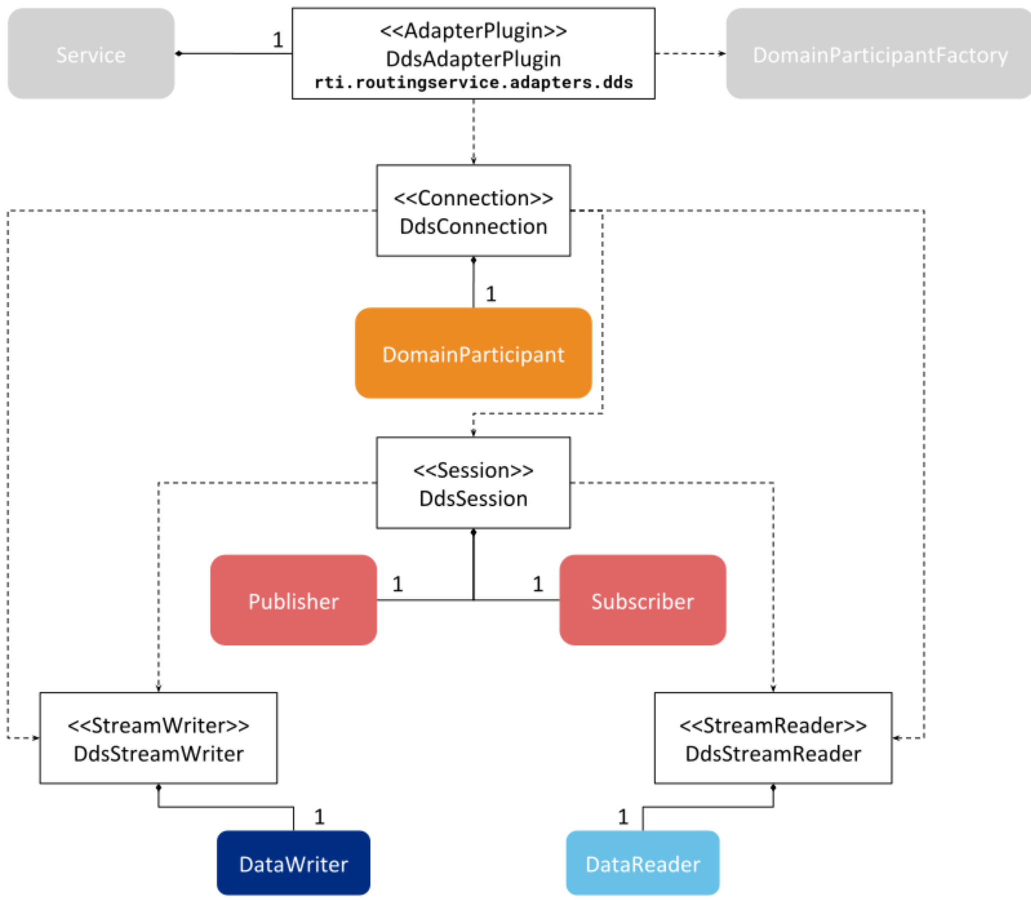


Figure 10.12: DDS *Adapter* architecture

These entities are equivalent to the generic entities shown in Figure 10.1 except that the *Adapter* entity they enclose is created from the builtin DDS *Adapter* (*DDS Adapter*). Figure 10.13 shows the DDS specialization of the generic resource model.

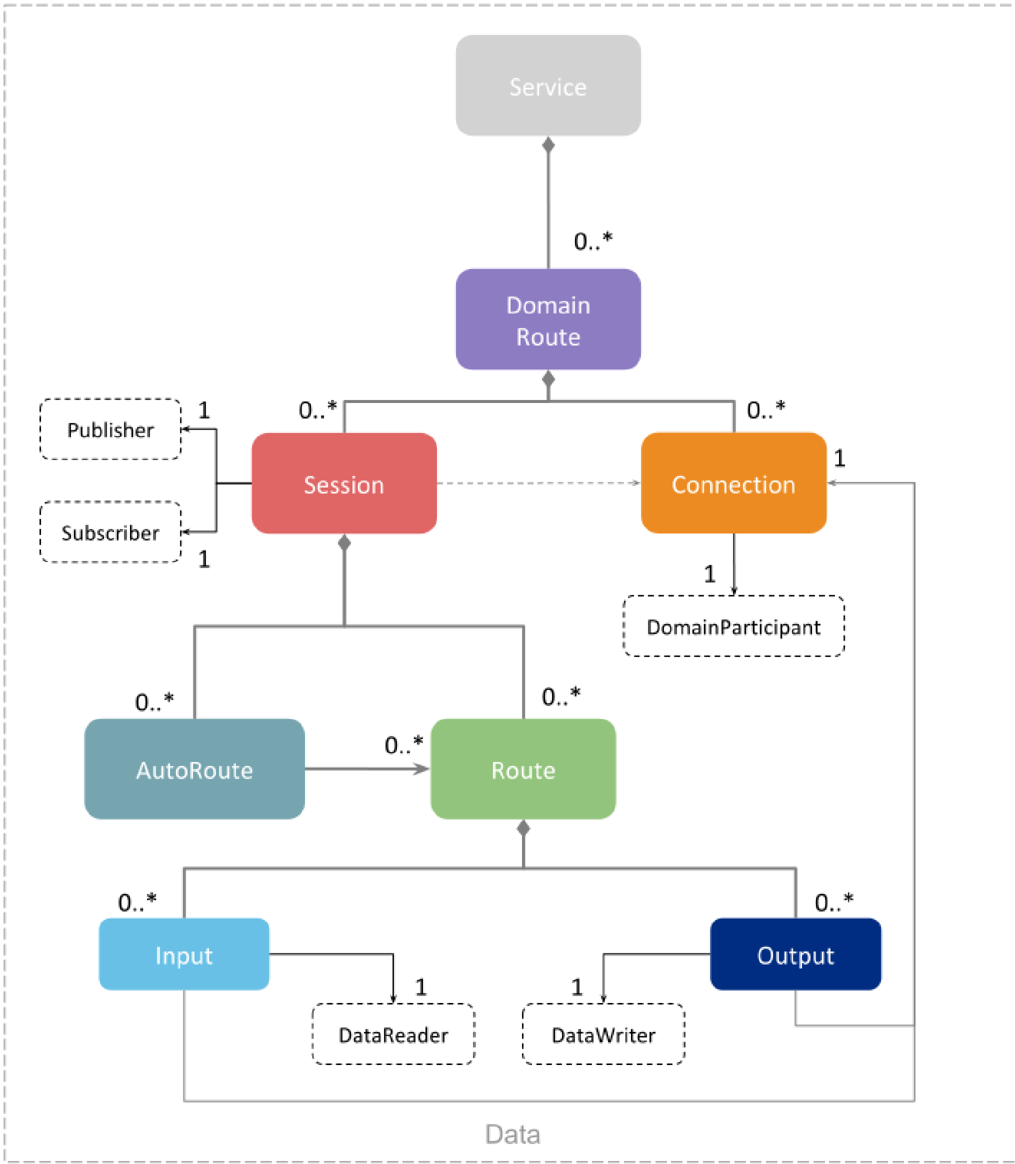


Figure 10.13: *Routing Service* DDS Application Resource Model

DDS AdapterPlugin

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

Table 10.10: 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 10.11: DDS *Connection*

Mapping	Configuration Tag
Composed of only one *DomainParticipant*	`<domain_route>/<participant>` (see Table 8.8)

DDS Session

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

Table 10.12: DDS *Session*

Mapping	Configuration Tag
Composed of only one *Publisher* and one *Subscriber*	`<session>/<subscriber_qos>` and `<session>/<publisher_qos>` (see Table 8.9)

Note that, as explained in *Plugin Interaction*, 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 10.13: DDS *StreamReader*

Mapping	Configuration Tag
Composed of only one *DataReader*	`<route>/<dds_input>` and `<topic_route>/<input>` (see Table 8.13)

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 10.14: DDS *StreamWriter*

Mapping	Configuration Tag
Composed of only one *DataWriter*	`<route>/<dds_output>` and `<topic_route>/<output>` (see Table 8.13)

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*.

10.2.2 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 *Propagation Mode*).

These functions are executed under the notification of the `DATA_ON_INPUTS` and `PERODIC_ACTION` events. 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 11

Advanced Use Cases

11.1 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 11.1 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* 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 11.2.

11.1.1 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 *Route*) and *AutoTopicRoute* configuration (see *Auto Route*).

[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

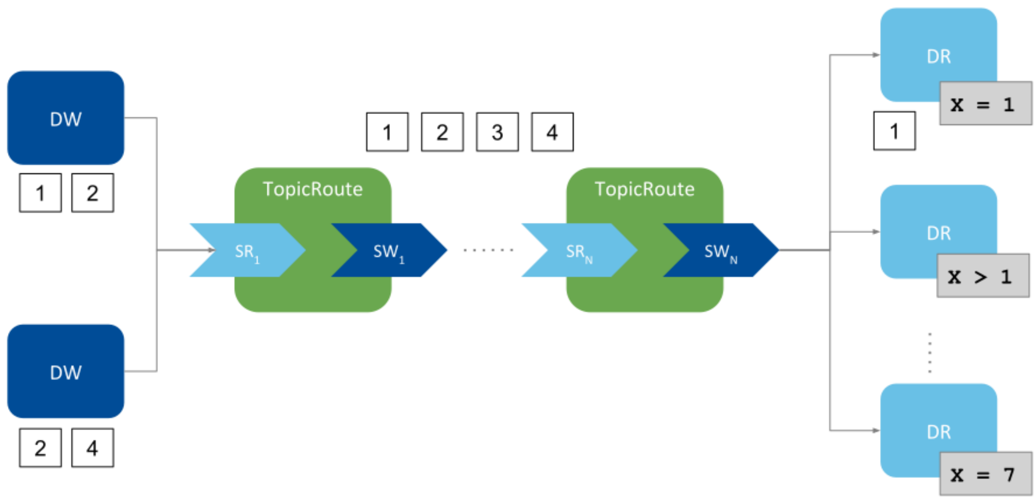


Figure 11.1: Without propagation, user *DataWriters* send all the samples; filtering occurs on the last route's *StreamWriter*

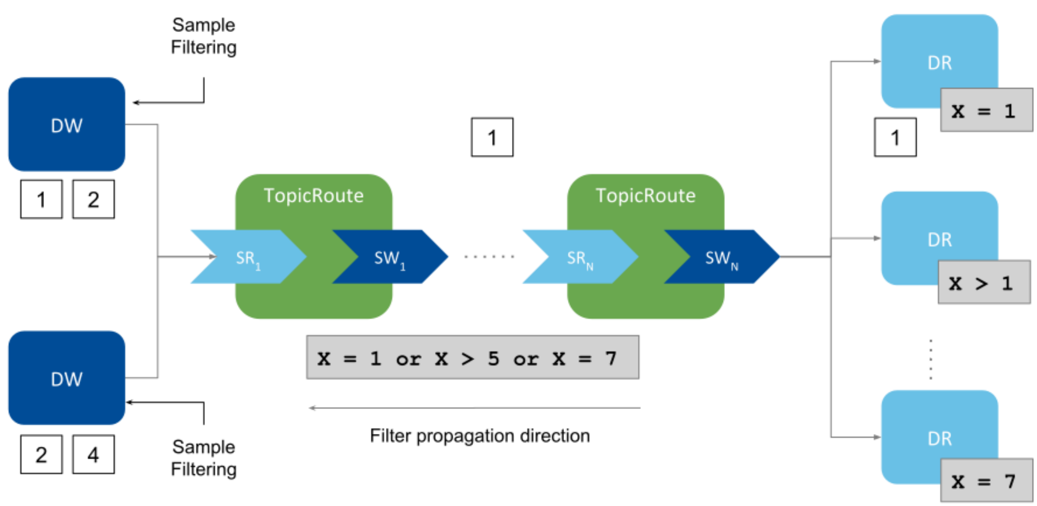


Figure 11.2: With propagation, user *DataWriters* receive a composed filter that allows writer-side filtering, thus sending only the samples of interest to the *DataReaders*

11.1.2 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. 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 11.3.

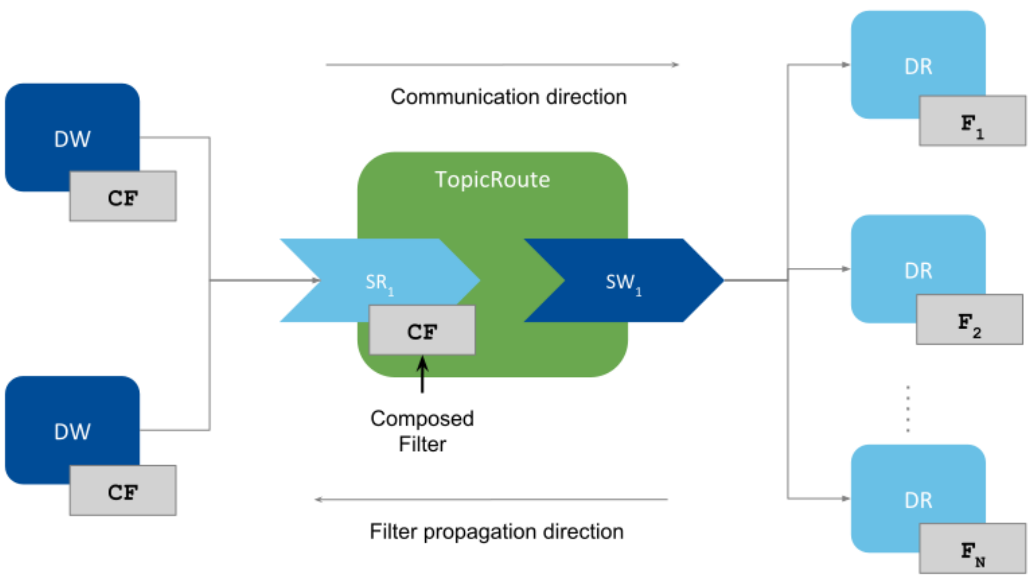


Figure 11.3: Filter Propagation Through Routing Service

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*.

11.1.3 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.

11.1.4 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.

11.2 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 *Route*) and *AutoTopicRoute* configuration (see *Auto Route*).

The following sections describe the *Routing Service* proxy modes. Figure 11.4 summarizes the symbols you will see in the figures that illustrate the modes' behaviors.

11.2.1 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 11.5 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 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.

11.2.2 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 11.6 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* requests, *TopicQuery* samples, and live samples interleaved.

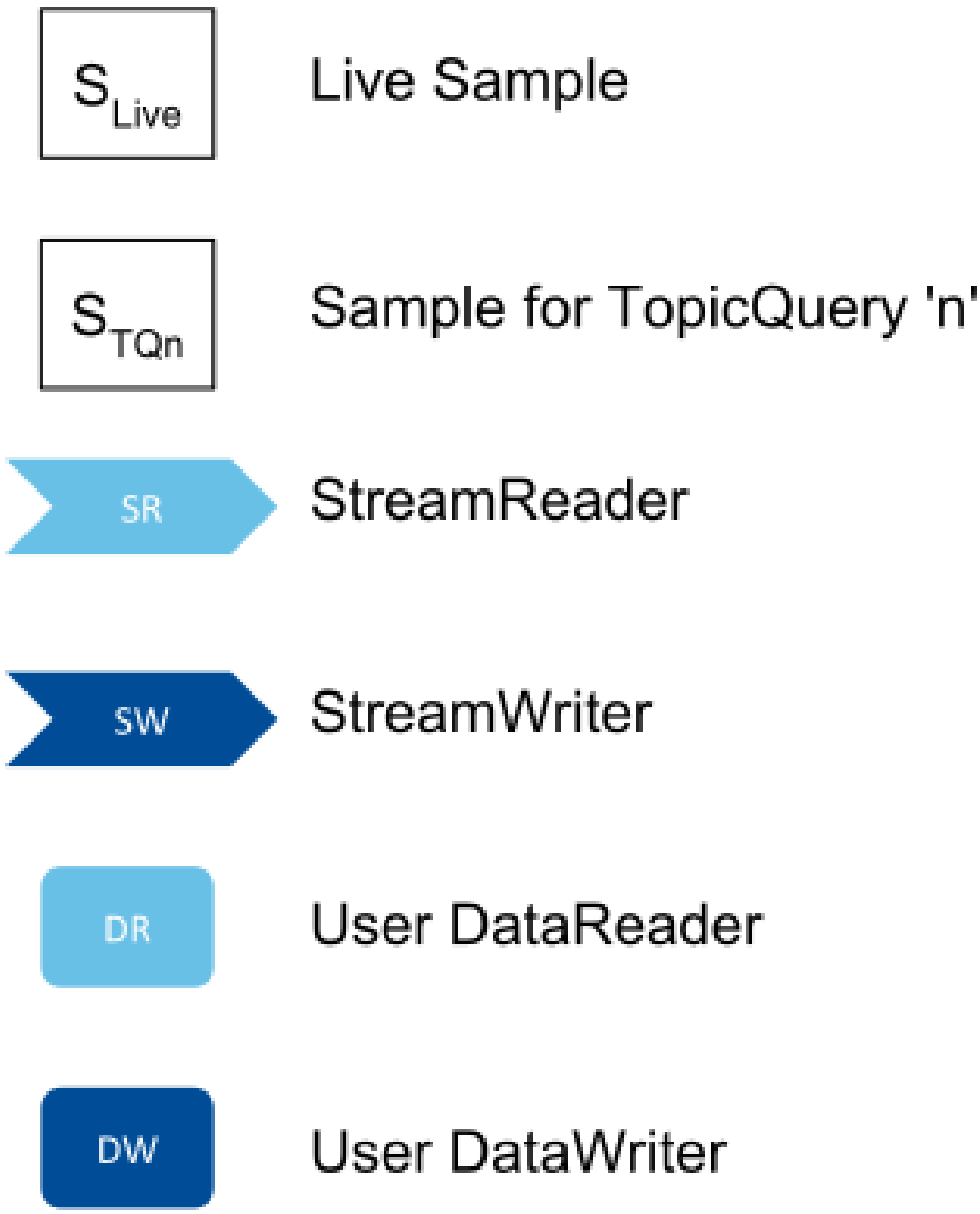


Figure 11.4: Symbol Legend for Proxy Modes Figures

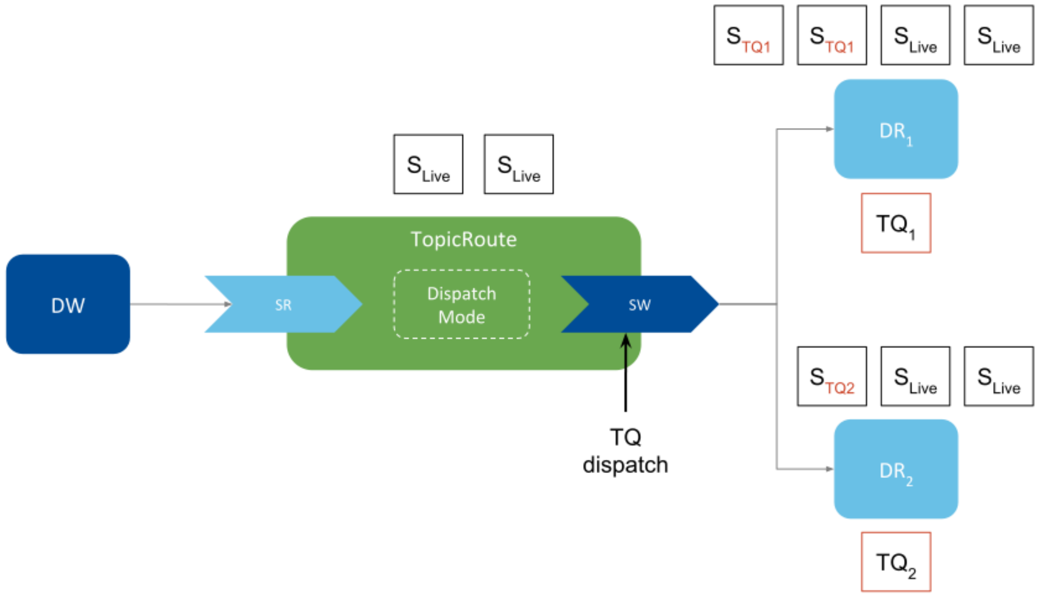


Figure 11.5: *TopicRoute* Enabling *TopicQuery* Proxy in Dispatch Mode

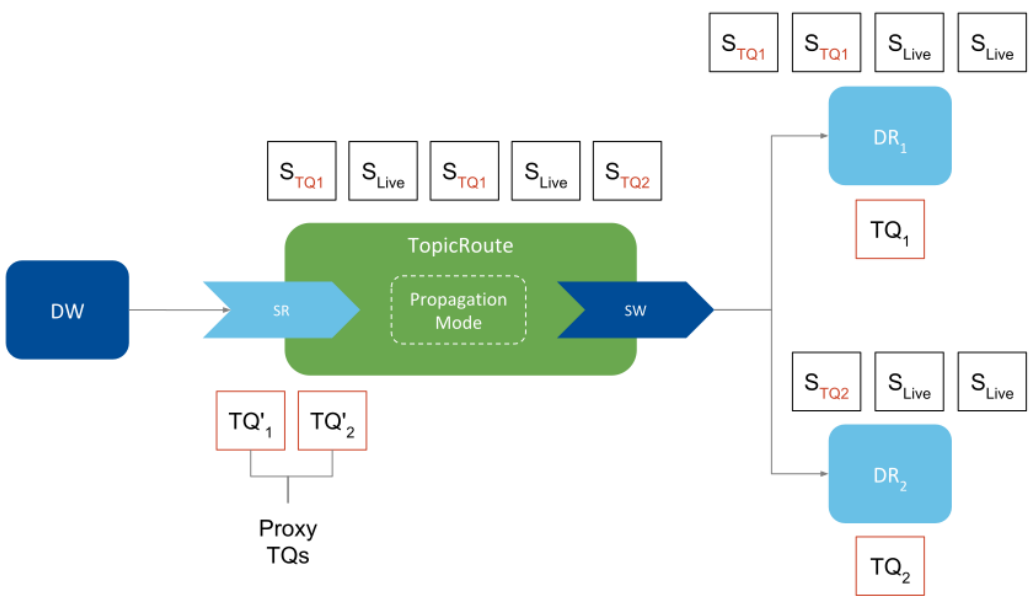


Figure 11.6: *TopicRoute* Enabling *TopicQuery* Proxy in Propagation Mode

TopicQuery propagation is also compatible with filter propagation (see *Propagating Content Filters*). 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.

11.2.3 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 12

Common Infrastructure

12.1 Configuring RTI Services

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

12.1.1 How to Load and Select an XML Configuration

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

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

Loading from Files

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

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

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

Shipped Executable

Use the `-cfgFile` option.

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

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

For example on Linux systems:

RTI Routing Service

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

RTI Recording Service

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

RTI Cloud Discovery Service

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

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

Library API

Set the `ServiceProperty::cfg_file` member.

For example in C++:

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

Loading from In-Memory Strings

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

For example in C++:

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

Selecting which Configuration to Run

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

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

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

RTI Routing Service

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

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

RTI Recording Service

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

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

RTI Cloud Discovery Service

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

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

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

Shipped Executable

Use the `-cfgName` option.

For example, on Linux systems:

RTI Routing Service

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

RTI Recording Service

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

RTI Cloud Discovery Service

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

Library API

Set the `ServiceProperty::cfg_name` member.

For example in C++:

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

Default Files

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

Table 12.1: RTI Services Default Files

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

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

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

These files are loaded only if present.

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

Shipped Executable

Use the -skipDefaultFiles option.

Library API

Set the ServiceProperty::skip_default_files member to true.

XML Syntax and Validation

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

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

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

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

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

Shipped Executable

Use the -ignoreXsdValidation option.

Library API

Set the ServiceProperty::enforce_xsd_validation member to false.

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

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="[NDDSHOME]/resource/schema/rti_routing_
↪service.xsd">
    <!-- ... -->
</dds>
```

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

Listing Available Configurations

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

RTI Routing Service

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

RTI Cloud Discovery Service

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

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

Configuration Variables

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

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

```
$(VAR_NAME)
```

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

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

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

1. Process environment.

   ```
   export VAR_NAME=my_value
   ```

2. Using a specific option when running the service.

   Shipped Executable

   Use the -DVAR_NAME=VALUE option

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

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

   Library API

   Set the ServiceProperty::user_environment member

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

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

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

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

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

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

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

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

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

12.1.2 How to Load Default QoS Profiles

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

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

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

12.1.3 How to Set Logging Properties

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

Command-Line Options

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

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

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

Library API

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

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

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

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

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

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

XML Configuration

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

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

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

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

See also:

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

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

12.1.4 How to Run as an Operating System Daemon

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

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

Linux and macOS Systems

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

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

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

    if (daemon(0,0)) {
        Logger::instance().error("Failed to create daemon process\n");
```

(continues on next page)

(continued from previous page)

```
        return -1;
    }

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

    service.start();
}
```

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

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

Windows Systems

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

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

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

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

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

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

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

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

- All AppParameters arguments must be enclosed in quotation marks.
- If you specify -cfgFile in the Start Parameters field, you must use the full path to the file.

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

12.1.5 How to use a License File with RTI Services

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

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

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

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

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

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

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

12.1.6 Key Terms

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

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

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

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

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

12.2 Application Resource Model

RTI Services are described through a *hierarchical application resource model*. In this model, an application is composed of a set of *Resources*, each representing a particular component within the application. *Resources* have a parent-child relationship. Figure 12.1 shows a general view of this concept.

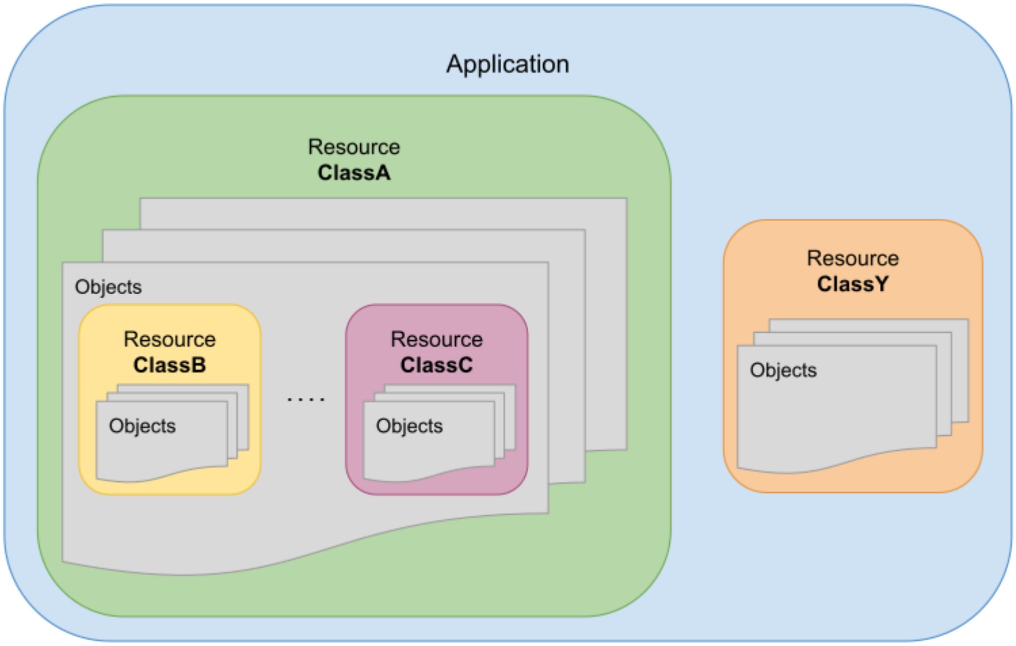


Figure 12.1: Application modeled as a set of related Resources

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.

12.2.1 Example: Simple Resource Model of a Connext Application

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

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

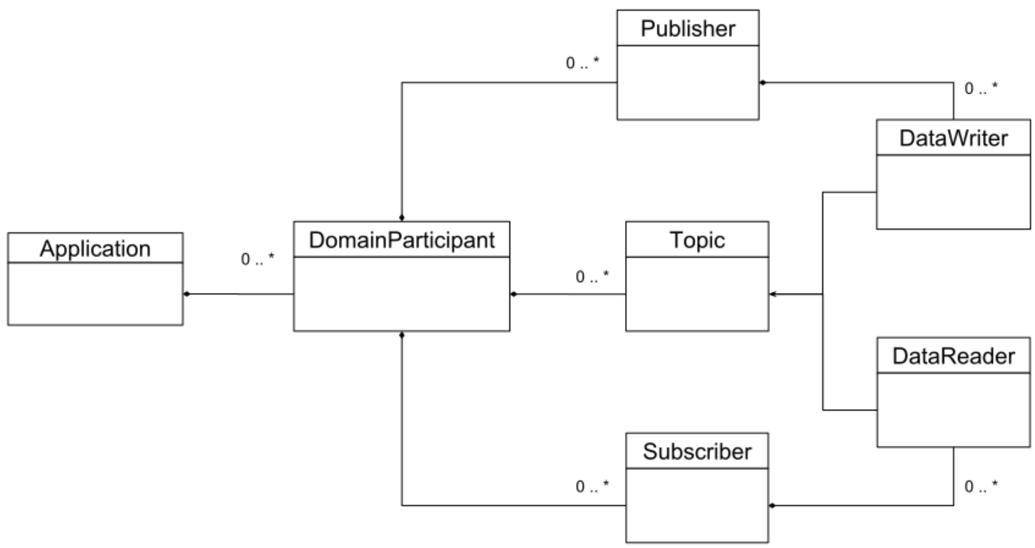


Figure 12.2: Connext DDS application resource model

12.2.2 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:

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

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

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

$\boxed{\text{/collection\_id}_1\text{/resource\_id}_1\text{/resource\_id}_2}$

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

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

The resource identifier format follows these conventions:

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

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

Escaped Identifiers

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

/"escaped_identifier"

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

For example, the following full resource path:

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

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

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

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

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

becomes

```
/routing_service/"A/B"

/routing_service/"A:B"
```

in the resource identifier.

Example: Resource Identifiers of a Generic Connext Application

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

```
/domain_participants/MyParticipant
```

In this case, "domain_participants" is the identifier of a collection resource that represents a set of *DomainParticipants* in the application and its value is fixed and specified by the application. In contrast, "MyParticipant" is the identifier of a simple resource that represents a particular *DomainParticipant* and its value is given by the user of the application at *DomainParticipant* creation time.

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

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

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

Example: Resource Identifiers Generated from XML Entity Model

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

```xml
<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"
```

12.3 Remote Administration Platform

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

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

The *RTI Remote Administration Platform* addresses two areas:

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

The combination of these two areas provides the general view of the *RTI Remote Administration Platform*, as shown in Figure 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*.

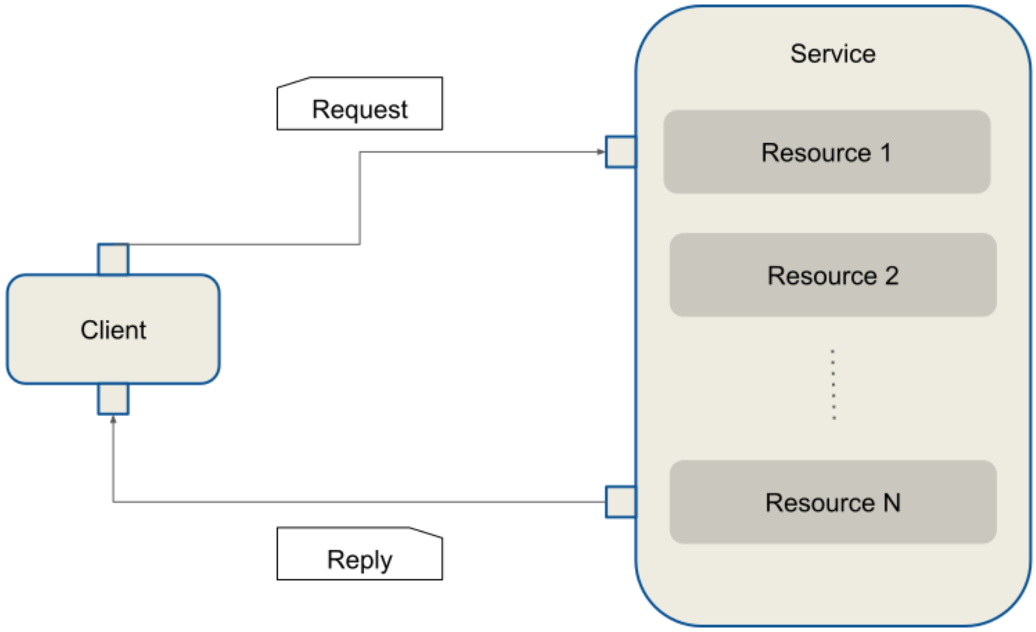


Figure 12.3: General View of the *RTI Remote Administration Platform* Architecture

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.

12.3.1 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 12.2: REST Interface

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

Standard Methods

The *RTI Remote Administration Platform* defines the methods listed in Table 12.3.

Table 12.3: 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
```

12.3.2 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*.

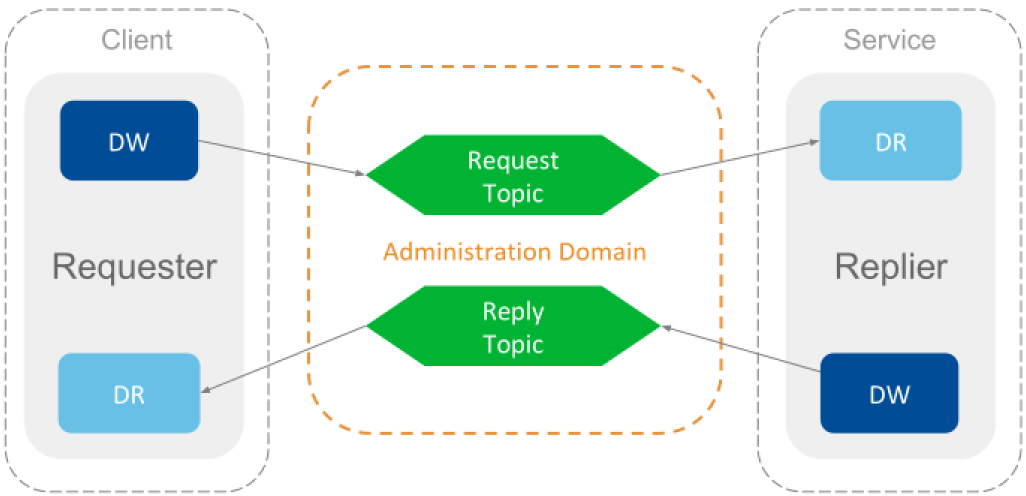


Figure 12.4: Communication in *RTI Remote Administration Platform* is Based on DDS Request-Reply

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 12.4:

Table 12.4: Remote Administration *Topics*

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

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

Listing 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 12.5: `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 12.6: `CommandReply`

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

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

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

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

Reply Sequence

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

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

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

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

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

Example: Controlling services remotely from a Connext Application

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

12.3.3 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:

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

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.

Update Resource

UPDATE [resource_identifier]

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

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

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

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

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

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

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

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

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

Request body

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

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

- Example `file://` request body:

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

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

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

Set Resource State

`UPDATE [resource_identifier]/state`

Sends a state change request to the specified resource object.

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

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

Request body

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

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

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

Get Resource State

`GET [resource_identifier]/state`

Gets the current state of the specified resource object.

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

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

Request body

- Empty

Reply body

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

Return codes

The operation may return a reply with error if:

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

Delete Resource

`DELETE [resource_identifier]`

Deletes the specified resource object.

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

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

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

Request body

- Empty.

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

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

12.4 Monitoring Distribution Platform

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

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

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

These three *Topic*s are shared across all services for the distribution of the monitoring information. Table 12.7 provides a summary of these topics.

Table 12.7: 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*.

12.4.1 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.

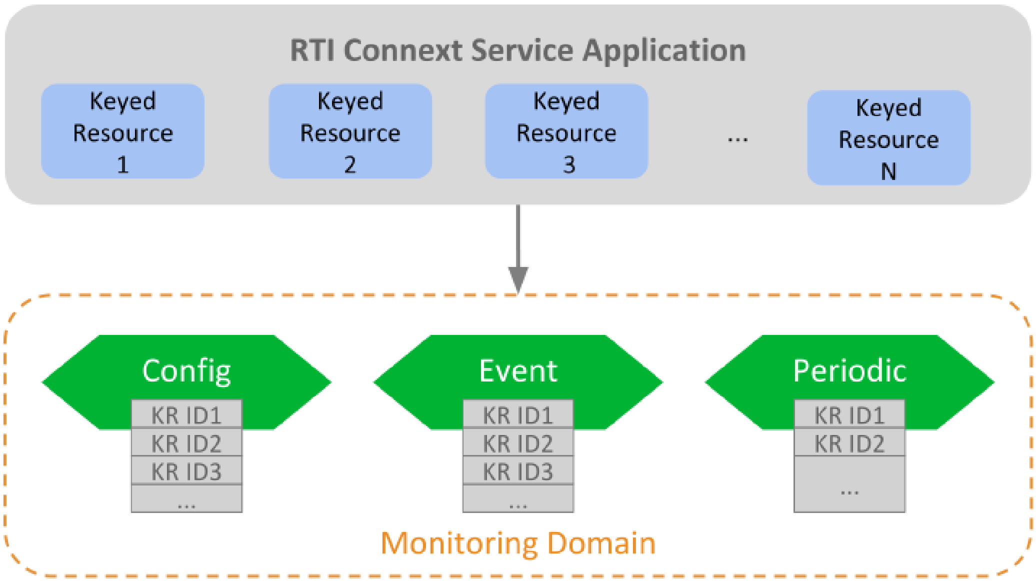


Figure 12.5: Monitoring Distribution *Topics* of *RTI* Services

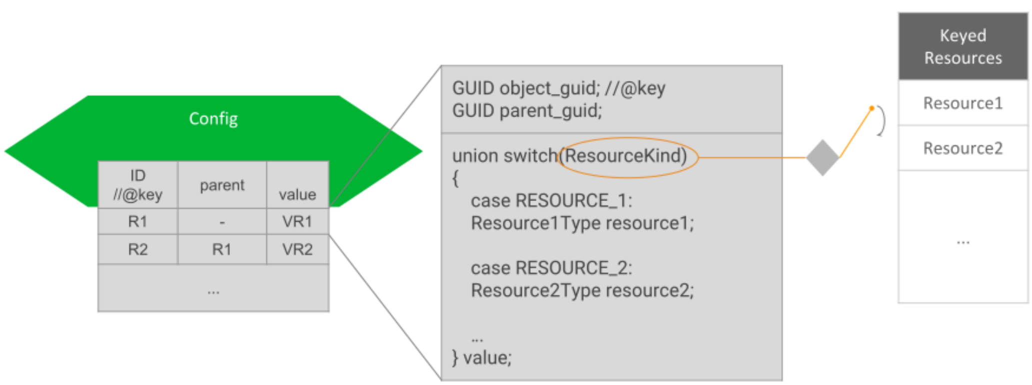


Figure 12.6: Monitoring Distribution *Topic* Definition

Keyed Resource Base Type Fields

This is the base type of all distribution *Topic*s and consists of two fields:

- `object_guid`: Key field. It represents a 16-byte sequence that uniquely identifies a *Keyed Resource* across all the available services in the monitoring domain. Hence, the associated instance handle key hash will be the same for all distribution *Topic*s, allowing easy correlation of a resource. It will also facilitate, as we will discuss later, easy instance data manipulation in a *DataReader*.
- `parent_guid`: It contains the object GUID of the parent resource. This field will be set to all zeros if the object is a top-level resource thus with no parent.

This base type, `KeyedResource`, is defined in `[NDDSHOME]/resource/idl/ServiceCommon.idl`.

Resource-Specific Type Fields

This is the type that conveys monitoring information for a concrete resource object. Since a distribution *Topic* is responsible for providing information about different resource classes, the resource-specific type consists of a single field that is a **Union of all the possible representations** for the keyed resources that provide that on the topic.

As expected, there must be consistency between the two parts of the distribution topic type. That is, a sample for a concrete resource object must contain the resource-specific union discriminator corresponding to the resource object's class.

Example: Monitoring of Generic Application

Assume a generic application that provides monitoring information about the modes of transports `Car`, `Boat` and `Plane`. Each mode is mapped to a keyed resource, each with a custom type that contains metrics specific to each class.

The monitoring distribution *Topic* top-level type, `TransportModeDistribution`, would be defined as follows, using IDL v4 notation:

```
#include "ServiceCommon.idl"

@nested
struct CarType {
    float speed;
    String color;
    String plate_number;
};

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

(continues on next page)

(continued from previous page)

```
@nested
struct PlaneType {
    float ground_speed;
    int32 air_track;
};

enum TransportModeKind {
    CAR_TRANSPORT_MODE,
    BOAT_TRANSPORT_MODE,
    PLANE_TRANSPORT_MODE
};

@nested
union TransportModeUnion switch (TransportModeKind) {
    case CAR_TRANSPORT_MODE:
    CarType car;

    case BOAT_TRANSPORT_MODE:
    BoatType boat;

    case PLANE_TRANSPORT_MODE:
    PlaneType plane;
}

struct TransportModeDistribution : KeyedResource {
    TransportModeUnion value;

};
```

Assume now that in the monitoring domain there are three resource objects, one for each resource class: a `Car` object 'CarA', a `Boat` object 'Boat1', and a `Plane` object 'PlaneX'. They all have unique resource GUIDs and each object represents an instance in the distribution *Topic*. The table shows the example of potential sample values:

Table 12.8: 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

12.4.2 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.

12.4.3 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 *Statistic Variable*).

There are two activities involved in the generation of the statistic variables: Calculation and Publication. All the configuration elements for these activities are available under the `<monitoring>` tag.

Calculation

The instrumented service periodically performs measurements on the metric. This activity is also known as *sampling* (don't confuse with data samples). The frequency of the measurements can be configured with the tag `<statistics_sampling_period>`. As a general recommendation, the sampling period should be a few times smaller than the publication period. A small sampling period provides more accurate statistics generation at the expense of increasing memory and CPU consumption.

Publication

The service periodically publishes a data sample containing a snapshot of the statistics generated during the calculation phase. The publication period can be configured with the tag `<status_publication_period>`.The value of a statistic variable corresponds to the time window of a publication period.

12.4.4 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 {
    uint64 period_ms;
    int64 count;
    float mean;
```

(continues on next page)

(continued from previous page)

```
        float minimum;
        float maximum;
        float std_dev;
    };

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

Table 12.9: 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;
    };

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

Table 12.10: `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 12.11: `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 {
    @optional StatisticVariable cpu_usage_percentage;
    @optional StatisticVariable physical_memory_kb;
    @optional StatisticVariable total_memory_kb;
    int32 uptime_sec;
};
```

Table 12.12: `ProcessConfig`

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

Table 12.13: `ProcessPeriodic`

Field Name	Description
cpu_usage_percentage	Statistic variable that provides the percentage of CPU usage of the process where the service is running. The field `count` of the variable contains the total CPU time in ms that the process spent during the publication period. Availability of this value is platform dependent.
physical_memory_kb	Statistic variable that provides the physical memory utilization in KiloBytes of the process where the service is running. Availability of this value is platform dependent.
total_memory_kb	Statistic variable that provides the virtual memory utilization in KiloBytes of the process where the service is running. Availability of this value is platform dependent.
uptime_sec	Time in seconds elapsed since the running service process started. Availability of this value is platform dependent.

Base Entity Resource Metrics

Listing 12.6: Base Entity Types

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

Table 12.14: `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 12.15: `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 12.16: `NetworkPerformance`

Field Name	Description
samples_per_sec	Statistic variable that provides information about the number of samples processed (received or sent) per second.
bytes_per_sec	Statistic variable that provides information about the number of bytes processed (received or sent) per second.
latency_millisec	Statistic variable that provides information about the latency in milliseconds for the data processed. The latency in a refers to the total time elapsed during the associated processing of the data, which depends on the type of application.

Thread Metrics

Listing 12.8: Thread Metrics Type

```
@mutable @nested
struct ThreadPeriodic {
    uint64 id;
    @optional StatisticVariable cpu_usage_percentage;
};

@mutable @nested
struct ThreadPoolPeriodic {
    @optional sequence<Service::Monitoring::ThreadPeriodic>␣
↪threads;
};
```

Table 12.17: `ThreadPeriodic`

Field Name	Description
id	OS-assigned thread identifier
cpu_usage_percentage	Statistic variable that provides the percentage of CPU usage of the thread belonging to the process where the service is running. The field `count` of the variable contains the total CPU time in ms that the thread spent during the publication period. Availability of this value is platform dependent.

12.5 Plugin Management

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

Figure 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*.

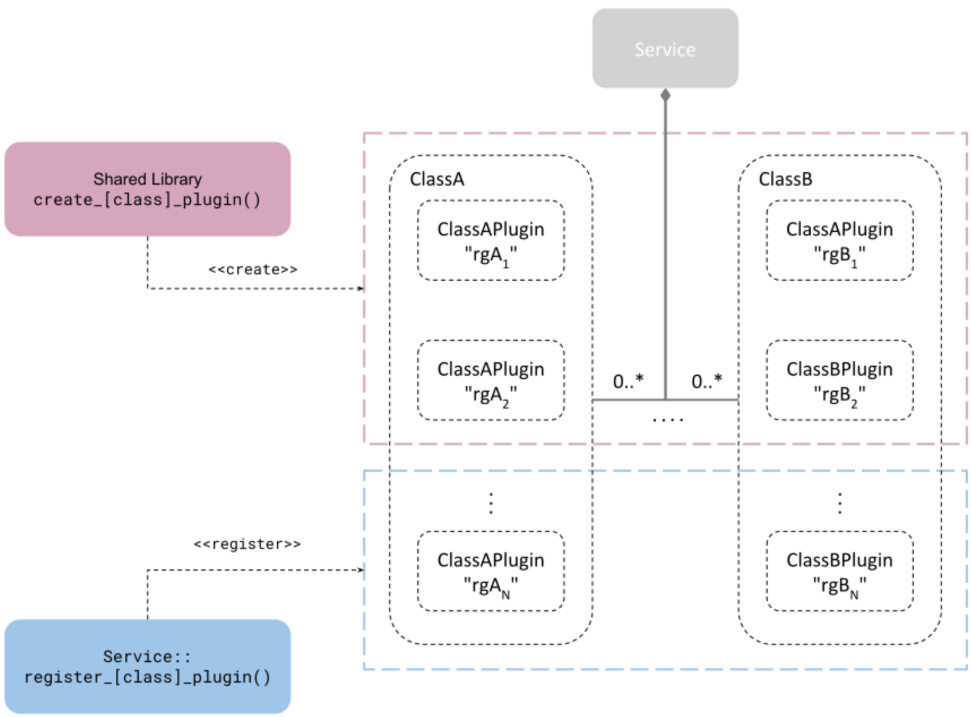


Figure 12.7: Plugin object management

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

12.5.1 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 *Configuration*). The *Service* loads the library the first time an instance of the plugin is needed (lazy initialization) and looks up the specified entry point symbol in the loaded library. The *Service* will always delete the plugin on *Service* stop.

This is the only method suitable when an RTI Service is deployed through an already linked executable, such as the shipped command-line executable (*Command-Line Executable*).

The plugin lifecycle is as follows:

1. After start, the *Service* creates a plugin object for each registered plugin in the configuration. The plugin object is instantiated through the shared library entry point, specified in the configuration.
2. The *Service* calls operations on the plugin objects as needed and keeps them alive while the *Service* remains started.
3. During stop, the *Service* deletes each plugin object by calling the class abstract deleter.

Configuration

An RTI Service configures the pluggable components within the `<plugin_library>` tag. RTI Services that support plugins will define a set of tags within in the form:

- `<[class]_plugin>` for C/C++ plugins
- `<java_[class]_plugin>` for Java plugins

where `[class]` refers to the name of the plugin class. For example, in *Routing Service* an available tag is `<adapter_plugin>`.

The definition of these tags is the same regardless of the plugin class and is described in the tables below.

Table 12.18 and Table 12.19 describe the configuration of each different plugin language.

Table 12.18: Configuration tags for C/C++ plugins.

Tags within <[class]_plugin>	Description	Multiplicity
<dll>	Shared library containing the implementation of the adapter plugin. This tag may specify the exact path (absolute or relative) of the file (for example, **lib/libmyplugin.so**) or a general name (no file extension). If no extension is provided, the path will be completed based on the running platform. For example, assuming a value for this tag of **dir/myplugin**: • Linux/macOS systems (or similar): **dir/libmyplugin.so** • Windows systems: **dir/myplugin.dll** If the library specified in this tag cannot be loaded (because the environment library path is not pointing to the path where the library is located), *Routing Service* will look for the library in the following locations, in this order: • **[plugin_search_path]**: Provided as part of the service parameters (see *Usage*) • **[executable_dir]**: Directory where the executable lives	1
<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 12.19: Configuration tags for Java plugins

Tags within <java_[class]_plugin>	Description	Multiplicity
<class_name>	Name of the class that implements the plugin. For example: com.myplugins.CustomPlugin The classpath required to run the Java plugin must be part of the RTI Service JVM configuration. See the `<jvm>` tag within the specific service configuration for additional information on JVM creation and configuration.	1

continues on next page

Table 12.19 – continued from previous page

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

<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

12.5.2 Library API

The user provides the plugin object via the Library API, through one of the available `attach_[class]_plugin()` operations. Upon successful return of the operation, the *Service* takes ownership of the plugin object and will delete it on *Service* stop.

The plugin lifecycle is as follows:

1. The user instantiates plugin objects and provides them to the *Service* through the `attach_[class]_plugin()` operation. This is allowed only before the *Service* starts.
2. After start, the *Service* becomes the owner of the registered plugin objects, calls operations on the plugin objects as needed, and keeps them alive while the *Service* remains started.
3. On stop, the *Service* deletes each registered plugin object by calling the class abstract deleter.

Chapter 13

Tutorials

This chapter describes several examples, all of which use *RTI Shapes Demo* to publish and subscribe to topics which are colored moving shapes. *Shapes Demo* is installed automatically with *RTI Connext* Professional. You'll find it in *RTI Launcher*'s Learn tab.

In each example, you can start all the applications on the same computer or on different computers in your network.

Important Notes:

- Please review *Paths Mentioned in Documentation* 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

Linux/macOS

```
$ export NDDS_DISCOVERY_PEERS=<host2>,<host3>
```

Windows

```
> set NDDS_DISCOVERY_PEERS=<host2>,<host3>
```

Host 2

Linux/macOS

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

Windows

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

Host 3

Linux/macOS

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

Windows

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

13.1 Starting Shapes Demo

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

Or from a command shell:

Linux/macOS

```
$ NDDSHOME/bin/rtishapesdemo
```

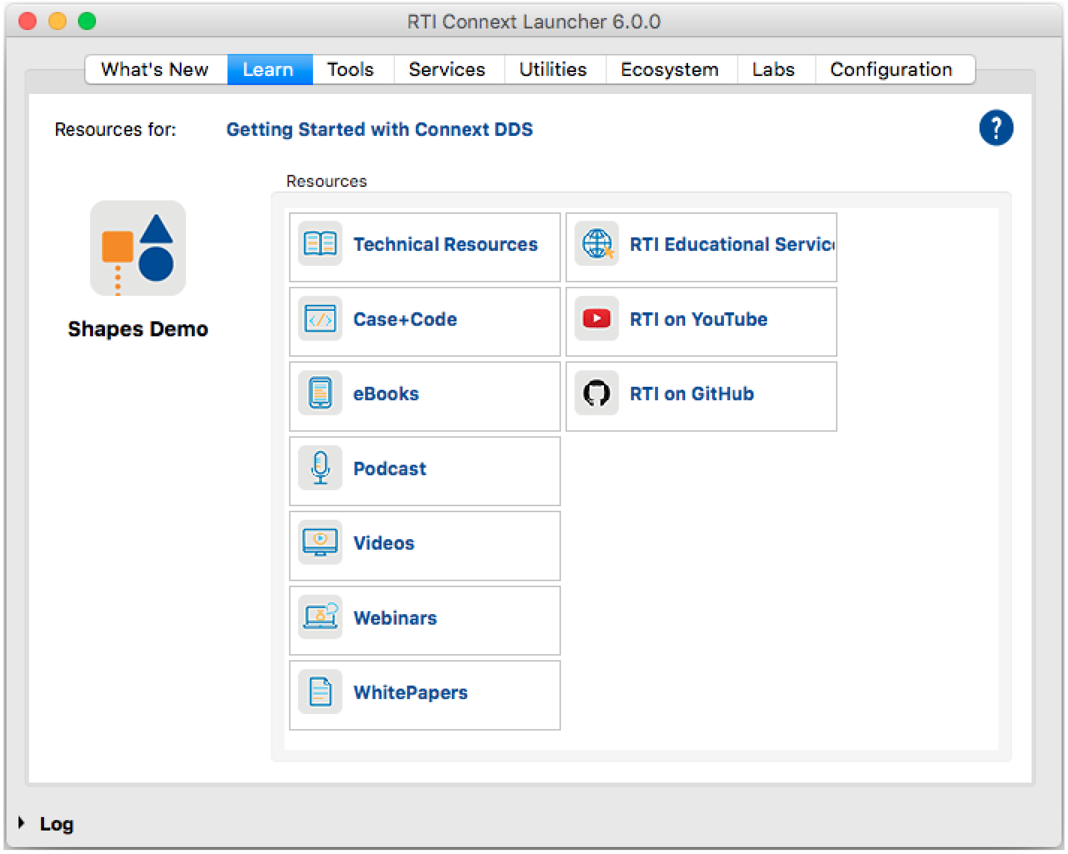
Windows

```
> %NDDSHOME%\bin\rtishapesdemo.bat
```

NDDSHOME is described in *Paths Mentioned in Documentation*.

13.2 Example: Routing a single specific Topic

This example routes the samples for the *Topic* `Square` from domain 0 to 1.



13.3 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).
   - 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:

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

```
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.

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

   Notice that the subscriber does not receive any samples, because the publisher and subscriber are in different domains.
5. 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
```

Notice that the Subscribing Demo only receives Circles, which match the movement of the Squares being published by the Publishing Demo. This is because the Squares are being republished as topic Circle.

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.

13.5 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 use a Transformation to see how to change the value of some fields in the samples and republish them.

Note: *Routing Service* provides a transformation that is able to map fields of the input type to fields of the output type using the property tag inside the transformation to provide this mapping. The `<name>` tag indicates the name of the field in the output type; the `<value>` tag indicates the name of the field in the input type. Use dot notation for nested fields (e.g., `position.x`).

Important: The assign transformation only supports the assignment of primitive fields (including strings) that are not part of arrays or sequences. So, for example, `x[0]` is not supported.

---

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.
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.

   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**.

13.6 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*; it monitors publications on any DDS domain.

Note: *Routing Service* provides a transformation that is able to map fields of the input type to fields of the output type using the property tag inside the transformation to provide this mapping. The `<name>` tag indicates the name of the field in the output type; the `<value>` tag indicates the name of the field in the input type. Use dot notation for nested fields (e.g., `position.x`).

Important: The assign transformation only supports the assignment of primitive fields (including strings) that are not part of arrays or sequences. So, for example, `x[0]` is not supported.

---

1. Start *Shapes Demo*. We'll call this the Publishing Demo. It uses domain ID 0.
2. In the Publishing Demo, publish some Squares.
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**.

13.7 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* 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 Linux/macOS systems:
     - Set the environment variable NDDSHOME (see *Paths Mentioned in Documentation*). An easy way to do this is to run *rtisetenv*: `$ source <installdir>/resource/scripts/rtisetenv_<architecture>.bash`. (For more information about *rtisetenv*, see Set Up Environment Variables (rtisetenv), in the RTI Connext DDS Getting Started Guide.)
     - Enter:

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

   - On Windows systems:
     - Set the environment variable NDDSHOME (see *Paths Mentioned in Documentation*). An easy way to do this is to run *rtisetenv*: `> <installdir>\resource\scripts\rtisetenv_<architecture>.bat`. (For more information about *rtisetenv*, see Set Up Environment Variables (rtisetenv), in the RTI Connext DDS 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.

13.8 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
```

13.9 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. 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*.

6. Now you can subscribe to the monitoring topics (see *Monitoring*). 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*.

---

7. 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 approximately 40 (on some systems like Windows it can be approximately 32 because of timer resolution within Shapes Demo). That value depends on the publication rate of *Shapes Demo*, which is configurable with the option **-pubInterval <milliseconds between writes>**. The default wait between writes in *Shapes Demo* is 50 ms, which results in a publication rate of approximately 20Hz (can be closer to 16Hz on Windows systems).

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

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

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

10. 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 approximately 120 samples per second (approximately 96 on Windows).

11. We can update the monitoring configuration at run time using the remote administration feature.

    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.

    ---

12. 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
    ```

Note that this change just increases the status publication rate, but does not change the statistics sampling period.

13. 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
```

14. To enable it again, enter:

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

15. 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.

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

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

17. Stop *rtiddsspy* by pressing **Ctrl-c**.
18. Stop the shell:

```
exit
```

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

13.10 Example: WAN Connectivity using the TCP transport

This example shows how to use *Routing Service* to bridge data between different LANs over the WAN using TCP. See *Traversing Wide Area Networks* for a guided and detailed explanation to understand the configuration for this example.

Figure 2.10 shows the example scenario. There are two instances of *Routing Service* acting as WAN gateways. `GatewaySiteA` is the *Routing Service* that connects the databus for domain 0 in the `site A` LAN. Similarly, `GatewaySiteB` is the *Routing Service* that connects the databus for domain 2 in the `site B` LAN. Note that `GatewaySiteA` runs in a host behind a NAT/Firewall, with a public address and forwarded public port to that host. Hence this information is required in the TCP configuration of `GatewaySiteA`.

This example uses XML configuration variables in order to reuse the same participant QoS and service configuration. You will need to set to appropriate values (if different than default) when you run *Routing Service* for each site. For the steps shown in this example, the following values are chosen:

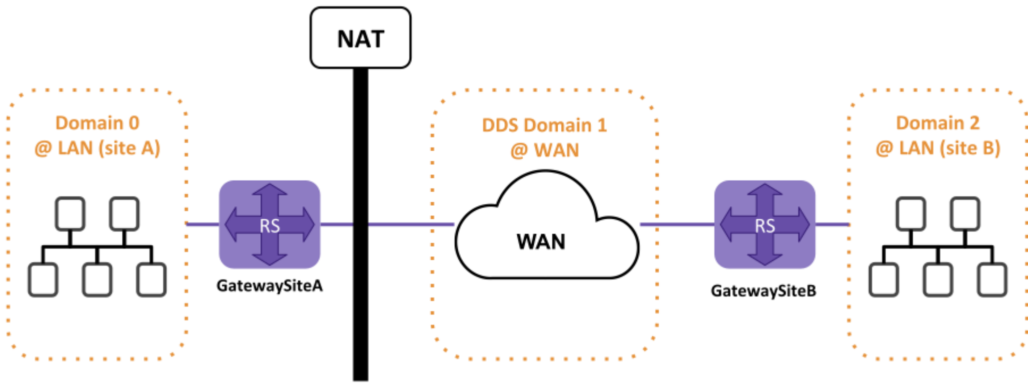


Figure 13.1: Example using the TCP transport to traverse WAN

Table 13.1: Values for configuration variables in this example

Variable	GatewaySiteA	GatewaySiteB
PUBLIC_ADDRESS	10.10.1.140	10.10.1.150
REMOTE_RS_PEER	tcpv4_wan://10.10.1.150:8400	tcpv4_wan://10.10.1.140:7400
BIND_PORT	7400	8400
LAN_DOMAIN_ID	0	2

- On GatewaySiteA host (behind a NAT/firewall with a public IP):

  1. In the Site A network, configure the firewall to forward the TCP ports used by *Routing Service*.

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

Note: You can use tools like Netcat or Ncat, depending on your platform, to verify that the port forwarding has been enabled before moving on with the next steps. For instance, you can run a simple client/server test between the machines running GatewaySiteA (server) and GatewaySiteB (client).

  1. Start *Routing Service* with the GatewaySiteA configuration

     To run with the default values for the XML variables:

```
cd <NDDSHOME>
bin/rtiroutingservice
        -cfgFile rti_rs_example_tcp_wan.xml \
        -cfgName WanGateway \
        -appName GatewaySiteA
```

     You can set the values for the XML configuration variables in the environment:

     Linux/macOS

```
$ export PUBLIC_ADDRESS=<Host Site A public IP>:<Host
↪Site A public Port>
$ export BIND_PORT=<RS TCP bind port>
```

```
> set PUBLIC_ADDRESS=<Host Site A public IP>:<Host Site␣
↪A public Port>
> set BIND_PORT=<RS TCP bind port>
> set REMOTE_RS_PEER=<discovery peer for GatewaySiteB>
> set LAN_DOMAIN_ID=<ID for the LAN domain in site A>
```

Now run the *Routing Service* instance:

```
cd <NDDSHOME>
bin/rtiroutingservice
        -cfgFile rti_rs_example_tcp_wan.xml \
        -cfgName WanGateway \
        -appName GatewaySiteA
```

For example:

```
cd <NDDSHOME>
export PUBLIC_ADDRESS=10.10.1.140:7400
export BIND_PORT=7400
export REMOTE_RS_PEER=tcpv4_wan://10.10.1.150:8400
export LAN_DOMAIN_ID=0

bin/rtiroutingservice
        -cfgFile rti_rs_example_tcp_wan.xml \
        -cfgName WanGateway \
        -appName GatewaySiteA
```

2. On any computer in Site A LAN, start *Shapes Demo* on domain 0 and publish Squares.

   If the computer running *Shapes Demo* is different than the host running `GatewaySiteA`, you may need to set the initial peers to the address of that host. You can do this by setting the `NDDS_DISCOVERY_PEERS` environment variable before starting *Shapes Demo*.

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

  1. In the `Site A` network, configure the firewall to forward the TCP ports used by *Routing Service*.

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

  2. Start *Routing Service* with the `GatewaySiteB` configuration

     To run with the default values for the XML variables:

```
cd <NDDSHOME>
bin/rtiroutingservice
        -cfgFile rti_rs_example_tcp_wan.xml \
        -cfgName WanGateway \
        -appName GatewaySiteB
```

You can set the values for the XML configuration variables in the environment:

Linux/macOS

```
$ export PUBLIC_ADDRESS=<Host Site B public IP>:<Host Site B␣
↪public Port>
$ export BIND_PORT=<RS TCP bind port>
$ export REMOTE_RS_PEER=<discovery peer for GatewaySiteA>
$ export LAN_DOMAIN_ID=<ID for the LAN domain in site B>
```

Windows

```
> set PUBLIC_ADDRESS=<Host Site B public IP>:<Host Site B␣
↪public Port>
> set BIND_PORT=<RS TCP bind port>
> set REMOTE_RS_PEER=<discovery peer for GatewaySiteA>
> set LAN_DOMAIN_ID=<ID for the LAN domain in site B>
```

Now run the *Routing Service* instance:

```
cd <NDDSHOME>
bin/rtiroutingservice
        -cfgFile rti_rs_example_tcp_wan.xml \
        -cfgName WanGateway \
        -appName GatewaySiteB
```

For example:

```
cd <NDDSHOME>
export PUBLIC_ADDRESS=10.10.1.150:8400
export BIND_PORT=8400
export REMOTE_RS_PEER=tcpv4_wan://10.10.1.140:7400
export LAN_DOMAIN_ID=2

bin/rtiroutingservice
        -cfgFile rti_rs_example_tcp_wan.xml \
        -cfgName WanGateway \
        -appName GatewaySiteB
```

3. On any computer in Site B LAN, start *Shapes Demo* on domain 2 and subscribe to Squares.

   If the computer running *Shapes Demo* is different than the host running `GatewaySiteB`, you may need to set the initial peers to the address of that host. You can do this by setting the `NDDS_DISCOVERY_PEERS` environment variable before starting *Shapes Demo*.

13.10.1 Important Considerations

- **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**. You will also need to set the peer prefix to **tcpv4_lan://** when setting the discovery peer in the `RS_REMOTE_PEER` configuration variable.

  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.

- **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://** when setting the discovery peer in the `RS_REMOTE_PEER` configuration 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://**.

13.11 Example: Using a File Adapter

The previous examples showed how to use *Routing Service* with *Connext*. 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.

You can find the full example in the RTI Community Examples Repository. To learn how to implement your own adapter, you can follow this example and the explanations from *Integrating a File-Based Domain*.

13.12 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 14

Release Notes

14.1 Supported Platforms

See the *Routing Service* column in the Table of Supported Platforms for Compiler-Dependent Products, in the RTI Connext Core Libraries Release Notes.

Routing Service can also be deployed as a C library linked into your application.

14.2 Compatibility

For backward compatibility information between the current and previous versions of *Routing Service*, 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*, 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* 5.1.0, the default `message_size_max` for the UDPv4, UDPv6, TCP, and shared-memory transports changed to provide better out-of-the-box performance. *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*. Please see the *RTI Connext DDS Core Libraries Release Notes* for instructions on how to resolve this compatibility issue with older *Connext* 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.

14.3 What's New in 7.2.0

14.3.1 Support for dynamic certificate renewal

A running *Routing Service* instance can use the new `authentication.identity_certificate_file_poll_period.millisec` property in *RTI Security Plugins* to renew its identity certificate without the need to restart the service. The `authentication.identity_certificate_file_poll_period.millisec` property must have a value greater than zero for the participant to periodically poll its identity certificate file for changes.

For more information, see the *Configuration* section in this manual and the Advanced Authentication Concepts section in the *Security Plugins User's Manual*.

14.3.2 Support for dynamic certificate revocation

A running *Routing Service* instance can use the `authentication.crl` and the new `authentication.crl_file_poll_period.millisec` properties in *RTI Security Plugins* to specify certificate revocations without the need to restart the service. The `authentication.crl_file_poll_period.millisec` property must have a value greater than zero for the *DomainParticipant* to periodically poll the provided CRL file for changes.

For more information, see *Support for RTI Security Plugins* in this manual and Advanced Authentication Concepts in the *RTI Security Plugins User's Manual*.

14.3.3 Support for Monitoring Library 2.0

RTI Routing Service now supports enabling the new *RTI Monitoring Library 2.0* to send monitoring information and metrics about the DDS entities it creates.

To enable *Monitoring Library 2.0* when using the *Routing Service* application, include the following code in the appropriate XML profile:

```
<qos_profile name="RoutingServiceProfile" is_default_participant_factory_
↪profile="yes">
 <participant_factory_qos>
     <monitoring>
         <enable>true</enable>
         <application_name>Routing Service</application_name>
     </monitoring>
 </participant_factory_qos>
</qos_profile>
```

When using *Routing Service* as a library, *Monitoring Library 2.0* can be enabled programmatically using the `RTI_Monitoring_enable_with_property` and `RTI_Monitoring_disable` methods.

14.3.4 Third-party software changes

The following third-party software used by *Routing Service* has been upgraded:

Table 14.1: Third-Party Software Changes

Third-Party Software	Previous Version	Current Version
libxml2	2.9.4	2.11.4
libxslt	1.1.35	1.1.38

For information on third-party software used by *Connext* products, see the "3rdPartySoftware" documents in your installation: <NDDSHOME>/doc/manuals/connext_dds_professional/release_notes_3rdparty.

14.4 What's Fixed in 7.2.0

14.4.1 Entity Listener API sometimes fired the STARTED event twice

There was a race condition in the Routing Service Entity Listener API where, in certain conditions, a STARTED event may have been fired twice for the same Topic Route. This issue has been resolved.

[RTI Issue ID ROUTING-892]

14.4.2 Possible race condition when propagating content filters

When several applications using different content filters were started simultaneously and discovered by one or more instances of *Routing Service*, it was possible that filter propagation did not propagate all filters properly upon route startup, resulting in an inconsistent state that may have led to data loss. This issue has been resolved.

[RTI Issue ID ROUTING-1055]

14.4.3 Overflows caused issues in period calculations

Routing Service had issues calculating period metrics due to overflows. This issue is resolved; in `ServiceCommon.idl`, the `StatisticMetrics` field's `period_ms` value was changed to uint64.

[RTI Issue ID ROUTING-1068]

14.5 Previous Release

14.5.1 What's New in 7.1.0

There are no changes to *Routing Service* in this release. The most recent changes are documented in *What's New in 7.0.0*.

14.5.2 What's Fixed in 7.1.0

Routing Service Crashed if -maxObjectsPerThread Set Too Small

Routing Service crashed if the command-line option `-maxObjectsPerThread` had a value less than 1024. This issue, which also affected Recording Service, has been resolved. Now instead of crashing, the service will log the following warning and the default value will be used.

```
Max objects per thread can't be lower than 1024. Setting MaxObjectsPerThread␣
↪to 1024.
```

[RTI Issue ID ROUTING-1024]

14.5.3 What's New in 7.0.0

The following third-party software used by *Routing Service* has been upgraded:

Table 14.2: Third-Party Software Changes

Third-Party Software	Previous Version	Current Version
libxml2	2.9.12	2.9.14
libxslt	1.1.34	1.1.35

For information on third-party software used by *Connext* products, see the "3rdPartySoftware" documents in your installation: `<NDDSHOME>/doc/manuals/connext_dds_professional/release_notes_3rdparty`.

14.5.4 What's Fixed in 7.0.0

Schema files not compliant with DDS-XML specification

The schema file `rti_service_common_definitions.xsd`, and its included files, have been changed as follows to make them compliant with the DDS-XML specification (https://www.omg.org/spec/DDS-XML/1.0/PDF):

- <participant_qos> has been renamed to <domain_participant_qos>.

The old tag is still accepted by the Connext XML parser and the XSD schema to maintain backward compatibility.

[RTI Issue ID ROUTING-814]

Samples published out of order from the same virtual GUID were dropped

If *Routing Service* received samples for a given virtual GUID with sequence numbers out of order, it dropped samples with sequence numbers lower than the highest received sequence number. This issue has been resolved.

[RTI Issue ID ROUTING-928]

Fourth digit of product version not logged by Routing Service at startup

The *Routing Service* executable did not log the fourth digit (revision) of the product version when the service started. This problem has been resolved.

[RTI Issue ID ROUTING-975]

Routing Service stream query propagation did not work when using more than one session

When propagating stream query result samples, *Routing Service* may have sent corrupted data if the configuration used more than one session and both sessions were writing stream query samples at the same time. This issue has been resolved.

[RTI Issue ID ROUTING-997]

14.6 Known Issues

Note: For an updated list of critical known issues, see the Critical Issues List on the RTI Customer Portal at https://support.rti.com/.

14.6.1 Attempting to route builtin Security Logging topic causes Routing Service crash

Routing the Security Logging builtin topic (`DDS:Security:LogTopic`) causes a crash if any of the participants involved in the route have security logging enabled (i.e., the `property com.rti.serv.secure.logging.distribute.enabled` is set to true).

Note that you can enable security logging on participants that talk to *Routing Service* and even route the Security Logging builtin topic that they use. This problem occurs only if the *Routing Service* participant itself has security logging enabled.

[RTI Issue ID ROUTING-727]

Chapter 15

Copyrights and Notices

Trademarks

Copy and Use Restrictions

Third-Party Software

Notices

Deprecations and Removals

Any deprecations or removals noted in this document serve as notice under the Real-Time Innovations, Inc. Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance and support of RTI's software.

Deprecated means that the item is still supported in the release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported. By specifying that an item is deprecated in a release, RTI hereby provides customer notice that RTI reserves the right after one year from the date of such release and, with or without further notice, to immediately terminate maintenance (including without limitation, providing updates and upgrades) for the item, and no longer support the item, in a future release.

Technical Support Real-Time Innovations, Inc. 232 E. Java Drive Sunnyvale, CA 94089 Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/

Index