# Contents

1 Introduction 2
   1.1 Introduction .................................................. 2
   1.2 The Basics .................................................... 2
   1.3 Paths Mentioned in Documentation ............................. 3

2 Installation 5

3 Recording Service 6
   3.1 Usage .......................................................... 6
      3.1.1 Starting Recording Service .............................. 6
      3.1.2 Stopping Recording Service ............................. 7
      3.1.3 Recording Service Command-Line Parameters .......... 7
      3.1.4 Controlling Recording Service’s Operation Mode .... 8
         Controlling Buffer Size in Buffering Mode ............... 9
         Configuring In-Memory Buffer Size ...................... 9
   3.2 Operating System Daemon ..................................... 10
   3.3 Configuration ................................................ 10
      3.3.1 Built-in Configuration of Recording Service ......... 10
      3.3.2 XML Tags for Configuring Recording Service ........ 10
      3.3.3 Recording Service Tag .................................. 11
         Example: Specify a Recording Service Configuration in XML 12
      3.3.4 Administration ........................................... 15
      3.3.5 Monitoring ............................................... 16
      3.3.6 Storage ................................................... 16
         SQLite ......................................................... 18
         Plugin ......................................................... 23
         Instance Indexing ............................................ 24
      3.3.7 DomainParticipant ....................................... 24
      3.3.8 Session .................................................. 26
      3.3.9 Topic Group ............................................. 28
      3.3.10 Topic .................................................. 29
      3.3.11 Support for RTI FlatData and Zero Copy Transfer Over Shared Memory 30
         Example: Configuration to enable both FlatData and Zero Copy transfer over shared memory 30
      3.3.12 Plugins .................................................. 31
      3.3.13 Enabling Distributed Logger ............................ 32
      3.3.14 Recording Service Built-in Configuration Details .... 32
3.4 Remote Administration ................................. 33
  3.4.1 Enabling Remote Administration ......................... 34
  3.4.2 Available Service Resources .......................... 34
  Example ............................................. 34
  3.4.3 Remote API Overview ................................ 35
  3.4.4 Recording Service .................................. 35
  3.4.5 Storage .......................................... 37
3.5 Monitoring .............................................. 38
  3.5.1 Overview .......................................... 38
  Enabling Service Monitoring ................................. 38
  Monitoring Types ....................................... 39
  3.5.2 Monitoring Metrics Reference .......................... 40
  Service ............................................... 40
  Session ............................................... 42
  TopicGroup ............................................. 43
  Topic .................................................. 44
3.6 Tutorials .................................................. 45
  3.6.1 Getting Started with Recording Service and Shapes Demo 45
  Edit the Configuration .................................. 46
  Start Shapes Demo ...................................... 47
  Start Recording Service .................................. 49
  View the Data in Sqlite3 .................................. 49
  3.6.2 Using Recording Service and Admin Console ............... 50
  Configuration ........................................... 50
  Start Recording Service .................................. 50
  Start Shapes Demo ...................................... 50
  Viewing with Admin Console ................................. 50
  Administering with Admin Console ........................... 52
  3.6.3 Using Recording Service as a Library .................... 55
  Include files ............................................ 55
  Using the RecordingService class ............................ 56
  3.6.4 Plugging in Custom Storage ............................ 57
  Custom Storage API Overview ............................... 57
  3.6.5 Accessing JSON samples through SQL ..................... 58
  3.6.6 Controlling Recording Service Remotely from an Application 58
  3.6.7 Listing the Timestamp Tags in a Recording ................. 59
3.7 Troubleshooting .......................................... 60
  3.7.1 Verbosity .......................................... 60
4 Replay Service ............................................. 61
  4.1 Usage ................................................ 61
  4.1.1 Starting Replay Service ................................ 61
  4.1.2 Stopping Replay Service ............................... 62
  4.1.3 Replay Service Command-Line Parameters ................. 62
  4.1.4 Replay Service Runtime Behavior ........................ 63
  4.1.5 Working With Large Data ................................ 63
  4.1.6 Choosing the Sample Order for Replaying Data ............. 64
  4.1.7 Recreating the State of the World when Replaying (Replaying Instance History) 64
4.1.8 Jumping in Time while Replaying ........................................ 65
4.1.9 Using Debug Mode while Replaying ................................. 66

4.2 Operating System Daemon ................................................ 66

4.3 Configuration .............................................................. 66
4.3.1 XML Tags for Configuring Replay Service ........................ 67
4.3.2 Replay Service Tag .................................................... 68
Example: Specify a Replay Service Configuration in XML ........ 71
4.3.3 Administration ......................................................... 72
4.3.4 Monitoring .............................................................. 73
4.3.5 Storage ................................................................. 73
SQLite .............................................................. 74
Plugin ........................................................... 75
4.3.6 Legacy ................................................................. 76
4.3.7 Domain Mapping ..................................................... 76
4.3.8 DomainParticipant ................................................... 77
4.3.9 Playback .............................................................. 79
Debug mode .......................................................... 81
4.3.10 Data Selection ....................................................... 82
4.3.11 Time Range .......................................................... 82
4.3.12 Session .............................................................. 83
4.3.13 Topic Group ........................................................ 85
4.3.14 Topic ................................................................. 86
4.3.15 Plugins .............................................................. 87

4.4 Remote Administration .................................................. 88
4.4.1 Enabling Remote Administration ................................ 88
4.4.2 Available Service Resources ...................................... 88
Example .............................................................. 89
4.4.3 Remote API Overview ............................................. 90
4.4.4 Replay Service ..................................................... 90

4.5 Monitoring ............................................................... 98
4.5.1 Overview .............................................................. 98
Enabling Service Monitoring ............................................. 98
Monitoring Types ......................................................... 98
4.5.2 Monitoring Metrics Reference ................................... 99
Service .............................................................. 99
Session ............................................................ 102
TopicGroup .......................................................... 103
Topic ............................................................. 104

4.6 Tutorials ................................................................. 105
4.6.1 Example: Getting Started with Replay and Shapes Demo 105
Start Shapes Demo and Subscribe to Squares .................. 105
Start Replay Service .................................................. 105
4.6.2 Example: Replaying Data at a Different Rate ....... 105
Edit the Replay Configuration ........................................ 105
Start Shapes Demo .................................................... 107
Start Replay Service ................................................ 107
4.6.3 Example: Plugging in Custom Storage ............ 107
Custom Storage API Overview ....................................... 107
# Indexing Application

## Indexing Instances

## Indexing SQLite Tables

## Running the Indexer

## Indexer Command-Line Parameters

# Software Development Kit

# Common Infrastructure

## Configuring RTI Services

### How to Load and Select an XML Configuration

#### Loading from Files

#### Loading from In-Memory Strings

#### Selecting which Configuration to Run

#### Default Files

#### XML Syntax and Validation

#### Listing Available Configurations

#### Configuration Variables

### How to Load Default QoS Profiles

### How to Set Logging Properties

#### Command-Line Options

#### Service API

#### XML Configuration

### How to Run as an Operating System Daemon

#### Linux and macOS Systems

#### Windows Systems

### Key Terms

## Application Resource Model

### Example: Simple Resource Model of a Connext DDS Application

### Resource Identifiers

#### Escaped Identifiers

#### Example: Resource Identifiers of a Generic Connext DDS Application

#### Example: Resource Identifiers Generated from XML Entity Model

## Remote Administration Platform

### Remote Interface

#### Standard Methods

#### Custom Methods

### Communication

#### Reply Sequence

#### Example: Accessing from Connext DDS Application

### Common Operations
Create Resource .......................... 167
Get Resource ............................. 168
Update Resource .......................... 169
Set Resource State ....................... 170
Get Resource State ....................... 170
Delete Resource .......................... 171

10.4 Monitoring Distribution Platform .................. 171
10.4.1 Distribution Topic Definition ................... 173
Example: Monitoring of Generic Application ........... 174
10.4.2 DDS Entities ......................... 175
10.4.3 Monitoring Metrics Publication .................. 175
Configuration Distribution Topic ....................... 175
Event Distribution Topic ........................ 176
Periodic Distribution Topic ....................... 176
10.4.4 Monitoring Metrics Reference .................. 176
Statistic Variable .......................... 177
Host Metrics ............................... 177
Process Metrics ............................. 178
Base Entity Resource Metrics ...................... 179
Network Performance Metrics ....................... 180
Thread Metrics ............................. 180

10.5 Plugin Management .................. 181
10.5.1 Shared Library ....................... 182
Configuration ............................. 182
10.5.2 Service API ......................... 184

11 Release Notes 185
11.1 Supported Platforms ...................... 185
11.2 Compatibility ........................... 186
11.3 What’s New in 6.1.0 ..................... 186
11.3.1 New platforms ........................ 186
11.3.2 Removed platforms ...................... 186
11.3.3 New message showing location for output recorded via RTI Launcher .......................... 186
11.3.4 Ability to replay by recorded source timestamp .................. 186
11.3.5 Instance History Replay .................. 187
11.3.6 Ability to index SQLite tables offline ............. 187
11.3.7 New debug mode for Replay Service .................. 187
11.3.8 Ability to change speed and jump in time during replay .......................... 187
11.3.9 Improved automatic recognition of recorded databases by SQLite tools .................. 188
11.3.10 New XML element to wait for acknowledgment .................. 188
11.3.11 Repetitive and misleading message repeated at end of recording (rollover not enabled) 188
11.3.12 Extended -verbosity command-line parameter to specify service and DDS verbosity independently .................. 188
11.3.13 Service monitoring now reports CPU usage for individual Session threads .................. 188
11.3.14 Added a Service constructor that receives a shutdown hook .................. 189
11.3.15 New API to execute administration commands directly on a Service .................. 189
11.4 What’s Fixed in 6.1.0 ...................... 189
11.4.1 Non-zero throughput monitoring metrics were provided while service was paused .......... 189
11.4.2 Executable did not log build ID for DDS libraries .......................... 189
11.4.3 XML Converter, rtxmlconverter, did not show its version properly ........ 189
11.4.4 XML Converter threw “multiple deserialization modes” warning in some cases .... 190
11.4.5 Segmentation fault when using non-compliant XML configuration on a QNX system 190
11.4.6 Segmentation fault after discovering application that used UserData QoS Policy ... 190
11.4.7 Replay Service did not detect termination correctly .......................... 190
11.4.8 Windows Debug assertions thrown when running in debug mode ............. 191
11.4.9 Monitoring Config Topic contained incomplete information about Topics created from TopicGroups .................................................. 191
11.4.10 Crash if monitoring enabled and TopicGroup matched with Topic name containing ‘/’ character ......................................................... 191
11.4.11 Replay Monitoring reported increasing current timestamp while service was paused 191
11.4.12 RTI Converter did not properly convert Sequence Numbers from SampleInfo field 192
11.4.13 Recording/Replay Service Crashed when Receiving Wrong Remote Admin Request for Updating Service State ........................................ 192
11.5 Known Issues .............................................................................. 192
11.5.1 Recording Service may fail when current working directory in c:\Program Files ... 192

12 Copyrights 193

Index 194
Welcome to RTI® Recording Service, an RTI Connext® DDS application that records and replays DDS Topics and discovery data.
Chapter 1

Introduction

1.1 Introduction

*RTI Recording Service* includes the following tools:

- *Recording Service*, an *RTI Connext DDS* application that records Topics and discovery data. *Recording Service* records updates to data along with a timestamp, so you can view or replay updates to data in your system as they occur over time. Recorded data is stored in SQLite® files by default. *Recording Service* also has an API to record to a custom data store.

- *Replay Service*, an application that can play back data recorded by *Recording Service*. *Replay Service* also has an API to allow plugging in custom storage.

- *Converter*, an application that converts binary (serialized) recorded data to deserialized data that can be viewed and queried. The most efficient way to record data is in serialized form. *Converter* allows data to be recorded efficiently, then post-processed into a queryable form. *Converter* also provides APIs to plug in custom storage.

1.2 The Basics

*Recording Service* is used when you need to record updates to system data over a period of time, and to access that data by time. One example of when you might use the *Recording Service*: if you are testing your system, you can use *Recording Service* to record all DDS data updates that occur during a particular test run. Then you can use a database tool or *Replay Service* to view what happened at specific times during your test.

*Recording Service's* built-in database is a SQLite file. *Recording Service* can store data in the SQLite database in two formats:

- CDR serialized format: This is the format in which data is sent over the network, so it is the most efficient way to store data. It is binary, so it is not queryable.

- JSON format: To store data this way, *Recording Service* must convert between the network format and the JSON format. This is queryable, but comes at a performance cost.
Recording Service also provides an API that allows you to implement your own storage backend. You will receive the data in serialized format, then use Connext DDS dynamic-data APIs to deserialize the data.

![Diagram of Recording Service]

Figure 1.1: Recording Service receives DDS samples over the network and records them to storage, based on their timestamps

Replay Service works with data recorded by Recording Service. It uses the timestamps of the recorded data to replay the data back into a DDS system at the original offsets between recorded timestamps. Replay Service also offers an API to retrieve your data from the storage of your choice to be replayed.

![Diagram of Replay Service]

Figure 1.2: Replay Service retrieves stored data samples from storage and replays it over DDS, maintaining the original timing of the data

Converter is a tool that can convert between the data formats used by Recording Service. Currently, the only built-in formats supported by Recording Service are the CDR and JSON formats described above, within a SQLite database. Converter allows you to record data in the efficient CDR format and later convert to a queryable JSON format.

### 1.3 Paths Mentioned in Documentation

This documentation refers to:

- `<NDDSHOME>` This refers to the installation directory for Connext DDS.
  
  The default installation paths are:
  
  - macOS® systems: `/Applications/rti_connext_dds-version`
  - Linux® systems, non-root user: `/home/your user name/rti_connext_dds-version`
Figure 1.3: *Converter* takes serialized (binary) data and converts it to deserialized JSON data

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

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

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

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

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

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

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

Default path to the examples:

- macOS systems: `/Users/your user name/rti_workspace/version/examples`
- Linux systems: `/home/your user name/rti_workspace/version/examples`
- Windows systems: your Windows documents folder\rti_workspace\version\examples. Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 10 systems, the folder is `C:\Users\your user name\Documents`. 
Chapter 2

Installation

RTI Recording Service comes pre-installed as part of RTI Connext DDS. Contact support@rti.com for information on how to obtain a package for your platform.
Chapter 3

Recording Service

3.1 Usage

This section explains how to run Recording Service from a command line. In particular, it describes:

- How to Start Recording Service (Section 3.1.1)
- How to Stop Recording Service (Section 3.1.2)
- Recording Service command-line parameters (Section 3.1.3)

Recording Service can also be deployed as a native library linked into your application on select architectures. For details on using Recording Service as a library, see Section 3.6.3.

3.1.1 Starting Recording Service

Recording Service runs as a separate application. The script to run the executable is in <NDDSHOME>/bin. (See Section 1.3 for the path to NDDSHOME.)

To start Recording Service with a default configuration, enter:

```
rtirecordingservice
```

This command will run Recording Service indefinitely until you stop it.

You can run Recording Service using the command-line parameters defined below.

Recording Service is pre-loaded with a builtin configuration that has default settings. See Section 3.3.14.

Note: To run Recording 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.

### 3.1.2 Stopping Recording Service

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

### 3.1.3 Recording Service Command-Line Parameters

The following table describes all the command-line parameters available in *Recording Service*. To list the available parameters, run `rtirecordingservice -help`.

All command-line parameters are optional; if specified, they override the values of any corresponding settings in the loaded XML configuration. See Section 3.3.2 for the XML elements that can be overridden with command-line parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-appName &lt;string&gt;</code></td>
<td>Application name used to identify this execution for remote administration and to name the Connext DDS participant.</td>
</tr>
<tr>
<td><code>-cfgFile &lt;string&gt;</code></td>
<td>Semicolon-separated list of configuration file paths. <strong>Default:</strong> unspecified</td>
</tr>
<tr>
<td><code>-cfgName</code></td>
<td>Configuration name used to find a <code>&lt;recording_service&gt;</code> matching tag in the configuration file.</td>
</tr>
<tr>
<td><code>-domainIdBase &lt;int&gt;</code></td>
<td>This value is added to the domain IDs in the <code>&lt;domain_participant&gt;</code> tag in the configuration file. For example, if you set <code>-domainIdBase</code> to 50 and use domain IDs 0 and 1 in the configuration file, then <em>Recording Service</em> will use domains 50 and 51 for recording (though the recorded data will appear as though it was recorded in domains 0 and 1). <strong>Default:</strong> 0</td>
</tr>
<tr>
<td><code>-D&lt;name&gt;=&lt;value&gt;</code></td>
<td>Defines a variable that can be used as an alternate replacement for XML environment variables, specified in the form <code>${VAR_NAME}</code>. Note that definitions in the environment take precedence over these definitions.</td>
</tr>
<tr>
<td><code>-help</code></td>
<td>Shows this help.</td>
</tr>
<tr>
<td><code>-heapSnapshotDir</code></td>
<td>Output directory where the heap monitoring snapshots are dumped. The filename format is: RTI_heap_&lt;appName&gt;<em>&lt;processId&gt;</em>&lt;index&gt;.log</td>
</tr>
<tr>
<td><code>-heapSnapshotPeriod &lt;sec&gt;</code></td>
<td>Period at which heap monitoring snapshots are dumped. Enables heap monitoring if &gt; 0. <strong>Default:</strong> 0 (disabled)</td>
</tr>
</tbody>
</table>

continues on next page
Table 3.1 – continued from previous page

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| -logFormat <format> | A mask to configure the format of the log messages for both Recording Service and Connext DDS:  
  • DEFAULT - Print message, method name, and activity context  
  • TIMESTAMPED - Print message, method name, activity context, and timestamp  
  • MINIMAL - Print only message number and method name  
  • MAXIMAL - Print all available fields  
  **Default**: DEFAULT |
| -maxObjectsPerThread <int> | Maximum number of thread-specific objects that can be created.  
  **Default**: 1024. |
| -remoteAdministrationDomainId <int> | Enables remote administration and sets the domain ID for communication.  
  **Default**: Remote administration is not enabled. |
| -remoteMonitoringDomainId <int> | Enables remote monitoring and sets the domain ID for status publication.  
  **Default**: Remote monitoring is not enabled. |
| -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 program version and exits. |

### 3.1.4 Controlling Recording Service’s Operation Mode

*Recording Service* can operate in different modes, regarding when data are written into disk.

In standard operation mode, *Recording Service* will store samples as they arrive. This is a purely reactive model. This mode is the default mode and is the best mode for scalability. To further improve performance and scalability, standard operation mode can be used in combination with the <thread_pool> settings for Sessions (see Section 3.3.8 for more information). These settings allow you to control the number of worker threads that process the data events, as well as the thread priorities, mask, etc.

In flush mode, you can define a flush period (see Section 3.3.6 for more information on how to enable it). When a flush period is defined, *Recording Service* will operate in a periodic fashion. This means that user-data samples will be written to disk only every time the flush period elapses.

The third operation mode is called buffering mode. This mode is controlled by setting the <enable_buffering_mode> in the storage configuration of the service (see Section 3.3.6). Unlike the other modes above, when operating in buffering mode, *Recording Service* will not do any writing to disk automatically. Instead, it will buffer samples in memory and wait for remote administration flush() commands to actually write them to disk.
Controlling Buffer Size in Buffering Mode

As mentioned above, there is an operation mode in Recording Service called buffering mode. This mode will not perform any automatic writing of samples to disk. Instead, this mode will wait for the reception of remote flush() commands that you send on demand.

Note: Recording Service will still write DDS discovery information automatically, even if buffering mode is enabled. This is done to ensure compatibility with Replay Service and Converter, by making sure no discovery samples are missing in the database files. Replay Service and Converter rely on this information to be able to function properly.

In between reception of flush commands, Recording Service will buffer samples in RAM memory, using the sample caches in its DataReaders. This buffering can lead to situations of unlimited memory growth. To avoid this problem, there are two options:

- Issue remote flush commands at a rate that allows Recorder’s writing to disk to keep up with the sample read rate. This rate is complex to calculate in many scenarios; if you find that you are sending flushing commands periodically, then it’s better to just use the <flush_period> setting and operate in periodic mode.

- Limit the size of the DataReader caches. This will allow the caches to work as a configurable, finite circular buffer with a fixed, immutable, oldest-first replacement policy. The following section explains the QoS properties used to control the DataReader’s cache.

Configuring In-Memory Buffer Size

The most important policy to work with is the History QoS policy. There are optional settings that can help improve the performance or that can affect the History QoS. Assuming that the intent is to configure Recording Service to keep a buffer of size N, the following table shows the DataReader QoS settings that should be used or are desirable (defined as optional):

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>History.kind</td>
<td>KEEP_LAST</td>
</tr>
<tr>
<td>History.depth</td>
<td>N</td>
</tr>
<tr>
<td>(Optional) ResourceLimits.max_samples</td>
<td>N</td>
</tr>
<tr>
<td>(Optional) ResourceLimits.max_instances</td>
<td>N</td>
</tr>
<tr>
<td>(Optional) ResourceLimits.max_samples_per_instance</td>
<td>N</td>
</tr>
<tr>
<td>(Optional) ResourceLimits.initial_samples</td>
<td>N</td>
</tr>
<tr>
<td>(Optional) ResourceLimits.initial_instances</td>
<td>N</td>
</tr>
</tbody>
</table>

Setting initial samples and instances is optional but recommended for performance because setting these properties will force the DataReader queue to start with a size of N samples directly, avoiding memory allocations during Recording Service’s operation.

3.1. Usage
Another important setting to take into account is `ReaderResourceLimits.max_samples_per_read`. This setting defines the maximum number of samples that a single read or take operation can return in one go. This setting will interact with the size of the buffer when a `flush` remote command is issued. When this setting is smaller than the buffer’s size, N, then Recording Service will need to generate several writing events, until N is reached or until there are no more samples to read.

See the following sections in the RTI Connext DDS Core Libraries User’s Manual for more information on these settings:

- History QoS Policy
- Resource Limits QoS Policy
- Reader Resource Limits QoS Policy

### 3.2 Operating System Daemon

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

### 3.3 Configuration

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

#### 3.3.1 Built-in Configuration of Recording Service

Recording Service is pre-configured with a built-in configuration. See Section 3.3.14 for more details.

#### 3.3.2 XML Tags for Configuring Recording Service

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

<table>
<thead>
<tr>
<th>Tags within <code>&lt;dds&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;qos_library&gt;</code></td>
<td>Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for the Connext DDS QoS profile file—see Configuring QoS with XML in the RTI Connext DDS Core Libraries User’s Manual.</td>
<td>0..*</td>
</tr>
</tbody>
</table>
3.3.3 Recording Service Tag

A configuration file must have at least one `<recording_service>` tag. This tag is used to configure an execution of *Recording Service*.

A configuration file may contain multiple `<recording_service>` tags. When you start *Recording Service*, you can use the `-cfgName` command-line parameter to specify which `<recording_service>` tag to use to configure the service. This means one file can be used to configure multiple *Recording Service* executions.

The following diagram and Table 3.4 describe the tags allowed within a `<recording_service>` tag.

### Table 3.4: Recording Service Tags in Recording Service's Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;recording_service&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;administration&gt;</code></td>
<td>Enables remote administration. When administration is enabled, monitoring is also enabled by default. If no domain ID is specified for monitoring, <em>Recording Service</em> will use the same domain as administration by default. See Section 3.3.4.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
### Table 3.4 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;recording_service&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;monitoring&gt;</code></td>
<td>Enables monitoring for the recording service, including statistics. See Section 3.3.5.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;storage&gt;</code></td>
<td>Describes how the data will be stored. If this is not specified, data will be stored in a SQLite file using the default name “rti_recorder_default.db”. See Section 3.3.6.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
| `<domain_participant>`           | **Required.** Specifies a DomainParticipant to use to record data. Attributes • name: Uniquely defines a DomainParticipant. **Required.**
Example
```xml
<domain_participant name="Participant3">
    <domain_id>3</domain_id>
    <!-- Participant QoS -->
</domain_participant>
```
See Section 3.3.7. | 1..* |
| `<session>`                      | **Required.** Active component of Recording Service for recording data. Contains one or more threads that can be used for recording. Attributes • name: Uniquely defines a recording session. **Required.** • default_participant_ref: Specifies a default DomainParticipant to be used by children of this recording session. Children can override this by specifying their own participant.
Example
```xml
<session name="Session" default_participant_ref="Participant3">
    <!-- ... topics / groups of topics to record -->
</session>
```
See Section 3.3.8. | 1..* |

### Example: Specify a Recording Service Configuration in XML

```xml
<dds>
    <recording_service name="MyRecorderService">
        <!-- ... Required entities -->
    </recording_service>
</dds>
```

Starting a Recording Service instance with the following command will use the `<recording_service>` tag with the name “MyRecorderService”:

```
$NDDSHOME/bin/rtirecordingservice -cfgFile file.xml -cfgName MyRecorderService
```
Figure 3.1: Top-level Tags in Recording Service’s Configuration File
Figure 3.2: Tags used to configure a Recording Service instance
3.3.4 Administration

The `<administration>` tag allows you to enable and configure remote administration of *Recording Service*, including stopping, starting, and pausing recording.

See Section 3.4 for details on using remote administration.

### Table 3.5: Administration Tags in Recording Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;administration&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;domain_id&gt;</code></td>
<td>Domain ID used for remote administration. Also used for monitoring by default.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;participant_qos&gt;</code></td>
<td>QoS used by the administration DomainParticipant. If the tag is not defined, <em>Connext DDS</em> defaults will be used.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;publisher_qos&gt;</code></td>
<td>QoS used by the administration Publisher. If the tag is not defined, <em>Connext DDS</em> defaults will be used.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;subscriber_qos&gt;</code></td>
<td>QoS used by the administration Subscriber. If the tag is not defined, <em>Connext DDS</em> defaults will be used.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
| `<datawriter_qos>` | QoS used by administration DataWriter(s). If the tag is not defined, *Connext DDS* defaults will be used, with the following changes:  
  • history.kind = DDS_KEEP_ALL_HISTORY_QOS  
  • resource_limits.max_samples = 32 | 0..1 |
| `<datareader_qos>` | Quality of Service (QoS) used by administration DataReader(s). If the tag is not defined, *Connext DDS* defaults will be used, with the following changes:  
  • reliability.kind = DDS_RELIABLE_RELIABILITY_QOS (this value cannot be changed)  
  • history.kind = DDS_KEEP_ALL_HISTORY_QOS  
  • resource_limits.max_samples = 32 | 0..1 |
| `<distributed_logger>` | When you enable Distributed Logger, *Recording Service* will publish its Log messages to *Connext DDS*. See Section 3.3.13. | 0..1 |

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

The `<monitoring>` tag allows you to enable and configure remote monitoring of Recording Service. See Section 3.5.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;monitoring&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;enabled&gt;</code></td>
<td>Whether to enable monitoring of the service.</td>
<td>0..1</td>
</tr>
<tr>
<td></td>
<td>Default: Disabled, unless administration is enabled.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;domain_id&gt;</code></td>
<td>Domain ID used for monitoring.</td>
<td>0..1</td>
</tr>
<tr>
<td></td>
<td>Default: The domain ID specified for monitoring.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;datawriter_qos&gt;</code></td>
<td>QoS used by monitoring DataWriter(s)</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;publisher_qos&gt;</code></td>
<td>QoS used by monitoring Publisher(s)</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;participant_qos&gt;</code></td>
<td>QoS used by monitoring DomainParticipant</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;statistics_sampling_period&gt;</code></td>
<td>How frequently to sample the service’s statistics, using the tags <code>&lt;sec&gt;</code> or <code>&lt;nanosec&gt;</code> . For example, <code>&lt;sec&gt;1&lt;/sec&gt;</code> samples the service’s statistics every second. Default: 1 second.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;status_publication_period&gt;</code></td>
<td>How frequently to publish the service status, using the tags <code>&lt;sec&gt;</code> or <code>&lt;nanosec&gt;</code> . For example, <code>&lt;sec&gt;1&lt;/sec&gt;</code> publishes the service’s status every second. Default: 5 seconds</td>
<td>0..1</td>
</tr>
</tbody>
</table>

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

3.3.6 Storage

The `<storage>` tag allows you to configure the storage to which data will be written. You can choose between using the builtin SQLite storage or implementing your own storage plugin. You can also specify the flush period of the service, defined as the time interval between consecutive writings of samples to disk. Within `<storage>`, Recording Service can also be set in buffering mode. This mode disables all automatic storage to disk and just writes the data to disk upon reception of a remote flush() command.
### Table 3.7: Storage Tags in Recording Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;storage&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;sqlite&gt;</td>
<td>Enables storing data in a SQLite database file. See Section 3.3.6.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;plugin&gt;</td>
<td>Enables storing data in an external library that you specify. Attributes: • plugin_name: Name of the plug-in that creates a storage plugin object. This name shall refer to a registered storage plug-in. See Section 10.5 for details on how the options of how to register plug-ins). See Section 3.3.6 for more about using this tag. See Section 3.6.4 for a tutorial on plugging in custom storage.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;flush_period&gt;</td>
<td>Defined as a duration (seconds and nanoseconds), &lt;flush_period&gt; represents the rate at which user-data samples will be written to disk. When this tag is present, Recording Service will work in a purely periodic fashion. If no remote flush() command is received, data will be written to disk only when this period elapses. This tag cannot be used at the same time as the &lt;enable_buffering_mode&gt; tag. Default: no default. If not set, Recording Service will work in a purely reactive way.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;enable_buffering_mode&gt;</td>
<td>When set to true, Recording Service will run in buffering mode. This mode is a listening-only mode and will not output data to disk automatically. Instead, Recording Service will wait continuously for remote flush() commands to trigger the storage to disk (see Section 3.4.5 for more information about the remote flush command). Recording Service will buffer samples using the caches in the DataReaders it creates, so it is important that the size of the caches is controlled so memory doesn’t grow indefinitely. The size can be controlled by controlling the size of the queue (see Section 3.1.4) or by ensuring that the rate at which flush remote commands are sent matches the sample rate. The general recommendation is to define a buffer size for all Topics and Topic Groups in the configuration. This tag cannot be used at the same time as the &lt;flush_period&gt; tag. This feature requires remote administration to be enabled. If remote administration isn’t enabled in the configuration, Recording Service will enable it automatically. Default: false (service will start in normal operation mode).</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;instance_indexing&gt;</td>
<td>When this setting is enabled, Recording Service will keep an internal index about instances and their values. It will also store this instance to disk once Recording Service shuts down. This can affect the recording performance. However, Replay Service startup time will be improved when the Instance History Replay setting is enabled (see Section 4.3.9). If this setting is not enabled, Replay will build the indexes while starting up, delaying the startup process. Indexing can also be done offline, see Section 8. See Section 3.3.6 for more details on instance indexing configuration.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
SQLite

The `<sqlite>` tag allows you to specify the name and file extension of a SQLite file in which to write data. It also allows you to choose the storage format in which to store the data. The default format, XCDR_AUTO, records data without deserializing it from the network format, so it is the most efficient way to store data; however, it is a binary format that cannot be queried without using Converter to convert it to a readable JSON_SQLITE format. The JSON_SQLITE format is slower to record, because it requires deserializing the data, but it can be queried using SQLite tools.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;sqlite&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;fileset&gt;</code></td>
<td>Set of files to write to, and parameters for creating files and directories in that set. See Section 3.3.6.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;file&gt;</code></td>
<td>File to write to. Default: rti_recorder_default</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;file_suffix&gt;</code></td>
<td>Allows you to add a suffix to the end of a filename.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;overwrite_policy_kind&gt;</code></td>
<td>Whether Recording Service is allowed to overwrite files. The options are OVERWRITE or DO_NOT_TOUCH. When DO_NOT_TOUCH is selected, Recording Service cannot overwrite an existing file, even if the rollover functionality is enabled. Default: OVERWRITE</td>
<td>0..1</td>
</tr>
</tbody>
</table>
| `<storage_format>`    | Specifies what format the data is stored in. The options are:  
  - XCDR_AUTO: This is the binary format used by Connext DDS when sending data over the network. This has the highest performance for recording, but can only be viewed by using Converter to convert the data to a readable format, or by using Replay to replay the data. This will internally store data in XCDR or XCDR2 depending on the format received.  
  - JSON_SQLITE: This format can be queried, but recording in this format has lower performance because data must be deserialized before it can be stored.  
  - XCDR: The format to use when communicating with Connext DDS before 6.0.0.  
  - XCDR2: More efficient than XCDR, used by Connext DDS 6.0.0 and later. Default: XCDR_AUTO | 0..1 |
### Table 3.8 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;sqlite&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<sql_initialization_string>` | Specifies a SQLite SQL expression to use when establishing sqlite connections using this plugin. This can be used to change the pragmas used by SQLite, or to do other database operations.  
**Note:** when using Recording Service and another application (either Replay Service or another SQLite application) at the same time to access the same database files, we recommend using SQLite’s WAL (write-ahead logging) mode. This can be done by adding `PRAGMA JOURNAL_MODE = WAL;` to this configuration setting. More information about SQLite’s WAL mode can be found [here](#).  
Default: `PRAGMA SYNCHRONOUS = OFF; PRAGMA JOURNAL_MODE = MEMORY;` | 0..1          |

### Fileset

The `<fileset>` tag allows you to specify a set of files for Recording Service to write to. This lets you specify behaviors such as “create a new directory with each run of Recording Service based on the timestamp when the tool was started” or “create a new file every time Recording Service is started, incrementing an integer in the filename.”

The `<fileset>` tag is also where the rollover behavior is specified.

### Table 3.9: Fileset Tags in Recording Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;fileset&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<workspace_dir>`       | Base directory where the database files for an instance of Recording Service (including discovery and user data files) will be stored. Depending on the value of the `<execution_dir_expression>` tag, this will either contain a set of files or a set of directories.  
Default: The current working directory. | 0..1         |

continues on next page
### Table 3.9 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;file-set&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;execution_dir_expression&gt;</code></td>
<td>When <em>Recording Service</em> starts, it will use this expression to create the directory where output files will be stored. Every time <em>Recording Service</em> starts, it will evaluate this expression to decide on its output directory. Stopping and restarting remotely will cause <em>Recording Service</em> to re-evaluate this expression and possibly change its output directory. This execution directory is a parameterisable expression. In it, it accepts text and any combination of the following:</td>
<td>0..1</td>
</tr>
</tbody>
</table>
| Autonumeric. **Format:** `%auto:M-N%`. This parameter describes an integer that auto-increments every time *Recording Service* starts. The numeric sequence is restarted when *Recording Service* is manually shut down and restarted. M must be lower than N; together they define a numeric range, both inclusive. N can be omitted (%auto:M%), resulting in an unlimited sequence of numbers starting at M. **Example:**  
```xml
<execution_dir_expression>
  test_run_%auto:0-3%
</execution_dir_expression>
```
This example will create directories named test_run_0, test_run_1, test_run_2, and test_run_3. | |
| Timestamp. **Format:** `%ts%`. This parameter will take the current timestamp in the system (the time represented as number of seconds since Epoch). | |
| Time. **Format:** `%T%`. Current time expressed in ISO 8601 time format (HH:MM:SS). Example: 14:55:02. This parameter uses the strftime() parameter `%T`. | |
| Short date. **Format:** `%F%`. Short date in YYYY-MM-DD format. Example: 2001-08-23. This parameter uses the strftime() parameter `%F`. | |
| Date and time. **Format:** `%c%`. Date and time representation, locale-dependent. This parameter uses the strftime() parameter `%c`. **Note:** Using parameters, *Recording Service* will check if the possible execution directories exist before overwriting any directory. If the execution directories exist, *Recording Service’s* behavior will be affected by the value of the `<overwrite_policy_kind>` tag. If `<overwrite_policy_kind>` is set to OVERWRITE, *Recording Service* will overwrite the first directory. Otherwise, *Recording Service* will not delete any of the old directories and will just exit. Using no parameters will yield the same execution directory every time the service is started. In this case, if the directory already contains database files, they may be overwritten (see the `<overwrite>` tag). Default: `%ts%` (current timestamp number since Epoch). | |

continues on next page
Table 3.9 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;file-set&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;filename_expression&gt;</td>
<td>Once Recording Service knows the exact directory in which to put the database files, it will use this parameter to determine the name(s) of the user data file(s) to be created. Right before the recording starts and every time Recording Service has to change its current file to a new one, it will use this parameterisable expression to generate the next file’s name. This setting accepts text and any combination of the following parameters:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Autonumeric. <strong>Format: %auto:M-N%.</strong> This parameter describes an integer that auto-increments every time Recording Service is started. However, the numeric sequence is restarted with every execution of the Recording Service application. M must be lower than N; together M and N define a numeric range, both inclusive. N may be omitted (%auto:M%), resulting in an unlimited sequence of numbers starting at M. <strong>Example:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;filename_expression&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>test_files_%auto:0-2%.db</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/filename_expression&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This will create files named test_files_0.db, test_files_1.db, and test_files_2.db. When a rollover event occurs, Recording Service will either create one of these files, or overwrite the next file in the sequence. When Recording Service is restarted, this will start over with overwriting test_files_0.db.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Timestamp. <strong>Format: %ts%.</strong> This parameter will take the current timestamp in the system (the time represented as number of seconds since Epoch).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Time. <strong>Format: %T%.</strong> Current time expressed in ISO 8601 time format (HH:MM:SS). Example: 14:55:02. This parameter uses the strftime() parameter %T.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Short date. <strong>Format: %F%.</strong> Short date in YYYY-MM-DD format. Example: 2001-08-23. This parameter uses the strftime() parameter %F. <strong>Note:</strong> This parameter will not vary in 24 hours, so use with caution in combination with the rollover time limit feature (time limit should be greater than 1 day; otherwise, you may overwrite the same file continuously).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Date and time. <strong>Format: %c%.</strong> Date and time representation, locale-dependent. This parameter uses the strftime() parameter %c. <strong>Note:</strong> Using no parameters will yield the same file name every time. Therefore, if a file rollover command is received or scheduled, Recording Service will be stopped (no more data can be stored without overwriting the current, and only, file). Default: rti_recorder_default_%auto:0%.db (auto-numeric starting at zero, unlimited).</td>
<td></td>
</tr>
<tr>
<td>&lt;rollover&gt;</td>
<td>Configuration for rolling over the file after a size or time limit is reached. See Section 3.3.6.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
Rollover

Rollover enables Recording Service to overwrite the oldest data file created by the current execution of the service when the last file in the set has reached a maximum size, or when a time limit is reached.

Note: In this release, rollover is only supported when the <filename_expression> tag specifies an auto-numeric filename. (See example below.)

```xml
<recording_service name="RolloverExample">
  <storage>
    <sqlite>
      <storage_format>JSON_SQLITE</storage_format>
      <fileset>
        <workspace_dir>output_dir</workspace_dir>
        <execution_dir_expression></execution_dir_expression>
        <!-- Rollover file when the size hits 50 KB. Files will be numbered 0-9. -->
        <filename_expression>file_rollover_%auto:0-9%.db</filename_expression>
        <rollover>
          <enabled>true</enabled>
          <file_size_limit unit="KILOBYTES">50</file_size_limit>
        </rollover>
      </fileset>
    </sqlite>
  </storage>
</recording_service>
```
### Table 3.10: Rollover Tags in Recording Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;rollover&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enabled&gt;</td>
<td>Whether <em>Recording Service</em> will roll over files when a limit is reached. Default: False.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
| <file_size_limit>     | The maximum allowed size for a file in a set. Note that setting this to a very low value (e.g., 1 KB) may yield unexpected behavior, because SQLite will take up more than that for even the simplest file. **Note:** The unit refers to the decimal prefix and not the binary prefix of the number, meaning 1 MEGABYTES = 1000 KB (and not 1024 KB). This is usually the standard way to refer to storage size. **Attributes:**  
  - unit: (Optional) The unit in which the size is expressed. The following values are allowed (Default: KILOBYTES):  
    - BYTES  
    - KILOBYTES  
    - MEGABYTES  
    - GIGABYTES | 0..1         |
| <time_limit>          | The maximum amount of time *Recording Service* can record to a file in a set. Specified with tags <days>, <hours>, <minutes>, <seconds>. **Attributes:**  
  - start_time: The time to do the first rollover. After that, rollover will be done when the time_limit or file_size_limit is reached. | 0..1         |

### Plugin

### Table 3.11: Storage plugin Tag in the Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;plugin&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <property>           | Name/value pairs of properties to pass to a storage plugin. **Example:**  
  
  ```xml  
  <property>  
    <value>  
      <element>  
        <name>Name</name>  
        <value>Value</value>  
      </element>  
    </value>  
  </property>  
  ``` | 0 or 1       |

### 3.3. Configuration
Instance Indexing

Table 3.12: Instance Indexing Tag in the Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;instance_indexing&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enabled&gt;</td>
<td>Set this to true to enable instance indexing. It’s recommended to enable instance indexing only when Replay Service is going to use Instance History Replay mode. You can also perform instance indexing offline, see Section 8. Default: false.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| <timestamps>                    | The type of timestamp (reception, source or both) to use when building the instance history index. The options are:  
  - RECEPTION: Create the index based on the time the DDS sample was received by the DataReader.  
  - SOURCE: Create the index based on the time the DDS sample was written by the DataWriter.  
  - BOTH: Create the index based on both the source timestamp and the reception timestamp.  
  Default: RECEPTION. | 0 or 1 |

3.3.7 DomainParticipant

Table 3.13: DomainParticipant Tags in Recording Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;domain_participant&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;domain_id&gt;</td>
<td>Required. DDS domain ID used for recording.</td>
<td>1</td>
</tr>
<tr>
<td>&lt;participant_qos&gt;</td>
<td>QoS used by this DomainParticipant. See Configuring QoS with XML, in the RTI Connext DDS Core Libraries User's Manual.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

continues on next page
Table 3.13 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;domain_participant&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;memory_management&gt;</td>
<td>Configures certain aspects of how Connext DDS allocates internal memory. The configuration is per DomainParticipant and therefore affects all the contained DDS entities.</td>
<td>0..1</td>
</tr>
<tr>
<td></td>
<td><strong>Example:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;memory_management&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;sample_buffer_min_size&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/sample_buffer_min_size&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;sample_buffer_trim_to_size&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>true</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/sample_buffer_trim_to_size&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/memory_management&gt;</td>
<td></td>
</tr>
<tr>
<td><strong>Tags within this tag:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• &lt;sample_buffer_min_size&gt;: For all DataWriters and DataReaders, the way Connext DDS allocates memory for samples is as follows: Connext DDS pre-allocates space for samples up to size X in the DataWriter and DataReader 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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• &lt;sample_buffer_trim_to_size&gt;: 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.</td>
<td></td>
</tr>
</tbody>
</table>

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.

continues on next page
### Table 3.13 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;domain_participant&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <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 a <topic>. **Attributes:**
|                                 |             |              |
|                                 | • name: Name that the data type is registered with if no <registered_name> is specified. The same data type may be registered with different names. **Required.**
|                                 | • type_ref: Definition of this data type. It must refer to one of the defined types in the <types> section by specifying the fully qualified name. **Tags within this tag:**
|                                 |             |              |
|                                 | • <registered_name>: Name the data type is registered with. The same data type may be registered with different names. Not required. | 0..* |

### 3.3.8 Session

The <session> tag configures the threads that will be used to record data. You also specify the Topics and groups of Topics to record inside the <session> tag.

#### Table 3.14: Session Tags in Recording Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;session&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;subscriber_qos&gt;</td>
<td>Specifies the QoS of DDS subscribers that will be used by the contained &lt;topic&gt; and &lt;topic_group&gt;. See Configuring QoS with XML, in the RTI Connext DDS Core Libraries User’s Manual.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

continues on next page
Table 3.14 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;session&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;thread_pool&gt;</td>
<td>Defines the number of threads used by this session to process Topics and Topic Groups and allows you to set the mask, priority, and stack size of each thread. This setting can improve the reactivity and scalability of the running Recording Service when there are multiple Topics and Topic Groups associated with the same session.</td>
<td>0..1</td>
</tr>
<tr>
<td></td>
<td>Example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 3.3.9 Topic Group

You can record a group of Topics, using regular expressions to describe which Topics to record.

#### Table 3.15: Topic Group Tags in Recording Service's Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;topic_group&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<allow_topic_name_filter>` | A regular expression (fnmatch) describing which Topics are allowed to be recorded. You may use a comma-separated list to specify more than one filter. **Example:**

```xml
<topic_group name="RecordAll">
  <allow_topic_name_filter>CONTROL_*,DATA_*</allow_topic_name_filter>
</topic_group>
```

| `<deny_topic_name_filter>` | A regular expression (fnmatch) describing which Topics are not allowed to be recorded. This tag is applied after the `<allow_topic_name_filter>` tag. You may use a comma-separated list to specify more than one filter. | 0..1 |
| `<allow_type_name_filter>` | A regular expression (fnmatch) describing the names of data types that are allowed to be recorded. You may use a comma-separated list to specify more than one filter. | 0..1 |
| `<deny_type_name_filter>` | A regular expression (fnmatch) describing the names of data types that are not allowed to be recorded. This tag is applied after the `<allow_type_name_filter>` tag. You may use a comma-separated list to specify more than one filter. | 0..1 |
| `<content_filter>` | A ContentFilteredTopic to use when recording data. See ContentFilteredTopics, in the RTI Connext DDS Core Libraries User's Manual. This allows you to record data samples only if their contents pass a filter that you specify in your configuration. The filter looks like a SQL WHERE clause. Note that XML reserved characters must be converted to their escape values; for example, > and < (greater than and less than signs) must be converted to &gt; and &lt; as shown in the example below. **Example:**

```xml
<topic_group name="RecordAll">
  <content_filter>
    <!-- Data will only be recorded when x is greater than 100 -->
    <expression>x &gt; 100</expression>
  </content_filter>
</topic_group>
```
| `<datareader_qos>` | The DataReader's QoS to use when recording data. | 0..1 |

---

### 3.3. Configuration
### 3.3.10 Topic

The `<topic>` tag specifies an individual Topic to record.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;topic&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;topic_name&gt;</code></td>
<td>The name of the DDS topic to be recorded. If this tag is not present, the name attribute of the <code>&lt;topic&gt;</code> will be used. Note: we recommend using this tag to define the topic name. There may be characters that cause the XML validation to fail if they are part of the topic name attribute. Also, the ‘/’ character and ‘::’ separator may cause Recorder to fail when found in the topic name attribute.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;registered_type_name&gt;</code></td>
<td>Required. The name of the data type that will be recorded for this topic.</td>
<td>1</td>
</tr>
<tr>
<td><code>&lt;transformation&gt;</code></td>
<td>The transformation library to be applied to this Topic’s data when recording. This is a user library that can modify the data after it is received by Connext DDS and before it is stored in the database. Transformations implement APIs identical to Routing Service’s transformations. For more on using transformations, see these sections in the RTI Routing Service User’s Manual: Data Transformation, Tutorials. Attributes: plugin_name: The name of the plugin to load, qualified by the plugin library name.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

**Example:**

```xml
<dds>
    <plugin_library name="RecordTransformations">
        <transformation_plugin name="ModifyTestID">
            <create_function>ModifyTestID_create</create_function>
            <dll>modify_test_id_library</dll>
        </transformation_plugin>
    </plugin_library>
    <!-- ... -->
    <recording_service>
        <topic name="TestTopic">
            <transformation plugin_name="RecordTransformations::ModifyTestID" />
        </topic>
    </recording_service>
</dds>
```

| `<datareader_qos>` | The DataReader QoS to use when recording this data. | 0..1 |

continues on next page
### Table 3.16 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<content_filter>` | A ContentFilteredTopic to use when recording data. See ContentFilteredTopics, in the RTI Connext DDS Core Libraries User’s Manual. Example of how to set a content filter expression:  

<topic_group name="RecordAll">  
<content_filter>  
<!-- Data will only be recorded when x -> is greater than 100 -->  
<expression>x &gt; 100</expression>  
</content_filter>  
</topic_group>                                                                                       | 0..1          |

### 3.3.11 Support for RTI FlatData and Zero Copy Transfer Over Shared Memory

Recording Service supports communication with applications that use RTI FlatData™ and Zero Copy transfer over shared memory. You can configure Recording Service to enable these capabilities for data reception.

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

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

**Example: Configuration to enable both FlatData and Zero Copy transfer over shared memory**

```xml
<dds>
  <types>
    <struct name="Point" transferMode="shmem_ref" languageBinding="flat_data" extensibility="final">
      <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>
      </domain_participant_qos>
    </qos_profile>
  </qos_library>
</dds>
```

(continues on next page)
3.3.12 Plugins

All the pluggable components specific to Recording Service are configured within the <plugin_library> tag. Table 3.17 describes the available tags.

Plug-ins are categorized and configured based on the source language. Recording Service supports C/C++ plug-ins.

<table>
<thead>
<tr>
<th>Tags within &lt;plugin_library&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;storage_plugin&gt;</td>
<td>Specifies a C/C++ Storage plug-in. See Table 10.18 and Section 3.3.6.</td>
<td>0..*</td>
</tr>
<tr>
<td>&lt;transformation_plugin&gt;</td>
<td>Specifies a C/C++ Transformation plug-in. See Table 10.18 and Section 3.3.10.</td>
<td>0..*</td>
</tr>
</tbody>
</table>
### 3.3.13 Enabling Distributed Logger

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

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

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

```xml
<recording_service name="RecordAll">
    <administration>
        ...
        <distributed_logger>
            <enabled>true</enabled>
        </distributed_logger>
        ...
    </administration>
</recording_service>
```

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

### 3.3.14 Recording Service Builtin Configuration Details

The *Recording Service* builtin configuration specifies:

- Recording all non-RTI Topics
- In domain 0
- Into a SQLite file named rti_recorder_default.db
- In the efficient XCDR format

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="../schema/rti_recording_service.xsd">
    <!-- Qos Library -->
    <qos_library name="QosLib">
        <qos_profile name="ReliableQos">
            <datareader_qos>
                <reliability>
                    <kind>RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
                <history>
                    <kind>KEEP_ALL_HISTORY_QOS</kind>
                </history>
            </datareader_qos>
        </qos_profile>
    </qos_library>
</dds>
```

(continues on next page)
<recording_service name="RecorderService">
    <!-- Top-level storage settings -->
    <storage>
        <sqlite>
            <file>rti_recorder_default</file>
            <file_suffix>dat</file_suffix>
            <storage_format>XCDR_AUTO</storage_format>
        </sqlite>
    </storage>
    <!-- Top-level domain settings -->
    <domain_participant name="Participant0">
        <domain_id>0</domain_id>
    </domain_participant>
    <session name="DefaultSession">
        <topic_group name="RecordAll" participant_ref="Participant0">
            <allow_topic_name_filter>*</allow_topic_name_filter>
            <deny_topic_name_filter>rti/*</deny_topic_name_filter>
            <datareader_qos base_name="QosLib::ReliableQos" />
        </topic_group>
    </session>
</recording_service>
</dds>

### 3.4 Remote Administration

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

**Note:** Recording Service remote administration is based on the RTI Remote Administration Platform described in Section 10.3. Please refer to that manual for a detailed discussion on the workings of remote administration in Recording Service.

Below you will find an API reference for all the supported operations.
3.4.1 Enabling Remote Administration

By default, remote administration is disabled in Recording Service.

To enable remote administration you can use the `<administration>` XML tag (see Section 3.3.4) or the `-remoteAdministrationDomainId` command-line parameter (see Section 3.1.3). Both of these methods enable remote administration and set the domain ID for remote communication.

3.4.2 Available Service Resources

Table 3.18 lists the public resources specific to Recording Service. Each resource identifier is expressed as a hierarchical sequence of identifiers, including parent and target resources. (See Section 10.2.2 for details.)

In the table below, the elements (rs), and (st) 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:

```
<recording_service name="MyRecorder">...
```

The resource identifier is:

```
/recording_services/MyRecorder
```

In the table below, the resource identifier is written as /recording_services/(rs), where (rs) is the service name. (st) is the storage 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>
<thead>
<tr>
<th>Resource</th>
<th>Resource Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recording Service</td>
<td>/recording_services/(rs)</td>
</tr>
<tr>
<td>Storage</td>
<td>/recording_services/(rs)/storage/(st)</td>
</tr>
</tbody>
</table>

**Example**

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

**Recording Service**

Entity with name “MyRecorder”:

```
<recording_service name="MyRecorder">...
```

Resource identifier:

```
/recording_services/MyRecorder
```
Storage

Entity with name “sqlite” (implicit name of the builtin storage):

```
<recording_service name="MyRecorder">
    <sqlite>...</sqlite>
</recording_service>
```

Resource identifier:
```
/recording_services/MyRecorder/storage/sqlite
```

3.4.3 Remote API Overview

Table 3.19: Remote Interface Overview

<table>
<thead>
<tr>
<th>Resource</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recording Service</td>
<td>DELETE /recording_services/(rs)</td>
<td>Shuts down a running Recording Service instance.</td>
</tr>
<tr>
<td>Recording Service</td>
<td>UPDATE /recording_services/(rs):flush</td>
<td>Flushes all buffered data in memory into disk.</td>
</tr>
<tr>
<td></td>
<td>UPDATE /recording_services/(rs)/state</td>
<td>Sets a Recording Service state.</td>
</tr>
<tr>
<td>builtin SQLite Storage</td>
<td>UPDATE /recording_services/(rs)/storage/sqlite:rollover</td>
<td>Continues the current recording in a new file (also known as a “shard” or file segment). Note: Only valid for builtin SQLite plugin.</td>
</tr>
<tr>
<td></td>
<td>UPDATE /recording_services/(rs)/storage/sqlite:tag_timestamp</td>
<td>Associates a symbolic name with the current time (with an optional offset) in the recording. Note: Only valid for builtin SQLite plugin.</td>
</tr>
</tbody>
</table>

3.4.4 Recording Service

DELETE /recording_services/(rs)
Operation shutdown

This operation will cause Recording Service to shutdown.

UPDATE /recording_services/(rs):flush
Operation flush

This operation will cause Recording Service to flush contents of all topics and topic groups into permanent storage. This operation can be used with any of Recording Service’s modes of operation, but is crucial.
when running in buffering mode, because it’s the only way to flush buffered samples into permanent storage.

This operation will store a total of N samples into storage, N being the depth of the History QoS policy. If there are fewer samples in the buffer, this operation will store all of them.

This operation will affect the whole service, meaning that all Topics and Topic Groups will be affected by it.

<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/recording_services/MyRecorder:flush</td>
</tr>
</tbody>
</table>

Request body

- Empty.

**UPDATE /recording_services/(rs)/state**

Operation **set_state**

Sets the state of a Recording Service object. The action is parametrized on octet_body, which could have the following values:

See **Set Resource State** (Section 10.3.3).

Valid requested states:

- STARTED
- STOPPED
- PAUSED
- RUNNING

Example

To pause an instance of Recording Service with the name “MyRecorder”:

<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/recording_services/MyRecorder/state</td>
</tr>
</tbody>
</table>
| octet_body       | to_cdr_buffer(RTI::Service::EntityStateKind::PAUSED)
3.4.5 Storage

**UPDATE** /recording_services/(rs)/storage/sqlite:rollover

**Operation** rollover

If the storage plugin being used is the builtin SQLite plugin, this operation will cause the recording to switch to a new file segment (also known as a “shard”) in the fileset.

<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/recording_services/MyRecorder/storage/sqlite:rollover</td>
</tr>
</tbody>
</table>

**Request body**

- Empty.

**UPDATE** /recording_services/(rs)/storage/sqlite:tag_timestamp

**Operation** tag_timestamp

If the storage plugin being used is the builtin SQLite plugin, this operation will introduce a new entry in the symbolic timestamps table, using the tag name, textual description, and offset time specified in the arguments to the operation (CDR-serialized in the octet-body with the RTI::RecordingService::DataTagParams data type).

- Example

To tag a moment with the name “/example/test1/tag1”, representing a moment that is 123 milliseconds in the past:
### 3.5 Monitoring

This section provides documentation on Recording Service remote monitoring.

**Note:** Recording Service monitoring is based on the Monitoring Distribution Platform described in Section 10.4. We recommend that you read Section 10.4 before using Recording Service monitoring.

#### 3.5.1 Overview

**Enabling Service Monitoring**

By default, monitoring is disabled in Recording Service. To enable monitoring you can use the `<monitoring>` tag (see Section 3.3.3) or the `-remoteMonitoringDomainId` command-line parameter, which enables remote monitoring and sets the domain ID for data publication (see Section 3.1.3).
Monitoring Types

The available Keyed Resource classes and their types that can be present in the distribution monitoring topics are listed in Table 3.20. The complete type relationship is shown in Figure 3.3.

<table>
<thead>
<tr>
<th>Keyed Class</th>
<th>Resource</th>
<th>Config</th>
<th>Event</th>
<th>Periodic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>ServiceConfig</td>
<td>ServiceEvent</td>
<td>ServicePeriodic</td>
<td></td>
</tr>
<tr>
<td>Session</td>
<td>SessionConfig</td>
<td>SessionEvent</td>
<td>SessionPeriodic</td>
<td></td>
</tr>
<tr>
<td>TopicGroup</td>
<td>TopicGroupConfig</td>
<td>TopicGroupEvent</td>
<td>TopicGroupPeriodic</td>
<td></td>
</tr>
<tr>
<td>Topic</td>
<td>TopicConfig</td>
<td>TopicEvent</td>
<td>TopicPeriodic</td>
<td></td>
</tr>
</tbody>
</table>

All the type definitions for Recording Service monitoring information are in [NDDSHOME]/resource/idl/ServiceCommon.idl and [NDDSHOME]/resource/idl/RecordingServiceMonitoring.idl.

Recording Service creates a DataWriter for each distribution Topic. All DataWriters are created from a single Publisher, which is created from a dedicated DomainParticipant. See Section 3.3.3 for details on configuring the QoS for these entities.
## 3.5.2 Monitoring Metrics Reference

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

### Service

#### Listing 3.1: *Recording Service* Types

```cpp
@mutable @nested
struct SqliteDatabaseConfig {
    Service::FilePath db_directory;
    @optional Service::FilePath execution_directory_expression;
    @optional Service::FilePath user_data_file_expression;
};

@mutable @nested
struct SqliteDatabaseEvent {
    @optional Service::FilePath current_db_directory;
    @optional Service::FilePath current_file;
    @optional int32 rollover_count;
};

@mutable @nested
struct SqliteDatabasePeriodic {
    @optional Service::FilePath current_file;
    @optional uint64 current_file_size;
    // These fields are no longer supported and carry no information.
    // Kept only to support older version.
    @deprecated int32 current_timestamp_sec;
    @deprecated uint32 current_timestamp_nanosec;
};

@mutable @nested
struct ParticipantInfo {
    Service::BoundedString name;
};

@mutable @nested
struct ServiceConfig : Service::Monitoring::EntityConfig {
    Service::BoundedString application_name;
    Service::Monitoring::ResourceGuid application_guid;
    @optional Service::Monitoring::HostConfig host;
    @optional Service::Monitoring::ProcessConfig process;
    @optional SqliteDatabaseConfig builtin_sqlite;
    @optional sequence<ParticipantInfo> participants;
};

@mutable @nested
struct ServiceEvent : Service::Monitoring::EntityEvent {
    // to avoid unused variable warnings
    @optional SqliteDatabaseEvent builtin_sqlite;
};
```

(continues on next page)
@mutable @nested
struct ServicePeriodic {
    @optional Service::Monitoring::HostPeriodic host;
    @optional Service::Monitoring::ProcessPeriodic process;
    int64 current_timestamp_nanos;
    @optional SqliteDatabasePeriodic builtin_sqlite;
};

Table 3.21: ServiceConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityConfig</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>application_name</td>
<td>Name of the Recording Service instance. The application name is provided through:</td>
</tr>
<tr>
<td></td>
<td>• appName command-line option when run as executable.</td>
</tr>
<tr>
<td></td>
<td>• ServiceProperty::application_name field when run as a library.</td>
</tr>
<tr>
<td>application_guid</td>
<td>GUID of the Recording Service instance. Unique across all service instances.</td>
</tr>
<tr>
<td>host</td>
<td>See Table 10.10.</td>
</tr>
<tr>
<td>process</td>
<td>See Table 10.12.</td>
</tr>
<tr>
<td>builtin_sqlite</td>
<td>See Table 3.22.</td>
</tr>
<tr>
<td>participants</td>
<td>Sequence of ParticipantInfo objects, one for each DomainParticipant inside the Recording Service. See Table 3.23.</td>
</tr>
</tbody>
</table>

Table 3.22: SqliteDatabaseConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_directory</td>
<td>Path to the base directory where the database files will live. This is the prefix directory, and still the execution directory expression below will be appended to it to determine the final directory.</td>
</tr>
<tr>
<td>execution_directory_expression</td>
<td>The expression used to generate the directory where the database files live. Note: this value is not set when running Replay Service. See Section 3.3.6</td>
</tr>
<tr>
<td>user_data_file_expression</td>
<td>The expression used to generate the names of the database files. Note: this value is not set when running Replay Service. See Section 3.3.6</td>
</tr>
</tbody>
</table>

Table 3.23: ParticipantInfo

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the DomainParticipant instance, as specified in the name attribute of the corresponding configuration tag.</td>
</tr>
</tbody>
</table>
Table 3.24: ServiceEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from</td>
<td>See Table 10.15.</td>
</tr>
<tr>
<td>EntityEvent</td>
<td></td>
</tr>
<tr>
<td>builtin_sqlite</td>
<td>See Table 3.25</td>
</tr>
</tbody>
</table>

Table 3.25: SqliteDatabaseEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_db_directory</td>
<td>Path to the current directory where files are being stored to. This is the actual final directory once the base directory and any execution directory expressions have been applied.</td>
</tr>
<tr>
<td>current_file</td>
<td>Path to the current file where data is being stored to. Note: this value is not set when running Replay Service.</td>
</tr>
<tr>
<td>rollover_count</td>
<td>The total number of file rollover events that have happened to this moment. Note: this value is not set when running Replay Service.</td>
</tr>
</tbody>
</table>

Table 3.26: ServicePeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>See Table 10.11.</td>
</tr>
<tr>
<td>process</td>
<td>See Table 10.13.</td>
</tr>
<tr>
<td>current_timeStamp_nanos</td>
<td>Time in nanoseconds Recording Service has been running.</td>
</tr>
<tr>
<td>builtin_sqlite</td>
<td>See Table 3.27</td>
</tr>
</tbody>
</table>

Table 3.27: SqliteDatabasePeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_file_size</td>
<td>The size in bytes of the current file where data is being stored. Note: this value is not set when running Replay Service.</td>
</tr>
</tbody>
</table>

Session

Listing 3.2: Session Types

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

@mutable @nested
struct SessionEvent : Service::Monitoring::EntityEvent {
    //to avoid unused variable warnings
    int32 _dummy;
};
```

(continues on next page)
@mutable @nested
struct SessionPeriodic {
    @optional Service::Monitoring::NetworkPerformance network_performance;
    @optional @optional Service::Monitoring::ThreadPoolPeriodic thread_pool;
};

Table 3.28: SessionConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityConfig</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>default_participant_name</td>
<td>The name of the default participant configuration.</td>
</tr>
</tbody>
</table>

Table 3.29: SessionEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityEvent</td>
<td>See Table 10.15.</td>
</tr>
</tbody>
</table>

Table 3.30: SessionPeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network_performance</td>
<td>Provides network performance metric as an aggregation of the same metric across the contained Topics and TopicGroups. See Section 10.4.4.</td>
</tr>
</tbody>
</table>

TopicGroup

Listing 3.3: TopicGroup Types

@mutable @nested
struct TopicGroupConfig : Service::Monitoring::EntityConfig {
    Service::BoundedString participant_name;
};

@mutable @nested
struct TopicGroupEvent : Service::Monitoring::EntityEvent {
    // to avoid unused variable warnings
    int32 _dummy;
};

@mutable @nested
struct TopicGroupPeriodic {
Table 3.31: TopicGroupConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityConfig</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>participant_name</td>
<td>Name of the DomainParticipant from which the Topic is created.</td>
</tr>
</tbody>
</table>

Table 3.32: TopicGroupEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityEvent</td>
<td>See Table 10.15.</td>
</tr>
</tbody>
</table>

Table 3.33: TopicGroupPeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network_performance</td>
<td>Provides network performance metric as an aggregation of the same metric across the contained Topics. See Section 10.4.4.</td>
</tr>
<tr>
<td>topic_count</td>
<td>Current number of Topics created from this TopicGroup.</td>
</tr>
</tbody>
</table>

Topic

Listing 3.4: Topic Types

```cpp
@mutable @nested
struct TopicConfig : Service::Monitoring::EntityConfig {
    Service::BoundedString topic_name;
    Service::BoundedString registered_type_name;
    Service::BoundedString participant_name;
    Service::Monitoring::ResourceGuid topic_group;
};

@mutable @nested
struct TopicEvent : Service::Monitoring::EntityEvent {
    //to avoid unused variable warnings
    int32 _dummy;
};

@mutable @nested
struct TopicPeriodic {
    @optional Service::Monitoring::NetworkPerformance network_performance;
```
Table 3.34: TopicConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityConfig</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>topic_name</td>
<td>Topic name as specified in the configuration.</td>
</tr>
<tr>
<td>registered_type_name</td>
<td>Topic registered type name as specified in the configuration.</td>
</tr>
<tr>
<td>participant_name</td>
<td>Name of the DomainParticipant from which the Topic is created.</td>
</tr>
<tr>
<td>topic_group</td>
<td>GUID of the TopicGroup from which this Topic was created. This field is set to zero for standalone Topics.</td>
</tr>
</tbody>
</table>

Table 3.35: TopicEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityEvent</td>
<td>See Table 10.15.</td>
</tr>
</tbody>
</table>

Table 3.36: TopicPeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network_performance</td>
<td>Provides network performance metric as an aggregation of the same metric across the contained Topics. See Section 10.4.4.</td>
</tr>
<tr>
<td>matched_status</td>
<td>Provides information about the matched endpoints associated with this Topic.</td>
</tr>
</tbody>
</table>

3.6 Tutorials

3.6.1 Getting Started with Recording Service and Shapes Demo

In this tutorial, you will edit a Recording Service configuration file to record only specific topics. Then you will start the Shapes Demo application and publish multiple topics. You will then inspect the database to see that only the specified topics were recorded.
Edit the Configuration

The first time you run any RTI tools, an rti_workspace directory is created in your home directory. (See Section 1.3 for the path to your workspace).

Inside the workspace, you will find a directory called user_config/recording_service. Open the USER_RECORDING_SERVICE.xml file in an editor. Notice that there are two <recording_service> tags in this file, one named UserRecorderService and one named UserRecorderServiceJson. The UserRecorderServiceJson configuration is shown below.

```xml
<recording_service name="UserRecorderServiceJson">
    <!-- Top-level storage settings -->
    <storage>
        <sqlite>
            <storage_format>JSON_SQLITE</storage_format>
            <fileset>
                <workspace_dir></workspace_dir>
                <execution_dir_expression>json_recording</execution_dir_expression>
                <filename_expression>rti_recorder_default_json.db</filename_expression>
            </fileset>
        </sqlite>
        <!-- Top-level domain settings -->
        <domain_participant name="Participant0">
            <domain_id>0</domain_id>
        </domain_participant>
        <session name="DefaultSession">
            <topic_group name="RecordAll" participant_ref="Participant0">
                <allow_topic_name_filter>*</allow_topic_name_filter>
                <deny_topic_name_filter>rti/*</deny_topic_name_filter>
            </topic_group>
        </session>
    </storage>
</recording_service>
```

This configuration is recording all topics in domain 0 into a database file named rti_recorder_default_json.db. It is recording in deserialized (JSON_SQLITE) mode. In the configuration, change the value in <allow_topic_name_filter> from * to Square. Now it will record only the Square topic.

```xml
<recording_service name="UserRecorderServiceJson">
    <!-- Top-level storage settings -->
    <storage>
        <sqlite>
            <storage_format>JSON_SQLITE</storage_format>
            <fileset>
                <workspace_dir></workspace_dir>
                <execution_dir_expression>json_recording</execution_dir_expression>
            </fileset>
        </sqlite>
    </storage>
</recording_service>
```

(continues on next page)
Start Shapes Demo

Use Launcher to start Shapes Demo.

![Shapes Demo from Launcher](image)

Figure 3.4: Open Shapes Demo from Launcher

In Shapes Demo, start two publishers: one Square publisher and one Circle publisher.
Figure 3.5: Create a new publisher in Shapes Demo
Start Recording Service

Start **Recording Service** with the **UserRecorderServiceJson** configuration and with verbosity level 3 by entering the following in a command shell:

```bash
cd <RTI_WORKSPACE>/user_config/recording_service
<NDDSHOME>/bin/rtirecordingservice -cfgName UserRecorderServiceJson -verbosity 3
```

See Section 1.3 for the path to your workspace and NDDSHOME.

You should see output indicating that the Square topic is being recorded:

```
[PARTICIPANT id=/recording_services/UserRecorderServiceJson/domain_participants/Participant0|STREAM_DISCOVERED name=Square|TOPIC id=/recording_services/UserRecorderServiceJson/sessions/DefaultSession/topics/RecordAll@Square|CREATE]
[PARTICIPANT id=/recording_services/UserRecorderServiceJson/domain_participants/Participant0|STREAM_DISCOVERED name=Square|TOPIC id=/recording_services/UserRecorderServiceJson/sessions/DefaultSession/topics/RecordAll@Square|ENABLE]
[TOPIC id=/recording_services/UserRecorderServiceJson/sessions/DefaultSession/topics/RecordAll@Square|START]
[TOPIC id=/recording_services/UserRecorderServiceJson/sessions/DefaultSession/topics/RecordAll@Square|RUN]
```

View the Data in Sqlite3

First, download the sqlite3 command-line browser from the SQLite download page [https://sqlite.org/download.html](https://sqlite.org/download.html). (If you are using a Linux or macOS system, you may already have this installed.)

Then open a command prompt and run the application:

```
sqlite3 json_recording/rti_recorder_default_json.db
```

Type the command `.tables` and you should see recorded data for topic Square:

```
sqlite> .tables
Square@0
```

Then you can view your recorded Square data by typing:

```
sqlite> select rti_json_sample from "Square@0";
{"color":"ORANGE","x":120,"y":195,"shapesize":30,"fillKind":"SOLID_FILL","angle":0}
{"color":"ORANGE","x":120,"y":197,"shapesize":30,"fillKind":"SOLID_FILL","angle":0}
{"color":"ORANGE","x":120,"y":199,"shapesize":30,"fillKind":"SOLID_FILL","angle":0}
{"color":"ORANGE","x":120,"y":201,"shapesize":30,"fillKind":"SOLID_FILL","angle":0}
```
3.6.2 Using Recording Service and Admin Console

You can use RTI Admin Console to monitor, pause and resume Recording Service.

Configuration

To use Recording Service with Admin Console, make sure that administration is enabled in the configuration. For example:

```
<recording_service name="AdminExample">
    <administration>
        <domain_id>0</domain_id>
    </administration>
    ...
</recording_service>
```

Note that enabling administration will also enable monitoring in the same domain by default. Admin Console cannot control the Recording Service instance unless monitoring is enabled.

Start Recording Service

Admin Console cannot start a new instance of Recording Service. It can only monitor and administer a instance of Recording Service that is already running.

To begin, start Recording Service with administration enabled:

```
<NDDSHOME>/bin/rtirecordingservice -cfgName AdminExample
```

Start Shapes Demo

Start Shapes Demo and publish Squares and Circles as described in Section 3.6.1.

Viewing with Admin Console

Use Launcher to start Admin Console (from the Tools tab).

You will see the Recording Service appear in Admin Console’s Physical View and Processes View. Click on the Recording Service in either view and a Recording Service tab will appear.

The first tab, RTI Recording Service Entities, shows your Recording Service, the session(s) it is running, the topics and topic groups it is configured to record, and which topics are being recorded.

Note that topics that are being recorded as part of a topic_group will appear side-by-side with topics that were configured individually. You can tell they are part of a topic_group because the name of the topic_group will appear along with the topic, such as RecordAll@Square.

The second tab, RTI Recording Service Configuration, displays the configuration that was used to configure the running Recording Service instance.
Figure 3.6: Open Admin Console from Launcher

Figure 3.7: View the Recording Service information in Admin Console
The third tab, **Graphical view**, displays the system in a graph. This allows you to visualize the entire system. You can see the DDS Entities that were discovered by Admin Console, including how they are connected and their matching endpoints.

The fourth tab, **DDS Entities**, displays the DDS entities that have been created by the Recording Service. If you click on individual DataReaders, you can view their QoS in the DDS QoS view.

The fifth tab, **RTI Recording Service Information**, shows details about the recording, such as:

- The **Connext DDS** version number
- The name of the database file it is recording (if you are using the builtin SQLite storage)
- The current size of the database file you are recording (if you are using the builtin SQLite storage)

The final tab, **Resource Charts**, allows you to monitor Recording Service’s CPU and memory usage.

**Administering with Admin Console**

*Recording Service* allows the following commands:

- **Pause**: This pauses all the topics in a running service.
- **Resume**: This restarts the sessions and topics in a *Recording Service* application.
- **Shutdown**: This shuts down the *Recording Service* application. To restart the application, you must re-run from *Launcher* or the command line.
Figure 3.9: View the Graphical View tab in Admin Console

Figure 3.10: View the DDS Entities tab in Admin Console
Figure 3.11: View the Recording Service Information tab in Admin Console

Figure 3.12: View the Resource Charts tab in Admin Console
3.6.3 Using Recording Service as a Library

It is possible to use Recording Service as a library in your application. All the necessary tools are included in the library rtirecordingservice (librtirecordingservice.so on Linux systems, librtirecordingservice.dylib on macOS systems, and rtirecordingservice.dll on Windows systems). The library can be used in any of the modes provided by Recording Service: recording data, replaying data or converting data. For more information, see: Recording Service C++ API.

A simple C++ example of how to link the Recording Service library into your application is available here: RTI Community Recording Service examples: Service as a Library.

Include files

When using Recording Service as a library, you should include the following header in your application code:

```
#include <rti/recording/RecordingService.hpp>
```
**Using the RecordingService class**

The main class used to instantiate *Recording Service* as a library is called `RecordingService`. To instantiate it, you need to define the properties to be used by the service for its configuration. The class used to define these properties is called `ServiceProperty`. Among the parameters that can be set to control the service's execution, you can find:

- **Application role**: record, replay or convert.
- **Configuration control**: configuration file as well as configuration name (service name).
- **Remote administration control**: flag to enable or disable it as well as the domain ID to use.
- **Monitoring control**: flag to enable or disable it as well as the domain ID to use.
- **DDS Domain ID control**: the domain ID base parameter allows the user to apply an offset to the domain IDs specified in the XML configuration to be loaded.
- **XML control**: whether to load the default Connext XML files or not or whether to enforce XSD validation of the supplied configuration file.

The following code snippet shows how to launch a *Recording Service* instance in record mode. This instance will use an XML configuration called `MyRecorderConfig` that can be found in file `my_recording_service.xml`. Administration and monitoring will be enabled and attached to domain ID 5. By using the domain ID base as well, all the domain IDs specified in the aforementioned XML configuration will be offset by 6.

```cpp
using namespace rti::recording;

// ...
ServiceProperty service_property;
service_property.application_role(ApplicationRoleKind::RECORD_APPLICATION).
   enable_administration(true).
   administration_domain_id(5).
   enable_monitoring(true).
   monitoring_domain_id(5).
   domain_id_base(6).
   cfg_file("my_recording_service.xml").
   service_name("MyRecorderConfig");
RecordingService recorder(service_property);
recorder.start();

// ...

// When it's time to stop the Recording Service instance, call the method
// below. The `stop()` method will also be called automatically by the
// RecordingService class destructor, so it will be called when the
// instance goes out of scope.
recorder.stop();
```
3.6.4 Plugging in Custom Storage

To configure a custom storage library, you must add the `<plugin_library>` tag inside the `<dds>` tag. This allows you to define one or more storage libraries that can be used to plug in custom storage.

In addition, you must associate the storage library with your Recording Service instance by specifying the plugin and its name in the `<storage>` tag.

You can pass custom properties to your plugin inside the `<plugin>` tag.

There are full examples written in C and C++ about plugging in custom storage in Recording Service, in: RTI Community Recording Service examples: C storage plugin and RTI Community Recording Service examples: C++ storage plugin.

Custom Storage API Overview

To store data, you must implement the following APIs:

- **RecordingServiceStorageWriter** create function: A creation function for the StorageWriter structure or class. This allocates a StorageWriter object, which is used to allocate StorageStreamWriter objects.

- **StorageWriter**:
  - A function for creating StorageStreamWriter objects for user-data topics when Recording Service notifies the plugin about a new stream. The user-data streams represent samples as Dynamic Data objects.
  - Three functions for creating StorageStreamWriter objects for the built-in discovery topics: DCPSParticipant, DCPSPublication and DCPSSubscription. These topics are represented by their specific types (e.g., DDS_ParticipantBuiltinTopicData type for DCPSParticipant). These functions are not required, thus when one of them is not implemented, no samples will be stored for that built-in topic.
  - A function for deleting user-data StorageStreamWriter objects. Recording Service expects this function to be able to work with streams writers created for user-data samples only.
  - Three functions to delete the StorageStreamWriter objects representing the DDS built-in topics. These functions are not required, but if the creation function was defined for a topic, the deletion function must also be defined.
  - A function for deleting the StorageWriter instance.

- **StorageSyncWriter**:
  - A function for storing data associated with a stream. For the stored data to be compatible with Replay Service, the reception timestamp of every sample should be stored with the data. It is also recommended that the valid data flag is recorded. For Replay Service to be able to replay instance states, it’s also necessary for Recording Service to store the instance state and instance handle fields.
  - There are three specific classes for the three different built-in discovery topics: StorageParticipantWriter, StoragePublicationWriter and StorageSub-
scriptionWriter. Each of them includes a store() function that is strongly typed to the topic’s type.

The C++ APIs provide a mechanism to have strongly typed StorageStreamWriter classes. There are four specific builtin StorageStreamWriter definitions to work with the different types of streams:

- A definition based on dds::core::xtypes::DynamicData, which should be used to store samples for user-data topics.
- A definition based on dds::topic::ParticipantBuiltInTopicData, which can be used to store samples of the builtin DDS discovery stream DCPSParticipant.
- A definition based on dds::topic::PublicationBuiltInTopicData, which can be used to store samples of the builtin DDS discovery stream DCPSPublication.
- A definition based on dds::topic::SubscriptionBuiltInTopicData, which can be used to store samples of the builtin DDS discovery stream DCPSSubscription.

More detailed API documentation is here:

- Recording Service C API documentation
- Recording Service C++ API documentation

### 3.6.5 Accessing JSON samples through SQL

When using the builtin SQLite JSON storage format, data samples are stored in the column called rti_json_sample using SQLite’s JSON extensions. The sample can thus be accessed using these extensions, namely, the json_extract() function.

As an example, suppose we have the following IDL type:

```c
struct BasicStruct {
    double member1;
    string member2;
};
```

We could access the sample’s data with a SQL query like this:

```
SELECT json_extract(rti_json_sample, '$.member2') FROM [MyTableName]
WHERE json_extract(rti_json_sample, '$.member1') > 2.0
```

### 3.6.6 Controlling Recording Service Remotely from an Application

Apart from the ability to use Admin Console to control a Recording Service instance, it is possible to control it using an application that issues command requests programmatically, using the Remote Administration Platform.

There is a C++ example in the RTI Community that provides an application that can produce command requests for Recording Service (or in general, for any RTI service that uses the common Remote Administration Platform): RTI Community Examples: C++ Service Administration.
3.6.7 Listing the Timestamp Tags in a Recording

If you are using the builtin SQLite plugin in Recording Service, and remote administration is enabled, then the remote administration exposes a command that allows you to tag timestamps with a symbolic name and description. These timestamp tags allow you to associate points in time in the recording with external events. The tags can be used later with Replay Service or Converter in place of timestamps to select what data to replay or convert.

The use case for timestamp tags is that they allow you to record, in a human-readable way, the time at which interesting events occur in your business processes along with the database that Recording Service is recording into. Later you can use these markers to replay data from (or up to) that point, instead of expressing start and end times in a numeric way.

For example, suppose you use Recording Service to record data while you are doing some business process. Then some event happens during the process that you want to somehow mark in the database. (Maybe this event marks the time at which some device starts behaving strangely during a troubleshooting session). If you create a custom GUI application that an operator can use, or any kind of application, you can use the timestamp tagging command described in this section. Then when that application sends the command, Recording Service will store in the database a record in which you can give the event a name and description. Later you can replay the recorded data starting (or ending) at that named event.

Note that you can use a time offset when submitting the timestamp tagging command. This allows you to create tags that refer to a time in the past or the future, relative to the time when you sent the tagging command.

Once you have a recorded database, you can list the timestamp tags that are in the recording. Use the command rtirecordingservice_list_tags and point it to the directory that contains your recorded database with the -d argument.

For example:

<NDSSHOME>/bin/rtirecordingservice_list_tags -d /database/directory/

This command will analyze the recording in /database/directory/ and list the details of any timestamp tags in the recording, including the tag names, descriptions, and associated timestamps. For example:

```
tag_name          timestamp_ms  tag_description
-----------------  ---------------  ------------------------
/my_example/my_events/tag1  1546484663309  first tag description
/my_example/my_events/tag2  1546484703360  a second tag description
```

For information on using timestamp tags with Replay Service, see Section 4.6.4. They can also be used with Converter, see Section 5.3.1.

A simple C++ example of how to use timestamp tags using remote administration is available here: RTI Community Examples: C++ Service Administration.
3.7 Troubleshooting

3.7.1 Verbosity

Increase the verbosity from the command line to view information about which Topics have been discovered. For example, running with `-verbosity 4` shows that `Recording Service` has discovered a Square Topic.

```
[recording_services/RecorderService/domain_participants/Participant0|STREAM_˓→DISCOVERED name=Square| type_name=ShapeType
[recording_services/RecorderService/domain_participants/Participant0|STREAM_˓→DISCOVERED name=Square|../../sessions/DefaultSession/topics/ ˓→RecordAll@Square|CREATE]
```

At verbosity level 4, it is also possible to see incompatible QoS.

```
[recording_services/RecorderService/sessions/DefaultSession/topics/ ˓→RecordAll@Triangle|START]
[recording_services/RecorderService/sessions/DefaultSession/topics/ ˓→RecordAll@Triangle|RUN]
FRESPsService_isRemoteWriterLocalReaderCompatible:incompatible ownership: ˓→writer 1 reader 0
[routing_services/RecorderService/domain_routes/DomainRoute/sessions/ ˓→DefaultSession/routes/RecordAll@Circle/inputs/DdsInput1] requested␣ ˓→incompatible qos: policy id=6, name=Ownership
```
Chapter 4

Replay Service

4.1 Usage

This section explains how to run Replay Service from a command line. In particular, it describes:

- How to Start Replay Service (Section 4.1.1).
- How to Stop Replay Service (Section 4.1.2).
- Replay Service command-line parameters (Section 4.1.3).

4.1.1 Starting Replay Service

Replay Service runs as a separate application. The script to run the executable is in <NDDSHOME>/bin. (See Section 1.3 for the path to NDDSHOME.)

rtireplayservice [options]

To start Replay Service with a default configuration, enter:

$NDDSHOME/bin/rtireplayservice

This command will run Replay Service indefinitely until you stop it.

Replay Service is pre-loaded with a built-in configuration that has default settings.

Note: To run Replay 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 rticommom_config.[sh/bat] to the name of the target architecture. (The file is resource/scripts/rticommom_config.sh on Linux or macOS systems, resource/scripts/rticommom_config.bat on Windows.
systems.) If the CONNEXTDDS_ARCH environment variable is set, the architecture in this file will be ignored.

4.1.2 Stopping Replay Service

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

4.1.3 Replay Service Command-Line Parameters

The following table describes all the command-line parameters available in Replay Service. To list the available parameters, run rtireplayservice -help.

All command-line parameters are optional; if specified, they override the values of any corresponding settings in the loaded XML configuration. See Section 4.3.1 for the XML elements that can be overridden with command-line parameters.

Table 4.1: Replay Service Command-Line Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-appName &lt;string&gt;</td>
<td>Application name used to identify this execution for remote administration, and to name the Connext DDS participant.</td>
</tr>
<tr>
<td>-cfgFile &lt;string&gt;</td>
<td>Semicolon-separated list of configuration file paths. <strong>Default:</strong> Unspecified</td>
</tr>
<tr>
<td>-cfgName</td>
<td>Configuration name Used to find a &lt;replay_service&gt; matching tag in the configuration file.</td>
</tr>
<tr>
<td>-debugMode</td>
<td>Enables debug mode. <strong>Default:</strong> Debug mode is not enabled.</td>
</tr>
<tr>
<td>-domainIdBase &lt;int&gt;</td>
<td>This value is added to the domain IDs in the &lt;domain_participant&gt; tag in the configuration file. For example, if you set -domainIdBase to 50 and use domain IDs 0 and 1 in the configuration file, Replay Service will read domains 0 and 1 from the database, but will replay that data into domains 50 and 51. <strong>Default:</strong> 0</td>
</tr>
<tr>
<td>-D&lt;name&gt;=&lt;value&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-help</td>
<td>Shows this help.</td>
</tr>
<tr>
<td>-heapSnapshotDir</td>
<td>Output directory where the heap monitoring snapshots are dumped. The filename format is: RTI_heap_&lt;appName&gt;_&lt;processId&gt;_index.log</td>
</tr>
<tr>
<td>-heapSnapshotPeriod &lt;sec&gt;</td>
<td>Period at which heap monitoring snapshots are dumped. Enables heap monitoring if &gt; 0. <strong>Default:</strong> 0 (disabled)</td>
</tr>
</tbody>
</table>

continues on next page
### Table 4.1 – continued from previous page

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-logFormat &lt;format&gt;</td>
<td>A mask to configure the format of the log messages for both <em>Replay Service</em> and <em>Connext DDS</em>.</td>
</tr>
<tr>
<td></td>
<td>• DEFAULT - Print message, method name, and activity context</td>
</tr>
<tr>
<td></td>
<td>• TIMESTAMPED - Print message, method name, activity context, and timestamp</td>
</tr>
<tr>
<td></td>
<td>• MINIMAL - Print only message number and method name</td>
</tr>
<tr>
<td></td>
<td>• MAXIMAL - Print all available fields</td>
</tr>
<tr>
<td></td>
<td><strong>Default:</strong> DEFAULT</td>
</tr>
<tr>
<td>-maxObjectsPerThread &lt;int&gt;</td>
<td>Maximum number of Thread-specific objects that can be created.</td>
</tr>
<tr>
<td></td>
<td><strong>Default:</strong> 1024.</td>
</tr>
<tr>
<td>-remoteAdministrationDomainId &lt;int&gt;</td>
<td>Enables remote administration and sets the domain ID for communication. <strong>Default:</strong> Remote administration is not enabled.</td>
</tr>
<tr>
<td>-remoteMonitoringDomainId &lt;int&gt;</td>
<td>Enables remote monitoring and sets the domain ID for status publication. <strong>Default:</strong> Remote monitoring is not enabled.</td>
</tr>
<tr>
<td>-verbosity &lt;service_level&gt;[:&lt;dds_level&gt;]</td>
<td>Controls what type of messages are logged. &lt;service_level&gt; is the verbosity level for the service logs and &lt;dds_level&gt; is the verbosity level for the DDS logs. Both can take any of the following values:</td>
</tr>
<tr>
<td></td>
<td>• SILENT</td>
</tr>
<tr>
<td></td>
<td>• ERROR</td>
</tr>
<tr>
<td></td>
<td>• WARN</td>
</tr>
<tr>
<td></td>
<td>• LOCAL</td>
</tr>
<tr>
<td></td>
<td>• REMOTE</td>
</tr>
<tr>
<td></td>
<td>• ALL</td>
</tr>
<tr>
<td></td>
<td><strong>Default:</strong> ERROR:ERROR</td>
</tr>
<tr>
<td>-version</td>
<td>Prints the program version and exits.</td>
</tr>
</tbody>
</table>

### 4.1.4 Replay Service Runtime Behavior

*Replay Service* currently does not delete *DataWriters*, even if all original *DataWriters* were deleted in the recorded database.

### 4.1.5 Working With Large Data

The built-in SQLite plugin implementation available in *Replay Service* is prepared to handle any type size and storage format. In order to improve the fidelity of the samples published with respect to the timestamps of the original samples in the database, *Replay Service* internally caches the next sample to be published, so that it can be accessed more quickly when it's time to publish. This behavior is particularly useful when replaying large data.

However, when working with large data types and massive files (or filesets), *Replay Service* may take a while to prepare SQL statements to work with the data, resulting in a delay in publishing the first samples for any large data topics.
Because of this delay, it is recommended that you index the user data tables for those large topics before running Replay Service on them. Indexing can massively improve Replay Service’s startup time for those topics. You can create the indexes offline, after Recording Service has finished recording all the data. Index the tables on the SampleInfo_reception_timestamp field. For example, imagine a table, VeryLargeTopic@0, has been created by Recording Service; you can use the following index creation statement:

```
CREATE INDEX IF NOT EXISTS [VeryLargeTopic@0_idx_rt] 
ON [VeryLargeTopic@0] (SampleInfo_reception_timestamp)
```

### 4.1.6 Choosing the Sample Order for Replaying Data

Replay Service has the capability to replay data ordered by reception timestamp or by source timestamp. Reception timestamp represents a monotonic ascending time series and the source timestamp comes from every recorded participant’s system clock, hence it can behave in non-monotonic fashion. The source timestamp can differ between one record and another with the same information.

Furthermore, it is important to consider the DDS_DestinationOrderQosPolicy, which can create “eventual consistency” between the different Recording Service instances. For more details, please see DESTINATION_ORDER QosPolicy, in the RTI Connext DDS Core Libraries User’s Manual.

That being said, Replay Service’s built-in SQLite plugin implementation will sort the database by using the chosen sample_order before it replays the data.

### 4.1.7 Recreating the State of the World when Replaying (Replaying Instance History)

Replay Service has the ability to replay what can be called the state of the world given a starting timestamp. The state of the world is the latest value for every alive instance at a certain timestamp. That’s why we also refer to this feature as Instance History Replay. When this feature is enabled, Replay Service will read the latest value for every instance that was alive, and publish it with the first batch of samples to be published for a topic. Of course this feature relates to keyed types and topics, for unkeyed types and topics it has no effect.

As an example, imagine a keyed topic \( T1 \) was recorded, and that there are three instances for the topic, define by IDs 1, 2 and 3. The following table shows the instances and values recorded for it during a certain period of time:

<table>
<thead>
<tr>
<th>Time</th>
<th>Instance ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ID=1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>ID=2</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>ID=3</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>ID=1</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>ID=3</td>
<td>310</td>
</tr>
<tr>
<td>6</td>
<td>ID=1</td>
<td>disposed</td>
</tr>
<tr>
<td>7</td>
<td>ID=2</td>
<td>210</td>
</tr>
<tr>
<td>8</td>
<td>ID=3</td>
<td>320</td>
</tr>
<tr>
<td>9</td>
<td>ID=1</td>
<td>120</td>
</tr>
<tr>
<td>10</td>
<td>ID=2</td>
<td>220</td>
</tr>
</tbody>
</table>
If the Instance History Replay feature is enabled, *Replay Service* will publish the instance values that compose the state of the world at the start time provided by the user. For example, for start time \( T=4 \), *Replay Service* will publish, ahead of any normal replay activity, samples \{ID=1,100\}, \{ID=2,200\} and \{ID=3,300\}. It will then start publishing samples normally, \{ID=1,110\}, etc. If start time is \( T=7 \), then the state of the world will be composed of two samples, because the sample with ID=1 was disposed at time \( T=6 \). Hence, *Replay Service* will publish samples \{ID=2,200\} and \{ID=3,310\} as the state of the world. Of course, if the start time is not provided (\( T=0 \)) then there is no history to replay.

An important note about how the instance history is replayed is that it’s published in a burst. This means that time separation between different values is not preserved. The goal of the feature is to publish a whole picture for the topic *before* starting with normal, time-preserving replay.

This feature can be useful in situations where very large databases are being replayed partially (this is, with a specific time range). When this is done, and this feature is not enabled, instances that were alive at the specified start time are just not present in the replay. By enabling this feature, *Replay Service* will publish a value for each instance that was alive at the start time provided, hence completing the whole picture for the topic before normal replay activity starts.

Under the hood, this feature uses a custom instance history index that is created by *Recording Service* (although the creation is disabled by default, as it can affect performance), or it can also be created offline. See Section 3.3.6 on how to enable this feature while recording, or Section 8 for how to index the database offline. If the index was not created while recording, or offline ahead of replaying, then *Replay Service* will create the index during startup, which can take some time in huge databases. So our general recommendation is to plan ahead whether this feature is going to be used when replaying your data and either use online indexing with *Recording Service* (by enabling the `<instance_indexing` tag) or save some time for the offline indexing of the database. It’s interesting to note that indexing, even when done by *Replay Service*, will only happen once, as it can be expected.

The searching for the instance history is quite optimized, but it takes some time. This delay can manifest during *Replay Service* startup, but no more searching will be needed after that during normal replay of samples. If if the looping feature is enabled, *Replay Service* will search for instance history again and re-publish it. This will restore the original state of the world ahead of replaying samples of the next loop.

This feature is integrated with replaying by source timestamp and other features, like jumping between timestamps. Take into account that if instance indexing was done, for example, based only for reception timestamp, but you want to Replay with instance history based on source timestamp, then *Replay Service* will proceed to indexing the database by source timestamp, incurring in the indexing delay. So planning ahead of time what sort of indexing and replay (source, timestamp) is going to be needed can also save time.

For more details on how to configure this feature, see Section 4.3.9.

### 4.1.8 Jumping in Time while Replaying

*Replay Service* has the capability of jumping in time. With this jump action, *Replay Service* will move the current replay position forward or backward in time.

For example, suppose you have a recording and in the middle there are some important events. During the replay, you can jump ahead to those events, skipping the unrelated events that came before.

When the jump in time is to the future, *Replay Service* will burst all the discovery samples between your previous position and your new position. This discovery phase can take some time depending on the amount of
discovery between both positions. To improve the performance of the jump, you can enable Instance History Replay to obtain the discovery state of the world and create all the new StreamReaders. For more information, see Section 4.1.7.

In order to jump in time, you have to enable the remote administration. Remote administration exposes some commands that allow you to perform this action. See Section 4.4.3 for details.

### 4.1.9 Using Debug Mode while Replaying

*Replay Service* has a secondary mode to start in a debug way. This mode will allow you to add breakpoints in the replay. The replay will be paused once it hits a breakpoint and the user will have the capability to decide the behavior of the replay.

For example, suppose you have a recording and in the middle there are some important events. During the replay, you can replay that section of events adding breakpoints at the beginning and at the end of the replay section. Once, the replay hits the end breakpoint you will be able to jump to the first breakpoint and reproduce that important event again.

If the Debug mode feature is enabled, *Replay Service* will first hit an initial breakpoint, which is a permanent breakpoint labeled “default_breakpoint”. This initial breakpoint is set to the start timestamp of the recorded database.

In this mode, you can continue the replay until hit a breakpoint or continue the replay for a period of time and then stop the replay without hitting a real breakpoint.

Under the hood, *Replay Service* will create a virtual breakpoint at the end of that period of time. The virtual breakpoint will be removed internally after it was hit or we jump to another breakpoint. This period of time can be a custom time set by the user or a fixed time that we call slice period. The slice period can be set inside the XML configuration and it can't be changed during the replay.

The debug mode use the remote administration system to receive the user’s order due to that you have to enable the remote administration to control this mode. See Section 4.4.3 for details.

### 4.2 Operating System Daemon

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

### 4.3 Configuration

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

This section describes the XML tags you can use in a Replay Service configuration file. Figure 4.1 and Table 4.2 describe the top-level tags allowed within the root `<dds>` tag.

![Diagram of XML Tags for Configuring Replay Service]

**Figure 4.1: Top-level Tags in the Replay Configuration File**

**Table 4.2: Top-Level Tags in Replay Service’s Configuration File**

<table>
<thead>
<tr>
<th>Tags within <code>&lt;dds&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;qos_library&gt;</code></td>
<td>Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for a Connext DDS QoS profile file — see Configuring QoS with XML, in the RTI Connext DDS Core Libraries User’s Manual.</td>
<td>0..*</td>
</tr>
<tr>
<td><code>&lt;types&gt;</code></td>
<td>Defines types that can be used by Replay Service. This tag is needed if data types are not available through discovery, or when using a transformation. The type description is done using the Connext DDS XML format for type definitions. See Creating User Data Types with Extensible Markup Language (XML), in the RTI Connext DDS Core Libraries User’s Manual.</td>
<td>0..*</td>
</tr>
</tbody>
</table>

continues on next page
Table 4.2 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;dds&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<plugin_library>`  | Contains a list of libraries that can be used to:  
• Plug in custom storage, such as custom databases. For more information, see Section 4.6.3.  
• Transform data after it is received from Connext DDS and before it is placed in storage. For more information, see Data Transformation, in the RTI Routing Service User’s Manual.. See Section 3.3.12 | 0..* |

| `<replay_service>`  | **Required.** Specifies a Replay Service configuration. See Section 4.3.2. Attributes  
• `name`: uniquely identifies a service configuration. Required.  
Example  
```xml
<replay_service name="ReplayAll">
  <!-- your service settings ... -->
</replay_service>
```
 | 1..* |

### 4.3.2 Replay Service Tag

A configuration file must have at least one `<replay_service>` tag. This tag is used to configure an execution of *Replay Service*.

A configuration file may contain multiple `<replay_service>` tags. When you start *Replay Service*, you can specify which `<replay_service>` tag to use to configure the service using the `-cfgName` command-line parameter. This means one file can be used to configure multiple *Replay Service* executions.

Figure 4.2 and Table 4.3 describe the tags allowed within a `<replay_service>` tag.

Table 4.3: Replay Service Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;replay_service&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;administration&gt;</code></td>
<td>Enables remote administration. When administration is enabled, monitoring is also enabled by default. See Section 4.3.3.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;monitoring&gt;</code></td>
<td>Enables monitoring for <em>Replay Service</em>, including statistics. See Section 4.3.4.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;storage&gt;</code></td>
<td>Describes how the data will be loaded from storage. See Section 4.3.5. If this is not specified, data will be loaded from the current working directory.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

continues on next page
### Table 4.3 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;replay_service&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;playback&gt;</code></td>
<td>Specifies the timing rules for how data is played back. See Section 4.3.9.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;data_selection&gt;</code></td>
<td>Enables selection of a subset of data to replay. Supports selecting replay data by time (or tagged time). See Section 4.3.10.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
| `<domain_participant>`         | **Required.** Specifies a DomainParticipant to use to replay data. The domain ID specified for the DomainParticipant will be used to determine which domains to select from the database and (combined with the domainIdBase) which domains to replay the data into. For example, if data was originally recorded in domains 0 and 1, and you want to replay in domains 50 and 51, you must:  
  - Specify `<domain_participant>` tags with `<domain_id>` tags set to 0 and 1 in your configuration file.  
  - Use `-domainIdBase 50` to specify that the domain IDs the data will actually be written into is offset by 50. See Section 4.3.8.  
  **Attributes:**  
  - `name`: Uniquely defines a DomainParticipant. **Required.**  
  **Example**  
  ```xml
  <domain_participant name="Participant3">
  <domain_id>3</domain_id>
  <!-- Participant QoS -->
  </domain_participant>
  ``` | 1..* |
| `<session>`                    | **Required.** Active component of `Replay Service` for replaying data. Contains one or more threads that can be used for replay. See Section 4.3.12.  
  **Attributes**  
  - `name`: Uniquely defines a replay session. **Required.**  
  - `default_participant_ref`: Specifies a default DomainParticipant to be used by topics and topic groups belonging to this replay session. Children can override this by specifying their own participant.  
  **Example**  
  ```xml
  <session name="Session" default_participant_ref="Participant3">
  <!-- ... topics / groups of topics to record -->
  </session>
  ``` | 1..* |
Figure 4.2: Tags used to configure a Replay Service instance
Example: Specify a Replay Service Configuration in XML

```xml
<dds>
  <replay_service name="MyReplayService">
    <!-- ... Required entities -->
  </replay_service>
</dds>
```

Starting a *Replay Service* with the following command will use the `<replay_service>` tag with the name “MyReplayService”.

```
$NDDSHOME/bin/rtireplayservice -cfgFile file.xml -cfgName MyReplayService
```

*Replay Service* uses the Topic names and domain IDs specified in the configuration file to determine which data to replay. To replay data into a different domain than the one it was recorded from, use the `-domainIdBase` command-line parameter to change the output domain ID.

Figure 4.3: The domain ID specified inside a DomainParticipant, in combination with the Topic name, is used to load data from recorded tables. In this case, the DomainParticipant’s domain ID is set to 3. The `topic_group` specifies all topics.

```
$NDDSHOME/bin/rtireplayservice -cfgName ReplayAll -domainIdBase 2
```

Figure 4.4: To replay data in a different domain than where it was recorded, use the `-domainIdBase` command-line parameter to specify a base ID for the replay domain. Here, by specifying a domainIdBase of 2, all Topics will be replayed in their original domain, plus 2.
4.3.3 Administration

The `<administration>` tag allows you to enable and configure remote administration of Replay Service, including stopping, starting, and pausing replay.

See Section 4.4 for details on using remote administration.

Table 4.4: Administration Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;administration&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;domain_id&gt;</code></td>
<td>Domain ID used for remote administration. Also used for monitoring by default.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;participant_qos&gt;</code></td>
<td>QoS used by the administration DomainParticipant. If the tag is not defined, Connext DDS defaults will be used.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;publisher_qos&gt;</code></td>
<td>QoS used by the administration Publisher. If the tag is not defined, Connext DDS defaults will be used.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;subscriber_qos&gt;</code></td>
<td>QoS used by the administration Subscriber. If the tag is not defined, Connext DDS defaults will be used.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
| `<datawriter_qos>`            | QoS used by administration DataWriter(s). If the tag is not defined, Connext DDS defaults will be used, with the following changes:  
  • history.kind = DDS_KEEP_ALL_HISTORY_QOS  
  • resource_limits.max_samples = 32 | 0..1          |
| `<datareader_qos>`            | Quality of Service (QoS) used by administration DataReader(s). If the tag is not defined, the Connext DDS defaults will be used, with the following changes:  
  • reliability.kind = DDS_RELIABLE_RELIABILITY_QOS  
  (this value cannot be changed)  
  • history.kind = DDS_KEEP_ALL_HISTORY_QOS  
  • resource_limits.max_samples = 32 | 0..1          |
| `<distributed_logger>`        | When you enable Distributed Logger, Replay Service will publish its Log messages to Connext DDS. See Section 3.3.13. | 0..1          |

The contents of the tags for configuring QoS are specified in the same manner as for the Connext DDS QoS profile file. See Configuring QoS with XML, in the RTI Connext DDS Core Libraries User’s Manual.
4.3.4 Monitoring

The `<monitoring>` tag allows you to enable and configure remote monitoring of *Replay Service*.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;monitoring&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;enabled&gt;</code></td>
<td>Whether to enable monitoring of the service. Default: Disabled unless administration is enabled</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;domain_id&gt;</code></td>
<td>Domain ID used for monitoring. Default: Use the domain ID specified for monitoring</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;datawriter_qos&gt;</code></td>
<td>QoS used by monitoring DataWriter(s)</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;publisher_qos&gt;</code></td>
<td>QoS used by monitoring Publisher(s)</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;participant_qos&gt;</code></td>
<td>QoS used by monitoring DomainParticipant</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;statistics_sampling_period&gt;</code></td>
<td>How frequently to sample the service’s statistics, using the tags <code>&lt;sec&gt;</code> or <code>&lt;nanosec&gt;</code>. For example, <code>&lt;sec&gt;1&lt;/sec&gt;</code> samples the service’s statistics every second. Default: 1 second.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;status_publication_period&gt;</code></td>
<td>How frequently to publish the service status, using the tags <code>&lt;sec&gt;</code> or <code>&lt;nanosec&gt;</code>. For example, <code>&lt;sec&gt;1&lt;/sec&gt;</code> publishes the service’s status every second. Default: 5 seconds</td>
<td>0..1</td>
</tr>
</tbody>
</table>

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

4.3.5 Storage

The `<storage>` tag allows you to configure the storage from which data will be read. You can choose between using the builtin SQLite storage or implementing your own storage plugin.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;storage&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sqlite&gt;</code></td>
<td>Enables replaying data from an SQLite database file See Section 4.3.5.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;plugin&gt;</code></td>
<td>Enables storing data in an external library that you specify. Attributes • <code>plugin_name</code>: Name of the plugin that creates a storage plugin object. This name shall refer to a registered storage plugin. See Section 10.5 for details on how to register plugins). See Section 4.3.5 for more about using this tag. See Section 4.6.3 for a tutorial on plugging in custom storage.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
SQLite

The `<sqlite>` tag allows you to specify a SQLite formatted file to read data from for replay.

### Table 4.7: SQLite Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;sqlite&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<storage_format>`     | The storage format of the user data files found in the database directory. If not specified, XCDR.AUTO is assumed. The options are:  
  - XCDR.AUTO: This is the binary format used by Connext DDS when sending data over the network. This has the highest performance for recording, but can only be viewed by using Converter to convert the data to a readable format, or by using Replay to replay the data. This will internally store data in XCDR or XCDR2 depending on the format received.  
  - JSON_SQLITE: This format can be queried, but recording in this format has lower performance because data must be deserialized before it can be stored.  
  - XCDR: The format to use when communicating with Connext DDS before 6.0.0.  
  - XCDR2: More efficient than XCDR, used by Connext DDS 6.0.0 and later.  
  Default: XCDR.AUTO. | 0..1 |
| `<database_dir>`       | The directory where to look for a recorded database. The directory must contain a valid metadata file, as well as a discovery file and one or many user data files. This field is optional. When not included, the current directory will be used.  
  Default: the current directory. | 0..1 |

continues on next page
Table 4.7 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;sqlite&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<sql_initialization_string>` | Allows configuring an SQLite SQL expression to use when establishing SQLite connections using this plugin. You can add an index to a table to speed up replay, but this must be done for each table you want to index. Note that you can perform this step using an SQLite client. **Example:**  
```
<sql_initialization_string>
    CREATE INDEX IF NOT EXISTS pingtopic_
    0_by_rcp_timestamp
    ON 'PingTopic@0' (SampleInfo_
    reception_timestamp);
</sql_initialization_string>
```
| 0..1 |

Note: when using Recorder and Replay at the same time for the same DB file(s), or for that matter, any other SQLite application accessing the data, we recommend using SQLite’s WAL (write-ahead logging) mode. This can be done by adding PRAGMA JOURNAL_MODE = WAL; to this configuration setting. More information about SQLite’s WAL mode can be found here.

Default: PRAGMA SYNCHRONOUS = OFF; PRAGMA JOURNAL_MODE = MEMORY;

Plugin

Table 4.8: Storage plugin Tag in the Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;plugin&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<property>` | Name/value pairs of properties to pass to a storage plugin. **Example:**  
```
<property>
    <value>
        <element>
            <name>Name</name>
            <value>Value</value>
        </element>
    </value>
</property>
```
| 0 or 1 |
4.3.6 Legacy

Table 4.9: Legacy Tags in the Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;legacy&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;file_path&gt;</td>
<td>Path to the legacy format recorded file. File set and version properties will be obtained automatically from the file itself.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;domain_mapping&gt;</td>
<td>This tag allows you to link legacy domain names with domain IDs. See Section 4.3.7.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

4.3.7 Domain Mapping

When converting a legacy database, the domain may not have been recorded. This tag provides a way to map a table with a domain ID.

Later versions of the old Recorder database allowed you to use field filters. By default, the domain ID field was not recorded (it was filtered out by default). Thus, there is no information available in these legacy databases to relate the domain name used to record the data with a domain ID. This tag allows you to link legacy domain names with domain IDs.

Table 4.10: Domain Mapping Tags in the Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;domain_mapping&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <domain_map>                | **Required.** A link between a recorded legacy domain name and a domain ID. **Attributes:**  
  • legacy_domain_name: Name of the recorded domain name as specified in the old Recording Service domain tag, for example: <domain name="domain0">.  
  • participant_ref: The name of a DomainParticipant specified below using a <domain_participant> tag. This DomainParticipant will be used to replay data specified by the legacy domain name. | 1..*          |
4.3.8 DomainParticipant

The `<domain_participant>` tag allows you to specify the DomainParticipants and domain IDs you want to query from the database for replay. You can replay in the same domain you used to record, or you can use the `-domainIdBase` command-line parameter to replay in a different domain.

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

<table>
<thead>
<tr>
<th>Tags within <code>&lt;domain_participant&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;domain_id&gt;</code></td>
<td>The domain ID of tables loaded from the database. This is used (in conjunction with the <code>-domainIdBase</code> command-line parameter) to choose which domain to replay into.</td>
<td>1</td>
</tr>
<tr>
<td><code>&lt;participant_qos&gt;</code></td>
<td>QoS used by this DomainParticipant.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

Table 4.11: DomainParticipant Tags in Replay Service’s Configuration File

continues on next page
Table 4.11 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;domain_participant&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <memory_management>              | Configures certain aspects of how Connext DDS allocates internal memory. The configuration is per DomainParticipant and therefore affects all the contained DDS entities. **Example:**<br>

```xml<br>
<memory_management><br>
  <sample_buffer_min_size><br>
    1024<br>
  </sample_buffer_min_size><br>
  <sample_buffer_trim_to_size><br>
    true<br>
  </sample_buffer_trim_to_size><br>
</memory_management><br>
```

This tag includes the following tags:

- `<sample_buffer_min_size>`: For all DataWriters and DataReaders, the way Connext DDS allocates memory for samples is as follows: Connext DDS pre-allocates space for samples up to size X in the DataWriter and DataReader 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.

continues on next page
Table 4.11 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;domain_participant&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <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 a <topic>. **Attributes:**  
  - name: Name that the data type is registered with if no <registered_name> is specified. The same data type may be registered with different names. **Required.**  
  - type_ref: Definition of this data type. It must refer to one of the defined types in the <types> section by specifying the fully qualified name.  
  **Tags within this tag:**  
  - <registered_name>: Name the data type is registered with. The same data type may be registered with different names. Not required. | 0..* |

4.3.9 Playback

The <playback> tag allows you to specify the timing rules for how the data is played back, such as the time on the local machine to start the replay, if it is not started immediately.

Table 4.12: Playback Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;playback&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <start_replay_local_time> | Selects the local time when Replay Service should start publishing samples. Specified in hours, minutes, and seconds. Specifying a full date is not currently supported, so if you specify a time that happened in the past (which Replay Service interprets as in the past today), replay will start immediately. **Example:**  
  
  `<start_replay_local_time>`  
  `<hour>14</hour>`  
  `<minute>30</minute>`  
  `<second>0</second>`  
  `</start_replay_local_time>`  
  The example above indicates that Replay Service will start the replay at 14:30 pm. Default: Replay will start immediately | 0..1 |
### Table 4.12 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;play-back&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <fidelity>             | Specifies the fidelity of the replayed data. Samples whose timestamp distance is less than the fidelity period may be published together in the same batch. For example, if your fidelity is 100 ms, data stored in your database within a 100 ms time period may be published together. **Tags within this tag:**  
  - `<sampling_period>`: the duration (in seconds and nanoseconds) that represents the maximum time period during which samples with different timestamps may be published together. Default: 1 ms | 0..1 |
| <rate>                 | Specifies the rate to replay data, as a double. For example, to play data at half speed, use 0.5. Default: 1 (normal speed) | 0..1 |
| <enable_looping>      | Plays the data in a repeating loop, until *Replay Service* is stopped. Note that this means that only a termination signal or a remote shutdown command will make it stop replaying data. When no time range data selection has been specified (see Section 4.3.10) then Replay will replay the whole contents of the database file(s) and when it's time to loop, it will start from the very beginning of the file or files. However, when a specific time range has been specified for replay, the looping will respect the time range and will restart at the specified start time and end at the specified end time. **Note:** in our default SQLite implementation, Replay is able to check if new data has been written by Recorder (or any application) for every given stream being replayed, since the start of the service. This feature is generally not supported when looping without specifying a time range and may yield unexpected results. Default: false. | 0..1 |
| <sample_order>        | Specifies the timestamp kind to replay data. If not specified, RECEPTION_TIMESTAMP is assumed. The options are:  
  - RECEPTION_TIMESTAMP: Time stored when the DDS sample was received by the DataReader. For more details, please see :link_connext_users_man_up_5_s: DESTINATION_ORDER QosPolicy, in the RTI Connext DDS Core Libraries User’s Manual */DESTINATION_ORDER_QosPolicy.htm».  
  - SOURCE_TIMESTAMP: Time stored by the DataWriter when the DDS sample was written. **Note:** For more information about source timestamp and reception timestamp, please see Section 4.1.6. Default: RECEPTION_TIMESTAMP. | 0..1 |

continues on next page
Table 4.12 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;play-back&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <instance_history_replay> | Specifies whether Instance History Replay (state of the world publication) is enabled. See Section 4.1.7 for a description of the feature. Tags within this tag:  
  • <enabled>: Whether to enable or disable the feature. When enabled, Replay Service will publish an initial value (or more, depending on the depth setting) for every alive instance before the specified start timestamp. After this startup publication, Replay Service will start normal replay. Default: false. | 0..1          |

**Debug mode**

The `<debug_mode>` tag allows you to enable the debug mode, such as set the slice period and the initial list of breakpoints.

Table 4.13: Debug Mode Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;debug_mode&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enabled&gt;</td>
<td>Enables debug mode.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;slice_period&gt;</td>
<td>Specifies the slice period of the replay. It will determine the replay period when you use a next_slice or continue operation. After that period of time Replay Service will pause the replay. Default: 1 s</td>
<td>0..1</td>
</tr>
</tbody>
</table>

continues on next page
### Table 4.13 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;debug_mode&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <initial_breakpoints>  | Specifies an initial list of breakpoints. Those breakpoints will be set once the replay start.  
**Tags within this tag:**  
• <element>: the timestamp (in nanoseconds) that will have the breakpoint. You can assign a label to the breakpoint as an identifier. Multiplicity: 1..*  
**Attributes:**  
  – label: Breakpoint identifier.  
**Example:**  
  ```xml
  <debug_mode>
    <enabled>true</enabled>
    <initial_breakpoints>
      <element>1600635588280996383</element>
      <element>label="MyBreakpoint">1600635598310952678</element>
    </initial_breakpoints>
  </debug_mode>
  ``` |
| 0..1                    |             |              |

### 4.3.10 Data Selection

Selection of data to replay from the database. Currently only supports selection using a time range.

#### Table 4.14: Data Selection Tags in Replay Service’s Configuration

<table>
<thead>
<tr>
<th>Tags within &lt;data_selection&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;time_range&gt;</td>
<td>Select data to replay from the database, based on start and end times. See Section 4.3.11.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

### 4.3.11 Time Range

The `<time_range>` tag allows you to specify the begin and end times of the data you want to replay. This can be specified either as timestamps or as symbolic timestamps called “timestamp tags.” These tags may have been added to the recording through remote administration, and can be viewed using the script `rtirecordingservice_list_tags` (see Section 3.6.7).
Table 4.15: Time Range Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;time_range&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;begin_time&gt;</td>
<td>Select data to replay from the database, with a timestamp beginning at this time. Specify the time in seconds and nanoseconds. Default: 0, start at beginning of the file.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;begin_tag&gt;</td>
<td>Can be used instead of begin_time. Select data to start replaying from the database by specifying the string name of a timestamp tag. Timestamp tags associate a timestamp with a name. Then you can refer to a timestamp by that name. The timestamp must have been tagged with that name during recording. See Section 3.6.6. Default: Start at beginning of the file.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;end_time&gt;</td>
<td>Select data to replay from the database, with a timestamp ending at this time. Specify the time in seconds and nanoseconds. Default: Stop at end of file.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;end_tag&gt;</td>
<td>Can be used instead of end_time. Select when to stop replaying data by specifying the string name of a timestamp tag. Timestamp tags associate a timestamp with a name. Then you can refer to a timestamp by that name. The timestamp must have been tagged with that name during recording. See Section 3.6.6. Default: Stop at end of file.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

4.3.12 Session

The <session> tag configures the threads that will be used to replay data. You also specify the Topics and groups of Topics to replay inside the <session> tag.

Table 4.16: Session Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;session&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;publisher_qos&gt;</td>
<td>Specifies the QoS of the Publisher that will be used by the contained &lt;topic&gt; and &lt;topic_group&gt;. For information on configuring Pubscriber QoS with XML, see Configuring QoS with XML, in the RTI Connext DDS Core Libraries User’s Manual.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

continues on next page
<table>
<thead>
<tr>
<th>Tags within <code>&lt;session&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<thread_pool>`         | Defines the number of threads used by this session to process Topics and Topic Groups and sets the mask, priority, and stack size of each thread.  

**Example:**  

```xml
<thread_pool>
    <mask>MASK_DEFAULT</mask>
    <priority>THREAD_PRIORITY_DEFAULT</priority>
    <stack_size>THREAD_STACK_SIZE_DEFAULT</stack_size>
</thread_pool>
```

Default values:  
- **size**: 1  
- **mask**: MASK_DEFAULT  
- **priority**: THREAD_PRIORITY_DEFAULT  
- **stack_size**: THREAD_STACK_SIZE_DEFAULT |

| `<topic>` | Specifies an individual Topic to replay. See Section 4.3.14.  

**Attributes:**  
- **name**: The name of the Topic to replay. This name is also used when monitoring and administering each Topic.  
- **participant_ref**: A DomainParticipant to use when replaying this Topic. If the parent `<session>` specifies a `default_participant_ref`, this attribute is optional. |

| `<topic_group>` | Specifies a group of Topics to replay. See Section 4.3.13.  

**Attributes:**  
- **name**: The name of the topic group. This name is used when monitoring and administering each topic group.  
- **participant_ref**: A DomainParticipant to use when replaying this topic group. If the parent `<session>` specifies a `default_participant_ref`, this attribute is optional. |
4.3.13 Topic Group

The `<topic_group>` tag allows you to replay a group of Topics, using regular expressions to describe which Topics to replay.

Table 4.17: Topic Group Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;topic_group&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<allow_topic_name_filter>` | A regular expression (fnmatch) describing which Topics are allowed to be replayed. You may use a comma-separated list to specify more than one filter. **Example:**

```xml
<topic_group name="ReplayAll">
  <allow_topic_name_filter>CONTROL_*,DATA_*</allow_topic_name_filter>
</topic_group>
```
| 0..1 |
| `<deny_topic_name_filter>` | A regular expression (fnmatch) describing which Topics are not allowed to be replayed. This is applied after the `allow_topic_name_filter`. You may use a comma-separated list to specify more than one filter. |
| 0..1 |
| `<allow_type_name_filter>` | A regular expression (fnmatch) describing the names of data types that are allowed to be replayed. You may use a comma-separated list to specify more than one filter. |
| 0..1 |
| `<deny_type_name_filter>` | A regular expression (fnmatch) describing the names of data types that are not allowed to be replayed. This is applied after the `allow_type_name_filter`. You may use a comma-separated list to specify more than one filter. |
| 0..1 |
| `<on_delete_wait_for_ack_timeout>` | Specifies a period for which the StreamWriter will wait for acknowledgment before its elimination. **Default:** 0 (no wait for acknowledgment) **Example:**

```xml
<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 |
| `<datawriter_qos>` | The DataWriter’s QoS to use when replaying this data. For information on configuring QoS with XML, see **Configuring QoS with XML**, in the *RTI Connext DDS Core Libraries User’s Manual*. |
| 0..1 |
### 4.3.14 Topic

The `<topic>` tag specifies an individual Topic to replay.

#### Table 4.18: Topic Tags in Replay Service’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;topic&gt;</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;topic_name&gt;</code></td>
<td>The name of the DDS topic to be replayed. If this tag is not present, the name attribute of the <code>&lt;topic&gt;</code> will be used. <em>Note:</em> we recommend using this tag to define the topic name. There may be characters that cause the XML validation to fail if they are part of the topic name attribute. Also, the ‘/’ character and '::' separator may cause Replay to fail when found in the topic name attribute.</td>
</tr>
<tr>
<td><code>&lt;registered_type_name&gt;</code></td>
<td>The name of the data type that will be replayed for this topic. <strong>Required.</strong></td>
</tr>
<tr>
<td><code>&lt;on_delete_wait_for_ack_timeout&gt;</code></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. <strong>Default:</strong> 0 (no wait for acknowledgment)</td>
</tr>
</tbody>
</table>

*Example:*

```xml
<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.

continues on next page
### Table 4.18 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;topic&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<transformation>`    | The transformation library to be applied to this Topic’s data when replaying. This is a user library that can modify the data after it is received from storage and before it is sent via Connext DDS. Transformations implement APIs identical to Routing Service’s transformations. For more on using transformations, see these sections in the RTI Routing Service User’s Manual:  
  - Data Transformation  
  - Tutorials  
  **Attributes:**  
  - `plugin_name`: The name of the plugin to load, qualified by the plugin library name.  
  **Example:**  
  ```xml  
  <dds>  
    <plugin_library name="ReplayTransformations">  
      <transformation_plugin name="ModifyTestID">  
        <create_function>ModifyTestID_create</create_function>  
        <dll>modify_test_id_library</dll>  
      </transformation_plugin>  
    </plugin_library>  
    <!-- ... -->  
    <replay_service>  
      <!-- ... -->  
      <topic name="TestTopic">  
        <transformation plugin_name="ReplayTransformations::ModifyTestID" />  
      </topic>  
    </replay_service>  
  </dds>  
  ``` | 0..1 |
| `<datawriter_qos>`  | The DataWriter QoS to use when replaying this data. For information on configuring QoS with XML, see Configuring QoS with XML, in the RTI Connext DDS Core Libraries User’s Manual. | 0..1 |

### 4.3.15 Plugins

All the pluggable components specific to Recording Service are configured within the `<plugin_library>` tag. Table 4.19 describes the available tags.

Plug-ins are categorized and configured based on the source language. Replay Service supports C/C++ plugins.
Table 4.19: Configuration tags for plugin libraries

<table>
<thead>
<tr>
<th>Tags within &lt;plugin_library&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;storage_plugin&gt;</td>
<td>Specifies a C/C++ Storage plugin. See Table 10.18 and Section 3.3.6.</td>
<td>0..*</td>
</tr>
<tr>
<td>&lt;transformation_plugin&gt;</td>
<td>Specifies a C/C++ Transformation plugin. See Table 10.18 and Section 3.3.10.</td>
<td>0..*</td>
</tr>
</tbody>
</table>

4.4 Remote Administration

This section provides documentation on Replay Service remote administration. A control client (such as RTI Admin Console) can use this interface to remotely control a Replay Service.

Note: Replay Service remote administration is based on the RTI Remote Administration Platform described in Section 10.3. Please refer to that manual for a detailed discussion on the workings of remote administration in Replay Service.

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

4.4.1 Enabling Remote Administration

By default, remote administration is disabled in Replay Service.

To enable remote administration you can use the <administration> XML tag (see Section 4.3) or the -remoteAdministrationDomainId command-line parameter (see Section 4.1). Both of these methods enable remote administration and set the domain ID for remote communication.

4.4.2 Available Service Resources

Table 4.20 lists the public resources specific to Replay Service. Each resource identifier is expressed as a hierarchical sequence of identifiers, including parent and target resources. (See Section 10.2.2 for details.)

In the table below, the elements (rs), and (st) 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:

<replay_service name="MyReplay">...</replay_service>

The resource identifier is:

/replay_service/MyReplay

In the table, the resource identifier is written as /replay_service/(rs), where (rs) is the service name. This nomenclature is used in the table to give you an idea of the structure of the resource identifiers. For actual (example) resource identifier names, see the example section that follows.
Table 4.20: Resources and Their Identifiers in Replay Service

<table>
<thead>
<tr>
<th>Resource</th>
<th>Resource Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replay Service</td>
<td>/replay_service/(rs)</td>
</tr>
</tbody>
</table>

**Example**

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

**Replay Service**

Entity with name “MyReplay”:

```xml
<replay_service name="MyReplay">...</replay_service>
```

Resource identifier:

```
/replay_services/MyReplay
```
4.4.3 Remote API Overview

Table 4.21: Remote Interface Overview

<table>
<thead>
<tr>
<th>Resource</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replay-Service</td>
<td>DELETE /replay_services/(rs)</td>
<td>Shuts down a running Replay Service</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/state</td>
<td>Sets a Replay Service state</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback/rate</td>
<td>Change the rate of the replay</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback/current_timestamp</td>
<td>Jump to a specific timestamp and continue the replay</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback/current_tag</td>
<td>Jump to a specific timestamp tag and continue the replay</td>
</tr>
<tr>
<td></td>
<td>GET /replay_services/(rs)/playback:state</td>
<td>Provide the current Replay Service state. It will allow us to detect when Replay Service hit a breakpoint.</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback:add_breakpoint</td>
<td>Add a new breakpoint</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback:remove_breakpoint</td>
<td>Remove an existing breakpoint</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback:goto_breakpoint</td>
<td>Jump to a specific breakpoint and stop the replay</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback:next_breakpoint</td>
<td>Jump to the next breakpoint and stop the replay</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback:continue</td>
<td>Continue the replay leaving the current breakpoint</td>
</tr>
<tr>
<td></td>
<td>UPDATE /replay_services/(rs)/playback:next_slice</td>
<td>Continue the replay for a slice of period</td>
</tr>
</tbody>
</table>

4.4.4 Replay Service

DELETE /replay_services/(rs)

Operation shutdown

Causes Replay Service to shutdown.

UPDATE /replay_services/(rs)/state

Operation set_state

See Set Resource State (Section 10.3.3).

Valid requested states:

- STARTED
- STOPPED
- PAUSED
• RUNNING

• Example

To pause a replay service with the name “MyReplay”:

<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/replay_services/MyReplay/state</td>
</tr>
<tr>
<td>octet_body</td>
<td>to_cdr_buffer(RTI::Service::EntityStateKind::PAUSED)</td>
</tr>
</tbody>
</table>

**UPDATE /replay_services/(rs)/playback/rate**

**Operation** set_rate

This operation will cause *Replay Service* to change its rate.

**Request body**

• The new rate of the replay.

**UPDATE /replay_services/(rs)/playback/current_timestamp**

**Operation** current_timestamp

This operation will cause *Replay Service* to change its current timestamp, modifying the replay location backward or forward in time.

It will affect the whole service, meaning that all StreamReaders will be affected by it.

• Example

To change the current timestamp in a replay service named “MyReplay”:  

4.4. Remote Administration  91
<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/replay_services/MyReplay/playback/current_timestamp</td>
</tr>
<tr>
<td>octet_body</td>
<td>RTI::RecordingService::TimestampHolder␣ → timestamp_holder;'</td>
</tr>
<tr>
<td></td>
<td>timestamp_holder_.→ nanos(1600635588280996383);</td>
</tr>
<tr>
<td></td>
<td>std::vector&lt;char&gt; timestamp_holder_.→ buffer;</td>
</tr>
<tr>
<td></td>
<td>dds::topic::topic_type_support</td>
</tr>
<tr>
<td></td>
<td>←<a href="">RTI::RecordingService::TimestampHolder</a></td>
</tr>
<tr>
<td></td>
<td>:to_cdr_buffer(</td>
</tr>
<tr>
<td></td>
<td>timestamp_holder_buffer,</td>
</tr>
<tr>
<td></td>
<td>timestamp_holder);</td>
</tr>
</tbody>
</table>

**UPDATE /replay_services/(rs)/playback/current_tag**

**Operation** current_tag

This operation will cause *Replay Service* to change its current timestamp, modifying the replay location backward or forward in time. The replay will jump to the timestamp of the selected timestamp tag.

It will affect the whole service, meaning that all StreamReaders will be affected by it.

**Request body**

- Tag name of a specific recorded timestamp tag.

**GET /replay_services/(rs)/playback:state**

**Operation** get_state

This operation will provide the current debug state of the *Replay Service* related to the debug mode. It will allow you to detect when *Replay Service* stop in a breakpoint

Valid provided states:

- BREAKPOINT
- WORKING

**Request body**

- Empty.

**Reply body**


**UPDATE /replay_services/(rs)/playback:add_breakpoint**

**Operation** add_breakpoint
This operation will create a new breakpoint. *Replay Service* will only accept as proper breakpoint those which timestamp is not out of the recorded timestamp period.

Also, we can use a recorded timestamp tag to create a breakpoint with it.

- **Examples**

  To create a new breakpoint in a replay service named “MyReplay”:

<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/replay_services/MyReplay/playback:add_breakpoint</td>
</tr>
<tr>
<td>octet_body</td>
<td>RTI::RecordingService::BreakpointParams␣ → breakpoint_arguments;</td>
</tr>
<tr>
<td></td>
<td>→ dds::core::optional&lt; std::string &gt;␣ → optional_label_value(&quot;breakpoint_1&quot;);</td>
</tr>
<tr>
<td></td>
<td>→ breakpoint_arguments.label(optional_label_value);</td>
</tr>
<tr>
<td></td>
<td>→ breakpoint_arguments.value().timestamp_.nanos(1600635588280996383);</td>
</tr>
<tr>
<td></td>
<td>→ <a href="">RTI::RecordingService::BreakpointParams</a> →</td>
</tr>
<tr>
<td></td>
<td>→ → to_cdr_buffer(</td>
</tr>
<tr>
<td></td>
<td>→ reinterpret_cast</td>
</tr>
<tr>
<td></td>
<td>→ &lt;std::vector&lt;char&gt; &amp;&gt; (request.octet_body()),</td>
</tr>
<tr>
<td></td>
<td>→ breakpoint_arguments);</td>
</tr>
</tbody>
</table>

  To create a new breakpoint using a timestamp tag:
<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/replay_services/MyReplay/playback:add_breakpoint</td>
</tr>
</tbody>
</table>
| octet_body           | RTI::RecordingService::BreakpointParams
                      → breakpoint_arguments;
                      ◄ dds::core::optional<std::string> ◄
                      → optional_label_value("breakpoint_tag ◄
                      → ");
                      breakpoint_arguments.label(optional_ ◄
                      → label_value);
                      breakpoint_arguments.value().tag_name(
                      → "example/tag1");
                      dds::topic::topic_type_support
                      ◄<RTI::RecordingService::BreakpointParams>
                      ◄
                      →
                      ◄::to_cdr_buffer(
                      reinterpret_cast
                      ◄<std::vector<char> &> (request.octet_ ◄
                      → body()),
                      breakpoint_arguments); |

**UPDATE /replay_services/(rs)/playback:remove_breakpoint**

**Operation** remove_breakpoint

This operation will remove an existing breakpoint for the replay. A breakpoint can be removed by label or by timestamp.

In order to clean the complete list of breakpoints (except the default one) you can use the special character “*” as label value.

If *Replay Service* is hitting the breakpoint to be removed, It won’t resume the replay.

- **Examples**
  
  To remove the breakpoint with label “MyBreakpoint” in a replay service named “MyReplay”:
### UPDATE /replay_services/(rs)/playback:goto_breakpoint

**Operation goto_breakpoint**

This operation will cause *Replay Service* to jump to a specific breakpoint. You can specify the jumping breakpoint by label or by timestamp.

It will affect the whole service, meaning that all StreamReaders will be affected by it.

-Example:

To jump to breakpoint with timestamp “1600635588280996383” in a replay service named “MyReplay”: 

```cpp
RTI::RecordingService::BreakpointParams breakpoint_arguments;
::dds::core::optional<std::string> optional_label_value("MyBreakpoint");
breakpoint_arguments.label(optional_label_value);
breakpoint_arguments.value().timestamp.nanos(0);
dds::topic::topic_type_support<RTI::RecordingService::BreakpointParams>
::to_cdr_buffer(
    reinterpret_cast<std::vector<char> &> (request.octet_body()),
    breakpoint_arguments);
```
<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/replay_services/MyReplay/playback:goto_breakpoint</td>
</tr>
<tr>
<td>octet_body</td>
<td>RTI::RecordingService::BreakpointParams::<a href="">RTI::RecordingService::BreakpointParams</a>::to_cdr_buffer</td>
</tr>
<tr>
<td></td>
<td>reinterpret_cast&lt;std::vector&lt;char&gt; &amp;&gt;(request.octet_body()), breakpoint_arguments;</td>
</tr>
</tbody>
</table>

**UPDATE /replay_services/(rs)/playback:next_breakpoint**

**Operation** `next_breakpoint`

This operation will cause Replay Service to jump to the next breakpoint on the list. If Replay Service was on the last breakpoint of the list, it will throw an exception.

It will affect the whole service, meaning that all StreamReaders will be affected by it.

**Request body**

- Empty.

**UPDATE /replay_services/(rs)/playback:continue**

**Operation** `continue`

This operation will cause Replay Service to continue the replay after it hit a breakpoint. If you do not add any CDR representation to the request, it will continue until it hits a new breakpoint or the replay end. Otherwise, this operation will continue the replay for the period of time specified by you. Replay Service has to be on BREAKPOINT state in order to make this action.

It will affect the whole service, meaning that all StreamReaders will be affected by it.

-Examples:

To continue the replay for 1 second in a replay service named “MyReplay”:
To continue the replay for 2 slices periods in a replay service named “MyReplay”:

```
<table>
<thead>
<tr>
<th>Request Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>command_action</td>
<td>UPDATE</td>
</tr>
<tr>
<td>resource_identifier</td>
<td>/replay_services/MyReplay/playback:continue</td>
</tr>
</tbody>
</table>
| octet_body        | RTI::RecordingService::ContinueParams␣
|                   |   ⏯ continue_arguments.value().slices(2);                                |
|                   |   ⏯ dds::topic::topic_type_support␣
|                   |   ⏯ <RTI::RecordingService::ContinueParams>␣
|                   |   ⏯ ::to_cdr_buffer(                                                |
|                   |     reinterpret_cast<std::vector<char>␣
|                   |       &) (request.octet_body()),                                    |
|                   |     continue_arguments);                                           |
```

**UPDATE /replay_services/(rs)/playback:next_slice**

**Operation next_slice**

This operation will cause Replay Service to resume the replay for the duration of a slice period. Replay Service has to be on BREAKPOINT state in order to make this action.

It will affect the whole service, meaning that all StreamReaders will be affected by it.

**Request body**

- Empty.

4.4. Remote Administration 97
4.5 Monitoring

This section provides documentation on Recording Service remote monitoring.

Note: Recording Service monitoring is based on the Monitoring Distribution Platform described in Section 10.4. We recommend that you read Section 10.4 before using Recording Service monitoring.

4.5.1 Overview

Enabling Service Monitoring

By default, monitoring is disabled in Recording Service. To enable monitoring you can use the <monitoring> tag (see Section 3.3.3) or the -remoteMonitoringDomainId command-line parameter, which enables remote monitoring and sets the domain ID for data publication (see Section 3.1.3).

Monitoring Types

The available Keyed Resource classes and their types that can be present in the distribution monitoring topics are listed in Table 4.22. The complete type relationship is shown in Figure 4.5.

Table 4.22: Recording Service Keyed Resources

<table>
<thead>
<tr>
<th>Keyed Class</th>
<th>Resource</th>
<th>Config</th>
<th>Event</th>
<th>Periodic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>Service</td>
<td>ServiceConfig</td>
<td>ServiceEvent</td>
<td>ServicePeriodic</td>
</tr>
<tr>
<td>Service</td>
<td>Session</td>
<td>SessionConfig</td>
<td>SessionEvent</td>
<td>SessionPeriodic</td>
</tr>
<tr>
<td>TopicGroup</td>
<td>Topic</td>
<td>TopicConfig</td>
<td>TopicEvent</td>
<td>TopicPeriodic</td>
</tr>
<tr>
<td>TopicGroup</td>
<td>Group</td>
<td>TopicGroupConfig</td>
<td>TopicGroupEvent</td>
<td>TopicGroupPeriodic</td>
</tr>
<tr>
<td>TopicGroup</td>
<td>Topic</td>
<td>TopicConfig</td>
<td>TopicEvent</td>
<td>TopicPeriodic</td>
</tr>
</tbody>
</table>

All the type definitions for Recording Service monitoring information are in [NDDSHOME]/resource/idl/ServiceCommon.idl and [NDDSHOME]/resource/idl/RecordingServiceMonitoring.idl.

Recording Service creates a DataWriter for each distribution Topic. All DataWriters are created from a single Publisher, which is created from a dedicated DomainParticipant. See Section 3.3.3 for details on configuring the QoS for these entities.
4.5.2 Monitoring Metrics Reference

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

Service

Listing 4.1: Recording Service Types

```cpp
@mutable @nested
struct SqliteDatabaseConfig {
    Service::FilePath db_directory;
    @optional Service::FilePath execution_directory_expression;
    @optional Service::FilePath user_data_file_expression;
};
@mutable @nested
struct SqliteDatabaseEvent {
    @optional Service::FilePath current_db_directory;
    @optional Service::FilePath current_file;
    @optional int32 rollover_count;
};
@mutable @nested
struct SqliteDatabasePeriodic {
    @optional Service::FilePath current_file;
};
```
@optional uint64 current_file_size;
// These fields are no longer supported and carry no information.
// Kept only to support older version.
@deprecated int32 current_timestamp_sec;
@deprecated uint32 current_timestamp_nanosec;
};

@mutable @nested
struct ParticipantInfo {
  Service::BoundedString name;
};

@mutable @nested
struct ServiceConfig : Service::Monitoring::EntityConfig {
  Service::BoundedString application_name;
  Service::Monitoring::ResourceGuid application_guid;
  @optional Service::Monitoring::HostConfig host;
  @optional Service::Monitoring::ProcessConfig process;
  @optional SqliteDatabaseConfig builtin_sqlite;
  @optional sequence<ParticipantInfo> participants;
};

@mutable @nested
struct ServiceEvent : Service::Monitoring::EntityEvent {
  //to avoid unused variable warnings
  @optional SqliteDatabaseEvent builtin_sqlite;
};

@mutable @nested
struct ServicePeriodic {
  @optional Service::Monitoring::HostPeriodic host;
  @optional Service::Monitoring::ProcessPeriodic process;
  int64 current_timestamp_nanos;
  @optional SqliteDatabasePeriodic builtin_sqlite;
};
### Table 4.23: ServiceConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityConfig</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>application_name</td>
<td>Name of the <em>Recording Service</em> instance. The application name is provided through:</td>
</tr>
<tr>
<td></td>
<td>• appName command-line option when run as executable.</td>
</tr>
<tr>
<td></td>
<td>• ServiceProperty::application_name field when run as a library.</td>
</tr>
<tr>
<td>application_guid</td>
<td>GUID of the <em>Recording Service</em> instance. Unique across all service instances.</td>
</tr>
<tr>
<td>host</td>
<td>See Table 10.10.</td>
</tr>
<tr>
<td>process</td>
<td>See Table 10.12.</td>
</tr>
<tr>
<td>builtin_sqlite</td>
<td>See Table 4.24</td>
</tr>
<tr>
<td>participants</td>
<td>Sequence of ParticipantInfo objects, one for each <em>DomainParticipant</em> inside the <em>Recording Service</em>. See Table 4.25.</td>
</tr>
</tbody>
</table>

### Table 4.24: SqliteDatabaseConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_directory</td>
<td>Path to the base directory where the database files live.</td>
</tr>
<tr>
<td>execution_directory_expression</td>
<td>This value is not set when running <em>Replay Service</em>. See Section 3.3.6</td>
</tr>
<tr>
<td>user_data_file_expression</td>
<td>This value is not set when running <em>Replay Service</em>.</td>
</tr>
</tbody>
</table>

### Table 4.25: ParticipantInfo

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the <em>DomainParticipant</em> instance, as specified in the name attribute of the corresponding configuration tag.</td>
</tr>
</tbody>
</table>

### Table 4.26: ServiceEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityEvent</td>
<td>See Table 10.15.</td>
</tr>
<tr>
<td>builtin_sqlite</td>
<td>See Table 4.27</td>
</tr>
</tbody>
</table>

### Table 4.27: SqliteDatabaseEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_db_directory</td>
<td>Path to the current directory where files are being replayed from.</td>
</tr>
<tr>
<td>current_file</td>
<td>This value is not set when running <em>Replay Service</em>.</td>
</tr>
<tr>
<td>rollover_count</td>
<td>This value is not set when running <em>Replay Service</em>.</td>
</tr>
</tbody>
</table>
Table 4.28: ServicePeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>See Table 10.11.</td>
</tr>
<tr>
<td>process</td>
<td>See Table 10.13.</td>
</tr>
<tr>
<td>current_timestamp_nanos</td>
<td>Timestamp in nanoseconds at which data is being replayed, relative to recorded time.</td>
</tr>
<tr>
<td>builtin_sqlite</td>
<td>See Table 4.29</td>
</tr>
</tbody>
</table>

Table 4.29: SqliteDatabasePeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>current_file_size</td>
<td>This value is not set when running Replay Service.</td>
</tr>
</tbody>
</table>

Session

Listing 4.2: Session Types

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

@mutable @nested
struct SessionEvent : Service::Monitoring::EntityEvent {
    //to avoid unused variable warnings
    int32 _dummy;
};

@mutable @nested
struct SessionPeriodic {
    @optional Service::Monitoring::NetworkPerformance network_performance;
    @optional @optional Service::Monitoring::ThreadPoolPeriodic thread_pool;
};
```

Table 4.30: SessionConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityConfig</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>default_participant_name</td>
<td>The name of the default participant configuration.</td>
</tr>
</tbody>
</table>
### Table 4.31: SessionEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityEvent</td>
<td>See Table 10.15.</td>
</tr>
</tbody>
</table>

### Table 4.32: SessionPeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network_performance</td>
<td>Provides network performance metric as an aggregation of the same metric across the contained Topics and TopicGroups. See Section 10.4.4.</td>
</tr>
</tbody>
</table>

### TopicGroup

#### Listing 4.3: TopicGroup Types

```cpp
@mutable @nested
struct TopicGroupConfig : Service::Monitoring::EntityConfig {
    Service::BoundedString participant_name;
};

@mutable @nested
struct TopicGroupEvent : Service::Monitoring::EntityEvent {
    // to avoid unused variable warnings
    int32 _dummy;
};

@mutable @nested
struct TopicGroupPeriodic {
    @optional Service::Monitoring::NetworkPerformance network_
    
    //performance;
    int64 topic_count;
};
```

### Table 4.33: TopicGroupConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityConfig</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>participant_name</td>
<td>Name of the DomainParticipant from which the Topic is created.</td>
</tr>
</tbody>
</table>

### Table 4.34: TopicGroupEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields from EntityEvent</td>
<td>See Table 10.15.</td>
</tr>
</tbody>
</table>
Table 4.35: TopicGroupPeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network_performance</td>
<td>Provides network performance metric as an aggregation of the same metric across the contained Topics. See Section 10.4.4.</td>
</tr>
<tr>
<td>topic_count</td>
<td>Current number of Topics created from this TopicGroup.</td>
</tr>
</tbody>
</table>

**Topic**

Listing 4.4: Topic Types

```cpp
@mutable @nested
struct TopicConfig : Service::Monitoring::EntityConfig {
    Service::BoundedString topic_name;
    Service::BoundedString registered_type_name;
    Service::BoundedString participant_name;
    Service::Monitoring::ResourceGuid topic_group;
};

@mutable @nested
struct TopicEvent : Service::Monitoring::EntityEvent {
    // to avoid unused variable warnings
    int32 _dummy;
};

@mutable @nested
struct TopicPeriodic {
    @optional Service::Monitoring::NetworkPerformance network_performance;
    @optional Service::Monitoring::CountStatus matched_status;
};
```

Table 4.36: TopicConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields</td>
<td>See Table 10.14.</td>
</tr>
<tr>
<td>topic_name</td>
<td>Topic name as specified in the configuration.</td>
</tr>
<tr>
<td>registered_type_name</td>
<td>Topic registered type name as specified in the configuration.</td>
</tr>
<tr>
<td>participant_name</td>
<td>Name of the DomainParticipant from which the Topic is created.</td>
</tr>
<tr>
<td>topic_group</td>
<td>GUID of the TopicGroup from which this Topic was created. This field is set to zero for standalone Topics.</td>
</tr>
</tbody>
</table>

Table 4.37: TopicEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited fields</td>
<td>See Table 10.15.</td>
</tr>
<tr>
<td>Field Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>network_performance</td>
<td>Provides network performance metric as an aggregation of the same metric across the contained Topics. See Section 10.4.4.</td>
</tr>
<tr>
<td>matched_status</td>
<td>Provides information about the matched endpoints associated with this Topic.</td>
</tr>
</tbody>
</table>

### 4.6 Tutorials

#### 4.6.1 Example: Getting Started with Replay and Shapes Demo

Start by recording Square data, as described in Section 3.6.1.

**Start Shapes Demo and Subscribe to Squares**

If you have any Shapes Demo windows publishing data, delete them.

Start Shapes Demo from Launcher and create a Square subscriber as described in Section 3.6.1.

**Start Replay Service**

Start Replay Service:

```
<NDDSHOME>/bin/rtireplayservice -cfgName UserReplayServiceJson -verbosity 3
```

You should see Square data replayed in Shapes Demo.

#### 4.6.2 Example: Replaying Data at a Different Rate

To replay data at a faster or slower rate than it was recorded, you can edit the configuration file to specify a playback rate.

Start by recording Square data as described in Section 3.6.1.

**Edit the Replay Configuration**

In your `<RTI_WORKSPACE>/user_config/recording_service` directory, edit the `USER_REPLAY_SERVICE.xml` file. Add a playback rate, as seen in the following XML:

```xml
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="../resource/schema/rti_replay_service.xsd">

(continues on next page)```
<replay_service name="UserReplayServiceJson">
  <!-- This will look for files in the directory 'json_recording' in the current working directory. This is integrated with the Recording Service configuration 'UserRecorderServiceJson' in the file USER_RECORDING_SERVICE.xml -->
  <storage>
    <sqlite>
      <storage_format>JSON_SQLITE</storage_format>
      <database_dir>json_recording</database_dir>
    </sqlite>
  </storage>
  <!-- Optionally select the begin and end times for the data to be replayed -->
  <!--data_selection>
    <time_range>
      <begin_time>
        <sec>0</sec>
        <nanosec>0</nanosec>
      </begin_time>
      </time_range>
    </data_selection-->
  <!-- Specify playback behavior, including what local time to start -->
  <playback>
    <rate>2</rate>
  </playback>
  <domain_participant name="DefaultParticipant">
    <domain_id>0</domain_id>
  </domain_participant>
  <session name="DefaultSession">
    <default_participant_ref="DefaultParticipant">
      <!-- Topics to replay in this session -->
      <topic_group name="DefaultTopicGroup">
        <!-- Topics to replay -->
        <allow_topic_name_filter>*</allow_topic_name_filter>
        <deny_topic_name_filter>rti/*</deny_topic_name_filter>
      </topic_group>
    </session>
  </session>
</replay_service>
Start Shapes Demo

Use Launcher to start Shapes Demo, then create a Square subscriber.

Start Replay Service

Start Replay Service:

```
<NDDSHOME>/bin/rtireplayservice -cfgName UserReplayService -verbosity 3
```

You should see Square data replayed in Shapes Demo twice as fast as it was originally recorded.

4.6.3 Example: Plugging in Custom Storage

If you created a storage plugin to record data into your own custom storage, you can also create a plugin to replay from that same storage.

There are full examples written in C and C++ about plugging in custom storage in Recording Service, in the RTI Community Recording Service examples: C storage plugin and RTI Community Recording Service examples: C++ storage plugin.

Custom Storage API Overview

To retrieve data for replay, you must implement the following APIs:

- **A create storage reader API:**
  - This is used to create a StorageReader object.
  - This API is a C function. In C++, you can use macros to declare and define the C function for your class. For example:
    
    ```
    * RTI_RECORDING_STORAGE_READER_CREATE_DECL(FileStorageReader)
    * RTI_RECORDING_STORAGE_READER_CREATE_DEF(FileStorageReader)
    ```
  - The StorageReader is used to create and delete StorageStreamInfoReaders and StorageStreamReaders.

- **StorageReader:**
  - A create stream info reader API, where you create a stream reader that provides information about what streams (topics) are in your storage.
  - A delete stream info reader API, where you delete a stream info reader
  - A create stream reader API, where you create a stream reader
  - A delete stream reader API, where you delete a stream reader
• **StorageStreamInfoReader:**
  - Read: An API to retrieve discovery data from the storage. It uses a selector object to determine the kind of samples (not read or any kind) to be returned, as well as the time range. The samples should be returned in increasing time-stamp order.
  - Return loan: An API that notifies the plugin that it can release resources associated with the data passed to the read API
  - Get service start time: An API to query the recorded time from which to start replaying
  - Get service stop time: An API to query the recorded time at which to stop replaying
  - Finished: An API to tell *Replay Service* and *Converter* that there are no more stream infos to read.
  - Reset: An API called by the *Replay Service* to tell the plug-in to reset its state because it is looping. After this method is called, the stream reader should be ready to start reading data from the beginning of the stream, again.

• **StorageStreamReader**
  - Read: An API to retrieve data from storage. It uses a selector object to determine the kind of samples (not read or any kind) to be returned, as well as the time range. The samples should be returned in increasing timestamp order.
  - Return loan: An API that notifies the plugin that it can release resources associated with the data passed to the take API
  - Finished: An API that notifies the plugin that there is no more data in this data stream

The APIs provide a mechanism to have strongly typed StorageStreamReader classes. A built-in one is provided, based on `dds::core::xtypes::DynamicData`.

More detailed API documentation is in:

• [Recording Service C API documentation](#)

• [Recording Service C++ API documentation](#)

### 4.6.4 Using Timestamp Tags with Replay Service

If your recording was originally made with the built-in SQLite storage plugin, and you used the `tag_timestamp` remote command to tag certain events, then your recording contains timestamp tags: symbolic timestamp names you can use in place of timestamps expressed in units of time. For more information on timestamp tags, see Section 3.6.6.

You can list the timestamp tags that are in your recorded database by using the `rtirecordingservice_list_tags` script. Use the `-d` argument to point to the directory that contains your recorded database, as follows:

```
<NDDSHOME>/bin/rtirecordingservice_list_tags -d /database/directory/
```

This command will analyze the recording in `/database/directory/` and list the details of any timestamp tags in the recording, including the tag names, descriptions, and associated timestamps.
You can use the tag_name of the timestamp tags you find in a recording when you are creating an XML configuration file for Replay Service by using the <data_selection> tag.

For example, if after running rtirecordingservice_list_tags, you see output such as:

<table>
<thead>
<tr>
<th>tag_name</th>
<th>timestamp_ms</th>
<th>tag_description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/my_example/my_events/tag1</td>
<td>1546484663309</td>
<td>first tag description</td>
</tr>
<tr>
<td>/my_example/my_events/tag2</td>
<td>1546484703360</td>
<td>a second tag description</td>
</tr>
</tbody>
</table>

Then you can have a <data_selection> tag in your XML for Replay Service, after the <storage> tag, that looks like the following:

```
<data_selection>
  <time_range>
    <begin_tag>/my_example/my_events/tag1</begin_tag>
    <end_tag>/my_example/my_events/tag2</end_tag>
  </time_range>
</data_selection>
```

Replay Service will replay data between those tags. Note that when expressing a <time_range> tag, you can mix and match timestamps and timestamp tags. For example, you can use a <begin_tag> (by referring to a tag_name) to express the time when replay should begin, and an <end_tag> with an end time timestamp (expressed in time units) to express when replay should end. If you do not provide one of the bounds, then the start of recording is the default begin bound and the end of recording is the default end bound.

### 4.6.5 Jump in time in Replay Service

This section shows a possible scenario where the jump in time operation can be useful. This scenario will be based on the following recorded database:

```sql
sqlite3 json_recording/rti_recorder_default_json.db
sqlite> select ROWID, SampleInfo_reception_timestamp, rti_json_sample from ...
```

```
1|1600635598310952678|{"color":"ORANGE","x":120,"y":195,"shapesize":30,...
2|1600635598460562312|{"color":"ORANGE","x":121,"y":197,"shapesize":30,...
3|1600635599612452987|{"color":"ORANGE","x":122,"y":199,"shapesize":30,...
4|1600635601310952678|{"color":"ORANGE","x":123,"y":201,"shapesize":30,...
      ...
1000|1600637901310952678|{"color":"ORANGE","x":129,"y":199,"shapesize":30,...
```

In order to reproduce this scenario, see Section 3.6.1.

Suppose you want to replay for some seconds, then you want to replay again samples 4 to 10, because they contain important events and continue the replay until the end. To do this, you need to use the jump in time operation to change the replay position, like this.
CommandReply reply;
CommandRequest request;
request.action(CommandActionKind::UPDATE_ACTION);
request.application_name(service_property.application_name());
request.resource_identifier("/playback/current_timestamp");
RTI::RecordingService::TimestampHolder timestamp_arguments;
timestamp_arguments.timestamp_nanos(1600635601310952678);
.dds::topic::topic_type_support<RTI::RecordingService::TimestampHolder>::to_cdr_buffer(
    reinterpret_cast<std::vector<char> &>(request.octet_body()),
    timestamp_arguments);
replay_service.execute_command(reply, request);

Once Replay Service has jumped to sample 4, it will replay those important samples again.

### 4.6.6 Using Debug mode in Replay Service

In this section you can see a possible scenario where the debug mode can be useful. This scenario will be based on the following recorded database:

```sql
sqlite> select ROWID, SampleInfo_reception_timestamp, rti_json_sample from "Square@0";
1|1600635598310952678|{"color":"ORANGE","x":120,"y":195,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
2|1600635598460562312|{"color":"ORANGE","x":121,"y":197,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
3|1600635599612452987|{"color":"ORANGE","x":122,"y":199,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
4|1600635601310952678|{"color":"ORANGE","x":123,"y":201,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
....
100|1600637398330952678|{"color":"ORANGE","x":126,"y":199,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
101|1600637498460562312|{"color":"ORANGE","x":127,"y":199,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
102|1600637799612452987|{"color":"ORANGE","x":128,"y":199,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
103|1600637901310952678|{"color":"ORANGE","x":129,"y":199,"shapesize":30, "fillKind":"SOLID_FILL","angle":0}
```

In order to reproduce this scenario, see Section 3.6.1.

Suppose you want to replay samples 1-4, then you want to skip samples 5-99 and continue the replay at sample 100. To do this, you need to add some breakpoints to stop Replay Service before it replays specific samples. So you need to add two breakpoints: one after the 4th sample and another before the 100th sample.

To add those breakpoints you can use the `<initial_breakpoint>` tags. For example:
Also, you can add those breakpoints by code using the remote administration system. See Section 4.4.3 for more details about the different debug mode operations.

Once the replay starts, it will hit the default breakpoint. This breakpoint is always added by Replay Service at the start timestamp of the replay.

To start the replay, we should execute the operation “continue” using remote administration. Replay Service will publish the first 4 samples, then it will stop when it hits breakpoint_afterSample4.

Once it hits this breakpoint, we don’t want to replay the 5th sample. Instead, we want to jump to the breakpoint “breakpoint_beforeSample100”, which is just before sample 100. To do this, we have to execute the operation “goto_breakpoint” like this:

```cpp
CommandReply reply;
CommandRequest request;
request.action(CommandActionKind::UPDATE_ACTION);
request.application_name(service_property.application_name());
request.resource_identifier("/playback:goto_breakpoint");
RTI::RecordingService::BreakpointParams breakpoint_arguments;
breakpoint_arguments.value().labels("breakpoint_beforeSample100");
dds::topic::topic_type_support<RTI::RecordingService::BreakpointParams>::to_cdr_buffer(
  reinterpret_cast<std::vector<char> &> (request.octet_body()),
  breakpoint_arguments);
replay_service.execute_command(reply, request);
```

Once Replay Service has jumped to “breakpoint_beforeSample100”, we need to execute the “continue” operation to resume replaying from sample 100 until the end.

### 4.6.7 Instance History replay

In this section, you will see instance history replay in action. For this scenario, we will need to record a database with a keyed type with only one instance. To do that, we will create an example from the following IDL:

```cpp
struct Hello {
  long id; // @key
  long sample_number;
};
```

In order to generate the example, you need to run rtiddsgen like this:
After generating the code, we need to make a small change in Hello_publisher.cxx, so that it only creates an instance. To do that, set data.id to 0, instead of to sample_written.

```cpp
// Instance history example: Set data.id to 0.
data.id(0);
for (unsigned int samples_written = 0;
    !application::shutdown_requested && samples_written < sample_count;
    samples_written++) {
    // Modify the data to be written here
    // Instance history example: Remove this data.id set line.
    // data.id(static_cast<int32_t>(samples_written));
    ....
```

To record the scenario, we need to enable Instance indexing. Please see Section 3.3.6 for more information.

Once we have the recorded database, we can move to the replay side. We will base the scenario on the following recorded database:

```sql
sqlite> select ROWID, SampleInfo_reception_timestamp from "Example Hello@0";
1|1600635598310952678
2|160063598460562312
3|160063599612452987
4|1600635601310952678
....
10|1600636398330952678
```

We need to make some changes in the replay configuration to start at sample 4 and to enable instance history replay. To do that, add the following tags:

```xml
<data_selection>
  <time_range>
    <begin_time>
      <sec>1600635600</sec>
      <nanosec>310952678</nanosec>
    </begin_time>
    </time_range>
  </time_range>
</data_selection>

<playback>
  <instance_history_replay>
    <enabled>true</enabled>
  </instance_history_replay>
</playback>
```

Also, the DataReader and the Replay Service DataWriter need to have their Durability QoS set to TRANSIENT_LOCAL in order to receive the historical sample. Please see Configuring QoS with XML, in the RTI Connext DDS Core Libraries User’s Manual for more information.

Once everything was set, we can run the replay scenario. The expected output is:
Notice that you also received sample 3, which was before the start timestamp. This sample was received as part of the state of the world. If we have more than one instance, we will receive one sample per instance (if they have a valid state on that start timestamp).

4.7 Troubleshooting

4.7.1 No Input File

If Replay Service is started but there is no file for it to read from, it will print error messages like the ones below and then exit. It is looking for a database located in a cdr_recording directory inside your current working directory. You can fix this by creating a configuration file that specifies the correct location of your database to replay, or by changing directories to a location that contains your cdr_recording directory that contains a recording.

```
[/replay_services/default|START] create_connection: caught exception from:
  set_properties: !No valid metadata file found in directory: cdr_recording
[/replay_services/default|START] ROUTERConnection_enable:(adapter=StorageAdapterPlugin,
  → retcode=0: set_properties: !No valid metadata file found in directory: cdr_recording)
[/replay_services/default|START] ROUTERDomainRoute_start:!enable Connection
[/replay_services/default|START] ROUTERService_startDomainRoute:!start domain route
[/replay_services/default|START] ROUTERService_createDomainRoute:!start domain route
[/replay_services/default|START] RTI_RoutingService_start:!start routing service
main:!!start RoutingService error
```

4.7.2 Table Not Found Errors

When recording a database with Recording Service, there may be topics that have no associated table, because they were discovered but were filtered out by using the <allow_topic_name_filter> or <deny_topic_name_filter> tags in Topic Group or by defining Topics (that target specific topic names). While the topic will be present in the DCPSPublication table in the discovery file, it won’t have a corresponding table in the user-data files.

If the same topics are not filtered in Replay Service (by using the same <allow_topic_name_filter> or <deny_topic_name_filter> tags, or Topics), then when Replay Service starts it will discover the...
topics without a table because they are available in the discovery information. When Replay Service attempts to create a stream reader for these topic(s), a failure message will be printed:

```
ROUTERConnection_createStreamReaderAdapter: (adapter=StorageAdapterPlugin, ...
  retcode=0: Function returned NULL)
ROUTERStreamReader_enable:!create stream reader adapter
ROUTERTopicRoute_enableInput:!enable stream reader
ROUTERTopicRoute_processEvent:!enable route input
ROUTERTopicRoute_onConditionTriggered:!process event
create_stream_reader_fwd:SQLiteStorageStreamReader:!Table not found in...
  database files: TopicNotRecorded@0
```

These messages are harmless, they are just informing you that a table could not be found for the topic (in the example above, TopicNotRecorded).

To get rid of the messages, use the same Topic Group filter expressions or Topics used in Recording Service.

### 4.7.3 Receiving the data twice

When a Subscriber uses a different typename than the current typename recorded, Replay Service will create two sessions for the same topic.

Since TypeObject is being used, discovery completes correctly, even though there are two different type names.

The behavior in this scenario is that the data for that topic will be received twice. To prevent this, make sure the Subscriber uses the same typename that was used in the recording.

---

**4.7. Troubleshooting**
Chapter 5

Converter

5.1 Usage

This section explains how to run Converter from a command line. In particular, it describes:

- How to Start Converter (Section 5.1.1).
- Converter command-line parameters (Section 5.1.2).

5.1.1 Starting Converter

Converter runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin`. (See Section 1.3 for the path to NDDSHOME.)

```plaintext
rticonverter [options]
```

To start Converter with a default configuration, enter:

```
$NDDSHOME/bin/rticonverter
```

Converter is pre-loaded with a builtin configuration that has default settings. See Section 5.2.21.

Note: To run Converter 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 `connexddsArchitecture` in the file `rticonfig]=[sh/bat]` to the name of the target architecture. (The file is `resource/scripts/rticonfig=[sh/bat]` in Linux or macOS systems, `resource/scripts/rticonfig.bat` on Windows systems.) If the `CONNEXTDDS_ARCH` environment variable is set, the architecture in this file will be ignored.
5.1.2 Converter Command-Line Parameters

The following table describes all the command-line parameters available in Converter. To list the available parameters, run rticonverter -help.

All command-line parameters are optional; if specified, they override the values of any corresponding settings in the loaded XML configuration. See Section 5.2.4 for the XML elements that can be overridden with command-line parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-cfgFile &lt;string&gt;</td>
<td>Semicolon-separated list of configuration file paths. <strong>Default</strong>: Unspecified</td>
</tr>
<tr>
<td>-cfgName</td>
<td>Configuration name. This name is used to find a matching &lt;converter&gt; tag in the configuration file.</td>
</tr>
<tr>
<td>-D&lt;name&gt;=&lt;value&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>-help</td>
<td>Shows this help.</td>
</tr>
<tr>
<td>-verbosity &lt;service_level&gt;[:&lt;dds_level&gt;]</td>
<td>Controls what type of messages are logged. &lt;service_level&gt; is the verbosity level for the service logs and &lt;dds_level&gt; is the verbosity level for the DDS logs. Both can take any of the following values: SILENT, ERROR, WARN, LOCAL, REMOTE, ALL. <strong>Default</strong>: ERROR:ERROR</td>
</tr>
<tr>
<td>-version</td>
<td>Prints the program version and exits.</td>
</tr>
</tbody>
</table>

5.1.3 Working With Large Data

The built-in SQLite plugin implementation available in Converter is prepared to handle any type size and storage format. However, when working with large data types and massive files (or filesets), Converter may take a while to prepare SQL statements to work with the data, resulting in a delay when starting the conversion process.

Because of this delay, it is recommended that you index the user data tables for those large topics before running Converter on them. Indexing can massively improve Converter’s startup time for those topics. You can create the indexes offline, after Recording Service has finished recording all the data. Index the tables on the SampleInfo_reception_timestamp field. For example, imagine a table, VeryLargeTopic@0, has been created by Recording Service; you can use the following index creation statement:

```
CREATE INDEX IF NOT EXISTS [VeryLargeTopic@0_idx_rt]
ON [VeryLargeTopic@0] (SampleInfo_reception_timestamp)
```
5.2 Converter Configuration

When you start Converter, you can specify a configuration file in XML format. In this file, you can specify the properties that control the behavior of the service. This section describes how to write a configuration file.

5.2.1 How to Load the XML Configuration

Converter loads its XML configuration from multiple locations. Here are the various sources of configuration files, listed in load order:

1. [working directory]/USER_CONVERTER.xml This file is loaded automatically if it exists.
2. [NDDSHOME]/resource/xml/RTI_CONVERTER.xml This file is loaded automatically if it exists.
3. One or more files (semicolon-separated) specified using the command-line parameter -cfgFile.

Note: [working directory] indicates the path to the current working directory from which you run Converter.

[NDDSHOME] indicates the path to your Connext DDS installation. See Section 1.3.

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

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

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="/resource/schema/rti_converter.xsd">
   <!-- Available types -->
   <types />

   <!-- A definition of a Converter instance to run -->
   <converter name="defaultToJson">
      <!-- Input storage settings -->
      <input_storage>
         <sqlite>
            <storage_format>XCDR_AUTO</storage_format>
            <database_dir>cdr_recording</database_dir>
         </sqlite>
      </input_storage>

      <!-- Output storage settings -->
      <output_storage>
         <sqlite>
            <storage_format>JSON_SQLITE</storage_format>
            <fileset>
               <!-- Further configuration for output storage -->
            </fileset>
         </sqlite>
      </output_storage>
   </converter>
</dds>
```

(continues on next page)
5.2.2 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, Converter relies on an XSD file that describes the format of the XML content. We recommend including a reference to this document in the XML file that contains the service's configuration; this document provides helpful features in code editors such as Visual Studio®, Eclipse®, and NetBeans®, including validation and auto-completion while you are editing the XML file.

The XSD definitions of the XML elements are in $NDDSHOME/resources/schema/rti_converter.xsd.

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

Converter comes pre-configured with a configuration that converts a file called rti_recorder_default.db from XCDR format to a file called rti_recorder_default_converted.db in JSON_SQLITE format. See Section 5.2.21 for details.

5.2.4 XML Tags for Configuring Converter

This section describes the XML tags you can use in a Converter configuration file. Figure 5.1 and Table 5.2 describe the top-level tags allowed within the root <dds> tag.

Figure 5.1: Top-level Tags in the Converter Configuration File
Table 5.2: Top-Level Tags in Converter’s Configuration file

<table>
<thead>
<tr>
<th>Tags within &lt;dds&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;types&gt;</td>
<td>Defines types that can be used by Converter. This tag is needed if data types are not available in the discovery table. The type description is done using the Connext DDS XML format for type definitions. See Creating User Data Types with Extensible Markup Language (XML), in the RTI Connext DDS Core Libraries User’s Manual.</td>
<td>0..*</td>
</tr>
<tr>
<td>&lt;plugin_library&gt;</td>
<td>Contains a list of storage libraries that you may have implemented to support custom databases. For more information, see Section 3.3.12 and Section 4.3.15.</td>
<td>0..*</td>
</tr>
</tbody>
</table>
| <converter>      | **Required.** Specifies a Converter configuration. **Attributes**  
• name: uniquely identifies a Converter configuration. **Required.**  
**Example**  
```
<converter name="ConvertAll">
  <input_storage>  
  <output_storage>
</converter>
```
| 1..*             | | |

### 5.2.5 Converter Tag

A configuration file must have at least one `<converter>` tag. This tag is used to configure an execution of Converter.

A configuration file may contain multiple `<converter>` tags. When you start Converter, you can specify which `<converter>` tag to use to configure the service using the `-cfgName` command-line parameter. This means one file can be used to configure multiple Converter executions.

Figure 5.2 and Table 5.3 describe the tags allowed within a `<converter>` tag.

Table 5.3: Converter Tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;converter&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;input_storage&gt;</td>
<td>Describes the storage that Converter will read as input. Converter expects the storage to be in CDR (binary) format. If storage is not specified, data will be stored in a SQLite file using a default name. See Section 5.2.6.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;output_storage&gt;</td>
<td>Describes the storage that Converter will write as output. This output will be in JSON format. If this storage is not specified, data will be stored in a SQLite file using a default name. See Section 5.2.7.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

continues on next page
### Table 5.3 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;converter&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;data_selection&gt;</code></td>
<td>Enables selection of a subset of data to convert. Supports selecting data for conversion by time (or tagged time). See Section 5.2.15.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;session&gt;</code></td>
<td>Allows you to create one or more sessions in which to convert data, which gives you the option of specifying multiple threads for conversion. See Section 5.2.18.</td>
<td>0..*</td>
</tr>
<tr>
<td><code>&lt;domainParticipant&gt;</code></td>
<td><strong>Required.</strong> Describes the domains Converter will select for conversion. Converter is not a DDS application so this definition will not create a DDS Domain Participant, it just works as a domain ID selection utility. Note: the domain ID has to correspond to one of the domain IDs defined in the Recorder configuration. See Section 5.2.17.</td>
<td>1..*</td>
</tr>
</tbody>
</table>

**Attributes:**
- name: Uniquely defines a session. **Required.**
- default_participant_ref: Refers to a `domainParticipant` tag, that specifies the domain ID to use from the database when converting this Topic. Children can override this by specifying their own participant.

### Example: Specify a Configuration in XML

```xml
<dds>
  <converter name="ConvertToJSON">
    <!-- ... Required entities -->
  </converter>
</dds>
```

Starting Converter with the following command will use the `<converter>` tag with the name “ConvertToJSON”.

```
$NDDSHOME/bin/rticonverter -cfgFile file.xml -cfgName ConvertToJSON
```
Figure 5.2: Converter Tags in the Converter Configuration File
5.2.6 Input Storage

The `<input_storage>` tag allows you to configure the storage from which data will be read. You can choose between using the builtin SQLite storage or implementing your own storage plugin.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;input_storage&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sqlite&gt;</code></td>
<td>Enables converting data from an SQLite database file. See Section 5.2.8.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;legacy&gt;</code></td>
<td>Enables converting data from a legacy version of <em>Recording Service</em> to a database usable by this version. See Section 5.2.12.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;plugin&gt;</code></td>
<td>Enables converting data from storage using an external library that you specify. See Section 5.2.14.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;max_samples_per_read&gt;</code></td>
<td>Specifies the maximum number of samples to be returned by the read() operation. When the recorded tables contain huge amounts of samples, this setting can help improve the responsiveness of <em>Converter</em>, because the tool won’t - for each table - attempt to get all samples at once. If not specified, the default value is 1024 samples. A negative value means an unlimited number of samples.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

5.2.7 Output Storage

The `<output_storage>` tag allows you to configure the storage into which converted data will be written. You can choose between using the builtin SQLite storage, CSV storage utility plugin or implementing your own storage plugin.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;output_storage&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sqlite&gt;</code></td>
<td>Enables writing converted data into an SQLite database file. See Section 5.2.8.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;csv&gt;</code></td>
<td>Enables writing converted data with CSV format into a text file. See Section 5.2.9.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;plugin&gt;</code></td>
<td>Enables writing converted data into storage of your choosing, using a plugin library that you specify. See Section 5.2.14.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
5.2.8 SQLite

The `<sqlite>` tag allows you to specify the name and file extension of a SQLite file to read data from, or write data to. Currently, the only storage format supported for input is CDR, and the only format supported for output is JSON.

Table 5.6: SQLite Tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;sqlite&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;fileset&gt;</code> (output only)</td>
<td>Set of files to write to, and parameters for creating files and directories in that set. Used when creating more than one output file. See Section 5.2.10.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;file&gt;</code> (output only)</td>
<td>Name of file to write converted database file(s) to. Used when creating only a single output file. Default: rti_recorder_default Converted.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;file_suffix&gt;</code> (output only)</td>
<td>Suffix of the output file. Used when creating only a single output file. Default: dat.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;database_dir&gt;</code> (input only)</td>
<td>Directory containing file(s) to convert. Default: Current working directory.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;storage_format&gt;</code></td>
<td>Specifies what format the data is stored in. The options are:</td>
<td>0..1</td>
</tr>
<tr>
<td></td>
<td>• [Input Default] XCDR_AUTO: This is the binary format used by Connext DDS when sending data over the network. This has the highest performance for recording, but can only be viewed by using Converter to convert the data to a readable format, or by using Replay to replay the data. This will internally store data in XCDR or XCDR2 depending on the format received.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• JSON_SQLITE: This format can be queried, but recording in this format has lower performance because data must be deserialized before it can be stored.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• XCDR: The format to use when communicating with Connext DDS before 6.0.0.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• XCDR2: More efficient than XCDR, used by Connext DDS 6.0.0 and later.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;overwrite_policy_kind&gt;</code></td>
<td>Defines what to do when database files are already present in the current Converter execution directory.</td>
<td>0..1</td>
</tr>
<tr>
<td></td>
<td><strong>There are two options:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• OVERWRITE: delete old files and replace them with newly created ones.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• DO_NOT_TOUCH: do not delete any old files and just exit.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default: OVERWRITE</td>
<td></td>
</tr>
</tbody>
</table>

continues on next page
Table 5.6 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within &lt;sqlite&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;sql_initialization_string&gt;</code></td>
<td>Configures a SQL expression to use when establishing SQLite connections using this plugin. You can use this to add an index to a table to speed up replay, but this must be done for each table you want to index.</td>
<td>0..1</td>
</tr>
<tr>
<td>Example:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;sql_initialization_string&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DROP INDEX IF EXISTS pingtopic_0_by_→src_timestamp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CREATE INDEX pingtopic_0_by_→src_timestamp ON 'PingTopic@0'→(SampleInfo_source_timestamp);</td>
<td></td>
</tr>
<tr>
<td>Default:</td>
<td>PRAGMA SYNCHRONOUS = OFF; PRAGMA JOURNAL_MODE = MEMORY;</td>
<td></td>
</tr>
</tbody>
</table>

5.2.9 CSV

The CSV storage utility plugin allows you to specify where to output the converted files, if you want to merge them and what the basename prefix should be, among other things. The table below describes the various configuration tags.

Table 5.7: CSV storage utility tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;file-set&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;workspace_dir&gt;</code></td>
<td>Absolute or relative path to where generated file(s) are placed. Default: . (working directory)</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;file_basename&gt;</code></td>
<td>Prefix for the name of the generated file(s). The file generated for each topic has the following name: [file_basename]-[TOPIC_NAME].csv If <code>&lt;merge_files&gt;</code> is set to true, then the final file name is equal to [file_basename].csv. Default: <code>csv_converted</code></td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;merge_files&gt;</code></td>
<td>Specifies whether the generated files shall be consolidated into a single file. Default: <code>false</code></td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;default_member_value&gt;</code></td>
<td>Sets the value used for data members that are not present or empty. Default: <code>nil</code></td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;enum_as_string&gt;</code></td>
<td>Indicates whether values for enumeration members are printed as their corresponding label string or as an integer. Default: <code>true</code></td>
<td>0..1</td>
</tr>
</tbody>
</table>

To learn more about the CSV storage utility plugin see Section 7.1.1.

5.2. Converter Configuration
5.2.10 Fileset

The `<fileset>` tag allows you to specify a set of files for Converter to write to.

The behavior allowed by Converter is more limited than the behavior allowed by Recording Service.

Table 5.8: Fileset Tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;fileset&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;workspace_dir&gt;</code></td>
<td>Required. Base directory where Converter should output files. Must be different from the input database_dir directory. Default: The current working directory</td>
<td>1</td>
</tr>
</tbody>
</table>
| `<filename_expression>` | Required. The file name Converter will use for the generated user data file(s). This expression in combination with the rollover configuration will determine how many (and the names of) files Converter will output. This setting accepts text and any combination of the following parameters in it:  
  - Autonumeric. **Format:** `%auto:M%`. This parameter describes an integer that auto-increments every time Converter has a rollover event due to time or file size. **Example:**
    
    `<filename_expression>`
    test_files_%auto:0%.db
    `<filename_expression>`
    
    This will create a series of files starting with test_files_0.db, and incrementing each time Converter rolls over (due to the file size or time limits being reached).  
  - Timestamp. **Format:** `%ts%`. This parameter will take the current timestamp in the system (the time represented as number of seconds since Epoch).  
  - Time. **Format:** `%T%`. Current time expressed in ISO 8601 time format (HH:MM:SS). Example: 14:55:02. This parameter uses the strftime() parameter `%T`.  
  - Short date. **Format:** `%F%`. Short date in YYYY-MM-DD format. Example: 2001-08-23. This parameter uses the strftime() parameter `%F`. **Note:** this parameter will not vary in 24 hours, so use with caution in combination with the rollover time limit feature (time limit should be greater than 1 day; otherwise, you may overwrite the same file continuously).  
  - Date and time. **Format:** `%c%`. Date and time representation, locale-dependent. This parameter uses the strftime() parameter `%c`. Default: rti_recorder_default_%auto:0%.db (auto-numeric starting at zero, unlimited) | 1 |
| `<rollover>`            | Configuration for rolling over the file after a size limit is reached. See Section 5.2.11. | 0..1 |
5.2.11 Rollover

Rollover enables Converter to write to the next file in a set when a limit is reached. Converter’s rollover functionality is decoupled from the files that are input, so many files may be merged into one, or one file may be split into many depending on Converter’s rollover configuration. Currently Converter supports rolling over by size.

Table 5.9: Rollover in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;rollover&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| <file_size_limit>      | The maximum allowed size for a file in a set. Units can be specified as an attribute. **Attributes:**
|                        | - unit: (Optional) The unit in which the size is expressed. The following values are allowed (Default: KILOBYTES):
|                        |   - BYTES
|                        |   - KILOBYTES
|                        |   - MEGABYTES
|                        |   - GIGABYTES
| Example:               | `<file_size_limit unit="KILOBYTES">500</file_size_limit>` | 0..1 |

5.2.12 Legacy

This tag configures how legacy databases are converted to the newer format. (Note: You can continue to use the older format with Replay, by specifying “legacy” storage in Replay’s configuration.)

Table 5.10: Legacy in the Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;legacy&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;file_path&gt;</td>
<td>The path to a legacy file to be converted. File set and version properties will be obtained automatically from the file itself.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;domain_mapping&gt;</td>
<td>This tag allows you to link legacy domain names with domain IDs. See Section 5.2.13.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
5.2.13 Domain Mapping

When converting a legacy database, the domain may not have been recorded. This tag provides a way to map a table with a domain ID.

Later versions of the old Recorder database allowed you to use field filters. By default, the domain ID field was not recorded (it was filtered out by default). Thus, there is no information available in these legacy databases to relate the domain name used to record the data with a domain ID. This tag allows you to link legacy domain names with domain IDs.

Table 5.11: Domain Mapping in the Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;domain_mapping&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<domain_map>`                | **Required.** A link between a recorded legacy domain name and a domain ID. **Attributes:**  
  - `legacy_domain_name`: The recorded domain name specified in the old Recording Service domain tag, for example: `<domain name="domain0">`.  
  - `domain_id`: The domain ID you want to associate with the legacy domain name. | 1..* |

5.2.14 Plugin

This tag enables you to convert data using an external library that you specify.

Table 5.12: Storage plugin Tag in the Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;plugin&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<property>`          | Name/value pairs of properties to pass to a storage plugin. **Example:**  
  ```xml  
  <property>  
  <value>  
    <element>  
      <name>Name</name>  
      <value>Value</value>  
    </element>  
  </value>  
  </property>  
  ``` | 0 or 1 |
5.2.15 Data Selection

This tag selects what data to convert from the database, based on a time range.

Table 5.13: Data Selection Tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;data_selection&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;time_range&gt;</td>
<td>Select data to convert from the database, based on start and end times. See Section 5.2.16.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

5.2.16 Time Range

The <time_range> tag allows you to specify the begin and end times of the data you want to convert. This can be specified either as timestamps or as symbolic timestamps called “timestamp tags.” Tags may have been added to the recording through remote administration, and can be viewed using the script `rtirecordingservice_list_tags` (see Section 3.6.7).

Table 5.14: Time Range Tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;time_range&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;begin_time&gt;</td>
<td>Select data to convert from the database, with a timestamp beginning at this time. Specify the time in seconds and nanoseconds. Default: 0, start at beginning of the file.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;begin_tag&gt;</td>
<td>Can be used instead of begin_time. Select data to start converting from the database by specifying the string name of a timestamp tag. Timestamp tags associate a timestamp with a name. Then you can refer to a timestamp by that name. The timestamp must have been tagged with that name during recording. See Section 3.6.6. Default: Start at beginning of the file.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;end_time&gt;</td>
<td>Select data to convert from the database, with a timestamp ending at this time. Specify the time in seconds and nanoseconds. Default: End at end of file.</td>
<td>0..1</td>
</tr>
<tr>
<td>&lt;end_tag&gt;</td>
<td>Can be used instead of end_time. Select when to stop converting data by specifying the string name of a timestamp tag. Timestamp tags associate a timestamp with a name. Then you can refer to a timestamp by that name. The timestamp must have been tagged with that name during recording. See Section 3.6.6. Default: Stop at end of file.</td>
<td>0..1</td>
</tr>
</tbody>
</table>
5.2.17 DomainParticipant

The <domain_participant> tag allows you to specify the domain IDs you want to convert from the database. Specify a name for the DomainParticipant, and the domain_id you want to convert. Associate the DomainParticipant name with a session, topic or topic_group that you want to convert, to convert data within a domain.

This does not create a DDS DomainParticipant.

<table>
<thead>
<tr>
<th>Tags within &lt;domain_participant&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;domain_id&gt;</td>
<td>The domain ID of tables loaded from the database. This is used to determine which domain IDs to convert.</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.15: DomainParticipant Tags in Converter’s Configuration File

Tags within <register_type>  
Description: 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 a <topic>.

Attributes:
- name: Name that the data type is registered with if no <registered_name> is specified. The same data type may be registered with different names. Required.
- type_ref: Definition of this data type. It must refer to one of the defined types in the <types> section by specifying the fully qualified name.

<table>
<thead>
<tr>
<th>Tags within this tag:</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;registered_name&gt;: Name the data type is registered with. The same data type may be registered with different names. Not required.</td>
</tr>
</tbody>
</table>

5.2.18 Session

The <session> tag configures the threads that will be used to convert data. You also specify the Topics and groups of Topics to convert inside the <session> tag.

<table>
<thead>
<tr>
<th>Tags within &lt;session&gt;</th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;thread_pool&gt;</td>
<td>Specifies the number of threads used by this session, and the settings used when creating them.</td>
<td>0..1</td>
</tr>
</tbody>
</table>

continues on next page
5.2.19 Topic Group

The `<topic_group>` tag allows you to convert a group of Topics, using regular expressions to describe which Topics to convert.

Table 5.17: Topic Group Tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;topic_group&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<allow_topic_name_filter>` | A regular expression (fnmatch) describing which Topics should be converted. You may use a comma-separated list to specify more than one filter. **Example:**  
  `<topic_group name="ConvertAll">  
  <allow_topic_name_filter>CONTROL_*,DATA_*</allow_topic_name_filter>  
  </topic_group>` | 0..1 |
| `<deny_topic_name_filter>`  | A regular expression (FNMATCH) describing which Topics should not be converted. This is applied after the `allow_topic_name_filter`. You may use a comma-separated list to specify more than one filter. | 0..1 |
5.2.20 Topic

The `<topic>` tag specifies an individual Topic to convert.

Table 5.18: Topic Tags in Converter’s Configuration File

<table>
<thead>
<tr>
<th>Tags within <code>&lt;topic&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;topic_name&gt;</code></td>
<td>The name of the DDS topic to be converted. If this tag is not present, the name attribute of the <code>&lt;topic&gt;</code> will be used. <strong>Note:</strong> we recommend using this tag to define the topic name. There may be characters that cause the XML validation to fail if they are part of the topic name attribute. Also, the ‘/’ character and ‘::’ separator may cause Converter to fail when found in the topic name attribute.</td>
<td>0..1</td>
</tr>
<tr>
<td><code>&lt;registered_type_name&gt;</code></td>
<td>The name of the data type that will be converted for this topic. <strong>Required.</strong></td>
<td>1</td>
</tr>
<tr>
<td><code>&lt;transformation&gt;</code></td>
<td>The transformation library to be applied to this Topic’s data when converting. This is a user library that can modify the data after it is received from input storage and before it is sent to output storage. Transformations implement APIs identical to Routing Service’s transformations. For more on using transformations, see these sections in the RTI Routing Service User’s Manual: • Data Transformation • Tutorials Attributes: • plugin_name: The name of the plugin to load, qualified by the plugin library name. <strong>Example:</strong></td>
<td>0..1</td>
</tr>
</tbody>
</table>

Example:

```xml
<dds>
  <plugin_library name="ConverterTransformations">
    <transformation_plugin name="ModifyTestID">
      <create_function>ModifyTestID_create</create_function>
      <dll>modify_test_id_library</dll>
    </transformation_plugin>
  </plugin_library>
  <!-- ... -->
  <converter>
    <!-- ... -->
    <topic name="TestTopic">
      <transformation plugin_name="ConverterTransformations::ModifyTestID" />
    </topic>
  </converter>
</dds>
```
5.2.21 Converter’s Built-in Configuration Details

Converter comes with a built-in configuration, which selects the name of an input file to convert from and an output file to convert to. The ‘defaultToJson’ configuration specifies JSON SQLite format by default as the output format, with ‘rti_recorder_default.db’ as the input file and ‘rti_recorder_default_converted.db’ as the output file. It expects to find the file(s) to convert in a directory called ‘cdr_recording’. This is the default directory when recording in XCDR format in the default Recorder configuration.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://community.rti.com/schema/6.0.0/rti_converter.xsd">

  <!-- Available types -->
  <types />

  <!-- A definition of a Converter instance to run -->
  <converter name="defaultToJson">
    <!-- Input storage settings -->
    <input_storage>
      <sqlite>
        <storage_format>XCDR_AUTO</storage_format>
        <database_dir>cdr_recording</database_dir>
      </sqlite>
    </input_storage>

    <!-- Output storage settings -->
    <output_storage>
      <sqlite>
        <storage_format>JSON_SQLITE</storage_format>
        <fileset>
          <workspace_dir>converted</workspace_dir>
          <filename_expression>rti_recorder_default_converted.db</filename_expression>
        </fileset>
      </sqlite>
    </output_storage>

    <!-- Domain selection: assume 0 by default -->
    <domain_participant name="Domain0">
      <domain_id>0</domain_id>
    </domain_participant>
  </converter>
</dds>
```
5.3 Tutorials

5.3.1 Using Timestamp Tags with Converter

If your recording was originally made with the builtin SQLite storage plugin, and you used the tag_timestamp remote command to tag certain events, then your recording contains timestamp tags: symbolic timestamp names you can use in place of timestamps expressed in units of time. For more information on timestamp tags, see Section 3.6.6.

You can list the timestamp tags that are in your recorded database by using the `rtirecordingservice_list_tags` script. Use the `-d` argument to point to the directory that contains your recorded database, as follows:

```bash
<NDDSHOME>/bin/rtirecordingservice_list_tags -d /database/directory/
```

This command will analyze the recording in `/database/directory/` and list the details of the timestamp tags in the recording, including the tag names, their descriptions, and associated timestamps.

You can use the `tag_name` of the tags you find in a recording when you are creating an XML configuration file for `Converter` by using a `<data_selection>` tag.

For example, if after running `rtirecordingservice_list_tags`, you see output such as:

<table>
<thead>
<tr>
<th>tag_name</th>
<th>timestamp_ms</th>
<th>tag_description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/my_example/my_events/tag1</td>
<td>1546484663309</td>
<td>first tag description</td>
</tr>
<tr>
<td>/my_example/my_events/tag2</td>
<td>1546484703360</td>
<td>a second tag description</td>
</tr>
</tbody>
</table>

Then you can have a `<data_selection>` tag in your XML for `Converter`, after the `<output_storage>` tag, that looks like the following:

```xml
<data_selection>
  <time_range>
    <begin_tag>/my_example/my_events/tag1</begin_tag>
    <end_tag>/my_example/my_events/tag2</end_tag>
  </time_range>
</data_selection>
```

`Converter` will convert data between those tags. Note that when expressing a `<time_range>` tag, you can mix and match timestamps and tags. For example, you can use a `<begin_tag>` (by referring to a `tag_name`) to express the time when conversion should begin, and use an `<end_tag>` with an `end_time` timestamp (expressed in time units) to specify when conversion should end. If you do not provide one of the bounds, the start of recording is the default begin bound, and the end of recording is the default end bound.
5.4 Troubleshooting

5.4.1 Table Not Found Errors

When recording a database with Recording Service, there may be topics that have no associated table, because they were discovered but were filtered out by using the `<allow_topic_name_filter>` or `<deny_topic_name_filter>` tags in Topic Group or by defining Topics (that target specific topic names). While the topic will be present in the DCPSPublication table in the discovery file, it won’t have a corresponding table in the user-data files.

If the same topics are not filtered in Converter (by using the same `<allow_topic_name_filter>` or `<deny_topic_name_filter>` tags, or Topics), then when Converter starts it will discover the topics without a table because they are available in the discovery information. When Converter attempts to create a stream reader for these topic(s), a failure message will be printed:

```
ROUTERConnection_createStreamReaderAdapter:(adapter=StorageAdapterPlugin,␣ →retcode=0: Function returned NULL)
ROUTERStreamReader_enable:!create stream reader adapter
ROUTERTopicRoute_enableInput:!enable stream reader
ROUTERTopicRoute_processEvent:!enable route input
ROUTERTopicRoute_onConditionTriggered:!process event
create_stream_reader_fwd:SQLiteStorageStreamReader:!Table not found in␣ →database files: TopicNotRecorded@0
```

These messages are harmless, they are just informing you that a table could not be found for the topic (in the example above, TopicNotRecorded).

To get rid of the messages, use the same Topic Group filter expressions or Topics used in Recording Service.
Chapter 6

XML Converter

Recording Service includes an application to convert a legacy XML configuration to the current XML configuration format when possible. It can convert a recording XML configuration, as well as a replay XML configuration.

6.1 Running the XML converter

XML Converter runs as a separate application. The script to run the executable is in <NDDSHOME>/bin. (See Section 1.3 for the path to NDDSHOME.)

rtixmlconverter record|replay [options]

If you are using features that are not available in Recording Service, you will see warnings such as:

Warning: Recording Service does not currently support 'verbosity' in XML. However, you can specify -verbosity at the command-line.

6.2 XMLConverter Command-Line Parameters

The following table describes all the command-line parameters available for the XML Converter. To list the available parameters, run rtixmlconverter -help.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>play</td>
</tr>
<tr>
<td>-cfgFile</td>
<td>Required. The legacy configuration file to be converted.</td>
</tr>
<tr>
<td>-cfgFileOut</td>
<td>Required. The name of the generated configuration file.</td>
</tr>
<tr>
<td>-help</td>
<td>Shows help for the command.</td>
</tr>
<tr>
<td>-version</td>
<td>Prints the program version and exits.</td>
</tr>
</tbody>
</table>
Chapter 7

Storage Utility Plugins

This module contains information about the various Recording Service Storage Plugins that are shipped in a separate library called rtistorageutils. This library is meant to be an extension of the capabilities supported by RTI Recording Service out-of-the-box. It is located under <NDDSHOME>/lib like other RTI Connext® DDS libraries.

In this release, there is one plugin for CSV format. Other plugins may be added in future releases.

Note: The different executables shipped with Recording Service will perform run-time loading for rtistorageutils if you use one of these Storage Utility Plugins in your XML configuration. Thus make sure that <NDDSHOME>/lib/<architecture> is included in your system’s library search path. See Section 7.2.1.

7.1 Storage Utility Plugins

7.1.1 CSV

This is a type of output storage plugin that stores the data provided to it in Comma-Separated Value (CSV) format. It is meant to be used with Converter to perform conversion of a recorded database into CSV output files. Typically it helps solve use cases like offline analysis or incident investigation. For more information about the configuration tags, see Section 5.2.9.

By default, the plug-in generates a separate csv file for each recorded Topic. The content and format of each file is as follows:
Table 7.1: CSV file format

<table>
<thead>
<tr>
<th>Row 1: Topic Entry</th>
<th>Topic name: &lt;Name&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 2: Type Header</td>
<td>timestamp</td>
</tr>
<tr>
<td>Row 3: Data values</td>
<td>Reception timestamp of the sample (in nanoseconds)</td>
</tr>
</tbody>
</table>

The filename matches the topic name, with a `.csv` extension. All the files are placed in a directory that can be specified in the plug-in configuration.

**Note:** If the topic name contains characters that cannot appear in the file name because they are reserved by the underlying operating system, these characters will be replaced by the token `#`.

**Mapping a data sample into columns**

**General case**

As shown in the table above, a data sample is represented in a single row comprised of multiple columns. Each cell holds only a value for a final or leaf member in a complex data type. That is, for a member whose type is a Simple type (integer, char, short), enumeration, or String.

Each data value in a cell corresponds to a member whose name is in the type header. Given a complex data type, the name for a member is constructed as follows:

```
.<parent_member1>.<parent_member2>...<parent_memberN>.<final_member>
```

where `<parent_member>` is the name of the parent complex member that contains the subsequent member.

If either `<parent_member>` or `<final_member>` is a Collection type (array or sequence), the member name is suffixed with `[<index>]`, and a column for each possible element in the Collection is created.

For example, consider the following type described in IDL:

```idl
struct NestedStruct {
  long m_long;
};

struct TopLevelStruct {
  String m_string;
  long m_array[2];
  NestedStruct m_complex;
  NestedStruct m_complex_array[2];
};
```

The resulting type header row will look like this:

```
<parent_member1>.<parent_member2>...<parent_memberN>.<final_member>
```
Note: Considerations about collection types: Due to the column consistency required by the CSV format, mapping a collection type requires us to generate as many columns as the number of elements that can be present in the collection, even if for a given data sample only a few of them are present (such as for a Sequence type). If the recorded type has collections with large sizes, the generated file may hit the column limit of some CSV processors.

Sequences

Mapping a Sequence type is based on the mapping of a Collection type explained above, plus an additional column to indicate the length of the sequence. That is:

```
.<seq_member.length>.<seq_member[0]>...<seq_member[N-1]>
```

where the column `<seq_member.length>` indicates how many elements are set in the sequence (the number of columns with non-null values) and N is the maximum length of the sequence.

For example, consider the following type described in IDL:

```
struct StructType {
    sequence<long, 4> m_seq;
};
```

The resulting type header row will look like this:

```
.m_seq.length  .m_seq[0]  .m_seq[1]  .m_seq[2]  .m_seq[3]
```

Unions

The mapping of a Union type is similar to a Struct type except that a discriminator column with name disc is placed before all the members.

For example, consider the following type described in IDL:

```
union UnionType switch (long) {
    case 0:
        long case1;
        
    case 1:
        StructType case2;
        
    default:
        long case_default;
}
```

(continues on next page)
The resulting type header row will look like this:

| .m_union.disc | .m_union.case1 | .m_union.case2 | .m_union.default |

**Data Values**

For a given data sample, the value for each member is placed under the corresponding column represented as a String, which applies to all primitive types. For primitive types, the output text will correspond to the standard conventions for the type (e.g., decimal point for floating point numbers, single quotes around a character, etc.). By default, enumerations are printed with their corresponding text label. This behavior for enums is configurable.

There may not be values for every column in a sample. This may occur for the following situations:

- A Sequence member that does not contain all the possible elements.
- A Union member, which can only set a member at a time.
- An optional member, which may or may not be set.

By default, the value of an empty member is represented as nil.

For example, consider the following type described in IDL:

```idl
struct StructType {
    UnionType m_union;
};
```

And two samples with the following values (represented in JSON):

```json
{
    "m_sequence": [1, 2],
    "m_optional": "hello"
}
{
    "m_sequence": [1],
    "m_optional":
}
```

The resulting type header row and the two data value rows will look like this:
### 7.2 Tutorials

#### 7.2.1 Using the CSV storage utility plugin with Converter

If you have a recorded database in any of the binary CDR serialized formats, it is pretty difficult to analyze the recorded data without replaying it. While there is an option to convert the recorded database to JSON (see Section 5.2.21), you can also convert it to CSV format.

The Converter can convert an XCDR database to CSV format using a default configuration called sqlite-ToCsv.

**Setup**

We need to make sure rtistorageutils can be loaded dynamically by rticonverter, set the library search path variable as follows:

**Linux**

```bash
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH;<NDDSHOME>/lib/<architecture>
```

**macOS**

```bash
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH;<NDDSHOME>/lib/<architecture>
```

**Windows**

```bash
set PATH=%PATH%;<NDDSHOME>\lib\<architecture>
```

**Note:** `<NDDSHOME>` is described in section Section 1.3.

**Execution**

To demonstrate the functionality we first need to record some data. Run Shapes Demo with the default configuration and publish some Triangles and Circles (using Shapes Demo is described in Section 3.6.1). Once Shapes Demo has started, run Recording Service as follows:

```
<NDDSHOME>/bin/rtirecordingservice
```
The default configuration should record data in a directory called `cdr_recording`.

Now run `Converter` to perform the conversion of the previously generated database into a single text file in CSV format:

```
<NDDSHOME>/bin/rticonverter -cfgName sqliteToCsv
```

If you need to customize the behavior of this configuration, you can do so by adding user variables to the command, as shown:

```
<NDDSHOME>/bin/rticonverter -cfgName sqliteToCsv -DCSV_WORKSPACE_DIR=my_custom_dir -DCSV_MERGE_FILES=false -DINPUT_DIR=my_cdr_recording_dir
```

Wait until conversion of the entire input database is performed and `rticonverter` exits automatically. Upon successful conversion, you will find a folder in the current directory with the name `csv_output`. That folder will contain a file called `csv_converted.csv` containing topic data for the Triangles and Circles published earlier by the `Shapes Demo` and recorded by `rtirecordingservice`. This can be verified by opening the file using an editor of your choice.
Chapter 8

Indexing Application

*Recording Service* includes a command-line utility for indexing the database files.

There are two types of indexing: instance indexing and SQLite table indexing.

### 8.1 Indexing Instances

Instance indexing is used by *Replay Service* for the Instance History Replay (state of the world publication) feature (see Section 4.1.7 for more information). *Replay Service* will perform the indexing if the index is not present. While this may be acceptable in some cases, it can take a long time if the database is large. This command-line option offers offline indexing, which can be done before running *Replay Service*. Recall that *Recording Service* also offers a way to index instances while recording (see Section 3.3.6 for more information).

### 8.2 Indexing SQLite Tables

Table indexing improves the performance of *Replay Service* and Converter when accessing databases recorded with the built-in SQLite plugin. In this case, the indexer will go over all files and all available tables and create two SQLite indexes: one on the reception timestamp column, and another on the source timestamp column.

For a given table named T1, the indexer will create an index called T1_rt_idx based on the Sample-Info_reception_timestamp column, and another index called T1_st_idx based on the Sample-Info_source_timestamp column.

When dealing with very large databases, having indexes on the timestamp column(s) can be extremely important because it can significantly improve performance.
8.3 Running the Indexer

The Indexer runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin`. (See Section 1.3 for the path to NDDSHOME.)

```
rtirecordingindexer [options]
```

8.4 Indexer Command-Line Parameters

The following table describes all the command-line parameters available for the Indexer. To list the available parameters, run `rtirecordingindexer -help`.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>instances</td>
<td>tables</td>
</tr>
<tr>
<td>-dbDirectory</td>
<td><strong>Required.</strong> The directory containing the SQLite database to be indexed.</td>
</tr>
<tr>
<td>-help</td>
<td>Shows help for the command.</td>
</tr>
</tbody>
</table>
| -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 program version and exits.                                       |
Chapter 9

Software Development Kit

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

The SDK is divided into the following modules:

- **RTI Recording Service** Service API: This module offers a set of APIs that allow you to instantiate **Recording Service** instances in your application. This allows you to run **Recording Service** as a library.

- **RTI Recording Service** Storage API: **Storage** is a pluggable component that allows **Recording Service** to write and read data from custom storage. This module offers a set of pluggable APIs to develop custom **StorageWriter** and **StorageReader**, which you can use through shared libraries or through the Service API. By default, **Recording Service** is distributed with a built-in SQLite® storage plugin that is part of the service library.

- **RTI Recording Service** Transformation API: **Transformations** are data-oriented pluggable components that allow you to perform conversions of the representation and content of the data that goes through **Recording Service**. This module offers a set of pluggable APIs to develop a custom **Transformations**, which you can use through shared libraries or through the Service API.

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

<table>
<thead>
<tr>
<th>Language API</th>
<th>Available Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTI Recording Service C API</td>
<td>• Storage</td>
</tr>
<tr>
<td></td>
<td>• Transformation (see RTI Routing Service C API)</td>
</tr>
<tr>
<td>RTI Recording Service C++ API</td>
<td>• Service</td>
</tr>
<tr>
<td></td>
<td>• Storage</td>
</tr>
<tr>
<td></td>
<td>• Transformation (see RTI Routing Service C++ API)</td>
</tr>
</tbody>
</table>
Chapter 10

Common Infrastructure

10.1 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.

10.1.1 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 Service API\(^1\).

Shipped Executable

Use the `-cfgFile` option.

\(^1\) Service 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:

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

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

Service API

Set the ServiceProperty::cfg_file member.

For example in C++:

```cpp
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 Service 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++:

```cpp
std::vector<std::string> xml_strings;
xml_strings.resize(2);
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 Service API.

For example, consider the following input XML document in a file named MyService.xml that contains two configurations for RTI Recording Service:
You can run the configuration for Service1 as follows:

**Shipped Executable**

Use the `-cfgName` option.

For example in Linux:

```
$ /opt/rti/RTI_Concurrent/rti/bin/rtiservices -cfgFile file.xml -cfgName Service1
```

**Service API**

Set the `ServiceProperty::cfg_name` member.

For example in C++:

```cpp
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>
<thead>
<tr>
<th>File</th>
<th>Allowed Content</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[working directory]/USER_[SERVICE].xml</code></td>
<td>• Service-specific elements</td>
</tr>
<tr>
<td></td>
<td>• QoS profiles</td>
</tr>
<tr>
<td></td>
<td>• Types</td>
</tr>
<tr>
<td><code>[NDDSHOME]/resource/xml/RTI_[SERVICE].xml</code></td>
<td>• Service-specific elements</td>
</tr>
<tr>
<td></td>
<td>• QoS profiles</td>
</tr>
<tr>
<td></td>
<td>• Types</td>
</tr>
<tr>
<td><code>[working directory]/USER_QOS_PRORFILES.xml</code></td>
<td>• QoS profiles</td>
</tr>
<tr>
<td></td>
<td>• Types</td>
</tr>
</tbody>
</table>
where [SERVICE] refers to the concrete product name in uppercase. For example ROUTING_SERVICE for RTI Routing Service or RECORDING_SERVICE for RTI Recording 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.

Service 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:

```
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.

Service 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* or `recording_service` for *RTI Recording 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
<?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 built-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:

```bash
rtiroutingservice -listConfig
Available configurations:
- default: (/Users/asanchez/rti/waveworks/develop/router.1.0/resource/xml/RTI_ROUTING_SERVICE.xml)
  Routes all topics from domain 0 to domain 1
- defaultBothWays: (/Users/asanchez/rti/waveworks/develop/router.1.0/resource/xml/RTI_ROUTING_SERVICE.xml)
  Routes all topics from domain 0 to domain 1 and the other way around
- defaultReliable: (/Users/asanchez/rti/waveworks/develop/router.1.0/resource/xml/RTI_ROUTING_SERVICE.xml)
  Routes all topics from domain 0 to domain 1 using reliable communication
```

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 built-in XML parser of the RTI service offers a special mechanism to reuse and customize content at runtime 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:

$$ (\text{VAR\_NAME}) $$

where \text{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.

```xml
<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.

   ```bash
   export VAR\_NAME=my_value
   ```

2. Using a specific option when running the service.

   **ShippedExecutable**

   Use the -DVAR\_NAME=VALUE option

   ```bash
   rtirecordingservice ... -DVAR\_NAME=my_value
   ```

   **Service API**

   Set the ServiceProperty::user_environment member

   ```java
   ServiceProperty property;
   property.user_environment()["VAR\_NAME"] = "var_value";
   ...
   ```

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

   ```xml
   <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:

10.1. Configuring RTI Services
export VAR_ATTR=expanded_attr
rtiroutingservice ... -DVAR_TEXT=expanded_text

will result in the following actual parsed XML with the expanded variables:

```xml
<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:

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

### 10.1.2 How to Load Default QoS Profiles

Generally, loading a default QoS profile follows the same mechanism as *Connext DDS* applications. The details on how to specify default QoS profiles in XML is explained in the section *Overwriting Default QoS* in the *RTI Connext DDS 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.
10.1.3 How to Set Logging Properties

You can configure different aspects of the logging infrastructure that is part of RTI services and *Connext DDS*. 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 DDS*.
- `-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.

**Service API**

To configure the service-level verbosity, use the `Logger` singleton class part of the service API. For example, the following sets `WARNING` level for the service logs:

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

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

```cpp
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 DDS* configuration logger API. For example, to redirect the logging to an output file:

```cpp
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 DDS 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 DDS* which one to use, you will need to mark the profile as the default using the `is_default_qos` attribute, or for the DomainParticipantFactory, the `is_default_participant_factory_profile` attribute.
See *How to Load Default QoS Profiles* for details on how to load default QoS profiles with RTI services. For example, you can set different properties for the logger by placing the XML code seen below in the USER_QOS_PROFILES.xml default file:

```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:

**Controlling Messages from Connext DDS** 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.

### 10.1.4 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 Service API to create your own executable that instantiates the RTI service and sets the running process as a daemon using the `deamon()` API. For example, for *Routing Service*:

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

int main(int argc, char **argv)
{
  using namespace rti::routing;
  // your service implementation here
}
```

(continues on next page)
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:

```bash
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:

Alternatively, you can use the Service API to embed the RTI service into your own executable and implement the Windows Service 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.

10.1. Configuring RTI Services
• In some scenarios, you may need to add a multicast address to your discovery peers or simply use RTI Cloud Discovery Service.

10.1.5 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.

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

10.2 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 10.1 shows a general view of this concept.

Figure 10.1: 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 or more *child* Resources.

### 10.2.1 Example: Simple Resource Model of a Connext DDS Application

Figure 10.2 depicts a UML class diagram to provide a generic resource model for *Connext DDS* applications.

![UML class diagram](image)

Figure 10.2: Connext DDS application resource model

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

### 10.2.2 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:

```
/collection_id/resource_id/resource_id
```

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

```
<service name="A/B">
<service name="A:B">
```
in the resource identifier.

**Example: Resource Identifiers of a Generic Connext DDS Application**

Consider the *Connext DDS* application resource model in *Example: Simple Resource Model of a Connext DDS 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 *DomainParticipant* 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"
```
10.3 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*, and *RTI Queuing Service*. The *RTI Remote Administration Platform* provides a common infrastructure that unifies and consolidates the remote interface to all RTI services.

**Note:** Remote administration of RTI services requires an understanding of the *application resource model*. We recommend that you read [Application Resource Model](#) 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 10.3. 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 10.3: General View of the RTI Remote Administration Platform Architecture](image)

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.
10.3.1 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 10.2: REST Interface

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REST Request</td>
<td>[method] + [resource_identifier] + [body]</td>
</tr>
<tr>
<td></td>
<td>• method: Specifies the action to be performed on a service resource.</td>
</tr>
<tr>
<td></td>
<td>• resource_identifier: Addresses a concrete service resource.</td>
</tr>
<tr>
<td></td>
<td>• body: Optional request data that contains necessary information to</td>
</tr>
<tr>
<td></td>
<td>complete the operation.</td>
</tr>
<tr>
<td>REST Reply</td>
<td>[return code] + [body]</td>
</tr>
<tr>
<td></td>
<td>• return code: Integer indicating the result of the operation.</td>
</tr>
<tr>
<td></td>
<td>• body: Optional reply data that contains information associated with the</td>
</tr>
<tr>
<td></td>
<td>processing of the request.</td>
</tr>
</tbody>
</table>

### Standard Methods

The *RTI Remote Administration Platform* defines the methods listed in Table 10.3.

Table 10.3: Standard Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>URI</th>
<th>Request Body</th>
<th>Reply Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>Parent collection resource identifier</td>
<td>Resource representation</td>
<td>N/A</td>
</tr>
<tr>
<td>GET</td>
<td>Resource identifier</td>
<td>N/A</td>
<td>Resource representation</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Resource identifier</td>
<td>Resource representation</td>
<td>N/A</td>
</tr>
<tr>
<td>DELETE</td>
<td>Resource identifier</td>
<td>Undefined</td>
<td>N/A</td>
</tr>
</tbody>
</table>
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
```

10.3.2 Communication

The information exchange between client and server is based on the DDS request-reply pattern, as shown in Figure 10.4. The client maps to a Requester, whereas the server maps to a Replier.

![Figure 10.4: Communication in RTI Remote Administration Platform is Based on DDS Request-Reply](image)

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 10.4:

<table>
<thead>
<tr>
<th>Topic</th>
<th>Name</th>
<th>Top-level Type Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommandRequestTopic</td>
<td>rti/service/administration/command_request</td>
<td>rti::service::administration::CommandRequest</td>
</tr>
<tr>
<td>CommandReplyTopic</td>
<td>rti/service/administration/command_reply</td>
<td>rti::service::administration::CommandReply</td>
</tr>
</tbody>
</table>

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

**Listing 10.1: CommandRequest Type**

```c
@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 10.5: CommandRequest**

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

**Listing 10.2: CommandReply Type**

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

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>retcode</td>
<td>Indicates the result of the operation.</td>
</tr>
<tr>
<td>native_retcode</td>
<td>Provides extra information about the result of the operation.</td>
</tr>
<tr>
<td>string_body</td>
<td>Return value of the operation, represented as a string.</td>
</tr>
<tr>
<td>octet_body</td>
<td>Return value of the operation, represented as binary.</td>
</tr>
</tbody>
</table>

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

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

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

### Reply Sequence

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

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

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

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

For more on SampleFlagBits, see documentation on the DDS_SampleInfo structure in the Connext DDS API Reference HTML documentation.
Example: Accessing from Connext DDS Application

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

The first step is to generate code for administration types in [NDDSHOME]/resource/idl/ServiceAdmin.idl (which has a dependency on [NDDSHOME]/resource/idl/ServiceCommon.idl):

```
rtiddsgen -language C++11 -unboundedSupport -qualifiedEnumerator
          ServiceCommon.idl
rtiddsgen -language C++11 -unboundedSupport -qualifiedEnumerator
          ServiceAdmin.idl
```

Note that there are two important options in the generation command:

- `-unboundedSupport` is required to support unbounded length for the request's and reply's octet and string bodies. This allows sending and receiving any-size data in a single sample. Because of the unbounded support, you will need to make sure that the requester's `DataWriter` and `DataReader` are configured with a QoS that sets the buffer pools’ memory settings (in addition to any other QoS). For example:

```
<!-- For requester's request DW -->
<datawriter_qos>
  <property>
    <element>
      <name>dds.data_writer.history.memory_manager.
          fast_pool.pool_buffer_max_size</name>
      <value>16384</value>
    </element>
  </property>
</datawriter_qos>

<!-- For requester's reply DR -->
<datareader_qos>
  <reader_resource_limits>
    <dynamically_allocate_fragmented_samples>true</dynamically_allocate_fragmented_samples>
  </reader_resource_limits>
  <property>
    <element>
      <name>dds.data_reader.history.memory_manager.
          fast_pool.pool_buffer_max_size</name>
      <value>16384</value>
    </element>
  </property>
</datareader_qos>
```

- `-qualifiedEnumerator` generates enum labels with the proper namespace based on the IDL mod-
Then you create your requester application that handles the requests and replies. For example:

```cpp
#include "ServiceAdmin.hpp"
#include "ServiceCommon.hpp"

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

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

int main(int, char *[]) {
  try {
    dds::domain::DomainParticipant participant(ADMIN_DOMAIN_ID);

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

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

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

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

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

(continues on next page)
typedef void (*RequestHandler)(Request *request);

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

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

return 0;

You will need to compile your application code along with the generated files and link with the standard Connext DDS libraries required for the Modern C++ API in addition to the message API (rticonnextmsgcpp2).

In RTI Community Examples Repository you can see a full example for an administration requester that shows how to send different commands to RTI Recording Service.

### 10.3.3 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:
  ```
  <my_resource name="MyObject">
  ...
  </my_resource>
  ```

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.
Update Resource

**UPDATE [resource_identifier]**

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

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

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

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

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

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

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

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

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

**Request body**

- `string_body`: XML representation of the resource object provided as file:// or str://.
- Example `str://` request body:
  ```
  str://"<my_resource name="MyResourceObject">...
  </my_resource>"
  ```

- Example `file://` request body:
  ```
  file:///home/rti/config/service_update_my_resource.xml
  ```

**Reply body**
• Empty.

Return codes
The operation may return a reply with error if:
• The specified resource identifier does not exist.
• The specified configuration is schematically invalid.
• The specified configuration contains changes in immutable properties.
• There was an error updating the resource object.

Set Resource State

UPDATE [resource_identifier]/state
Sends a state change request to the specified resource object.

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

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

Request body
• octet_body: CDR representation of an entity state.

Reply body
• Empty.

Return codes
The operation may return a reply with error if:
• The specified resource identifier does not exist.
• The target request is invalid.
• The resource object reported an error while performing the state transition.

Get Resource State

GET [resource_identifier]/state
Gets the current state of the specified resource object.

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

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

Request body
• Empty

Reply body
• octet_body: CDR representation of an entity’s current state.
Return codes

The operation may return a reply with error if:

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

Delete Resource

DELETE [resource_identifier]

Deletes the specified resource object.

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

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

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

Request body

- Empty.

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

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

10.4 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 applications 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 Topics are shared across all services for the distribution of the monitoring information. Table 10.7 provides a summary of these topics.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Name</th>
<th>Top-level Type Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigDistributionTopic</td>
<td>rti/service/monitoring/config</td>
<td>rti::service::monitoring::Config</td>
</tr>
<tr>
<td>EventDistributionTopic</td>
<td>rti/service/monitoring/event</td>
<td>rti::service::monitoring::Event</td>
</tr>
<tr>
<td>PeriodicDistributionTopic</td>
<td>rti/service/monitoring/periodic</td>
<td>rti::service::monitoring::Periodic</td>
</tr>
</tbody>
</table>

Figure 10.5 shows the mapping of the monitoring information into the distribution Topics. A distribution Topic is keyed on service resources categorized as keyed Resources. These are resources whose related monitoring information is provided as an instance on the distribution Topic.

Figure 10.5: Monitoring Distribution Topics of RTI Services
10.4.1 Distribution Topic Definition

All distribution Topics 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 10.6.

![Figure 10.6: Monitoring Distribution Topic Definition](image)

**Keyed Resource Base Type Fields**

This is the base type of all distribution Topics 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 Topics, 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:

```idl
#include "ServiceCommon.idl"

@nested
struct CarType {
    float speed;
    String color;
    String plate_number;
};

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

@nested
struct PlaneType {
    float ground_speed;
    int32 air_track;
};

enum TransportModeKind {
    CAR_TRANSPORT_MODE,
    BOAT_TRANSPORT_MODE,
    PLANE_TRANSPORT_MODE
};

@nested
union TransportModeUnion switch (TransportModeKind) {
    case CAR_TRANSPORT_MODE:
        CarType car;
    case BOAT_TRANSPORT_MODE:
        BoatType boat;
    case PLANE_TRANSPORT_MODE:
        PlaneType plane;
}

struct TransportModeDistribution : KeyedResource {
    TransportModeUnion value;
};
```
Assume now that in the monitoring domain there are three resource objects, one for each resource class: a Car object ‘CarA’, a Boat object ‘Boat1’, and a Plane object ‘PlaneX’. They all have unique resource GUIDs and each object represents an instance in the distribution Topic. The table shows the example of potential sample values:

<table>
<thead>
<tr>
<th>object_guid</th>
<th>CarA</th>
<th>Boat1</th>
<th>PlaneX</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent_guid</td>
<td>0x00</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>value_discriminator</td>
<td>CAR_TRANS-PORT_MODE</td>
<td>BOAT_TRANS-PORT_MODE</td>
<td>PLANE_TRANS-PORT_MODE</td>
</tr>
</tbody>
</table>

### 10.4.2 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 DataWriters.
- 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.

### 10.4.3 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 continuous flow of metrics, represented with the StatisticVariable type (see StatisticVariable).

There are two activities involved in the generation of the statistic variables: Calculation and Publication. All the configuration elements for these activities are available under the <monitoring> tag.

Calculation

The instrumented service periodically performs measurements on the metric. This activity is also known as sampling (don’t confuse with data samples). The frequency of the measurements can be configured with the tag <statistics_sampling_period>. As a general recommendation, the sampling period should be a few times smaller than the publication period. A small sampling period provides more accurate statistics generation at the expense of increasing memory and CPU consumption.

Publication

The service periodically publishes a data sample containing a snapshot of the statistics generated during the calculation phase. The publication period can be configured with the tag <statistics_publication_period>. The value of a statistic variable corresponds to the time window of a publication period.

10.4.4 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 10.3: Statistics

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

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

Table 10.9: StatisticMetrics

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>period_ms</td>
<td>Period in milliseconds at which the metrics are published.</td>
</tr>
<tr>
<td>count</td>
<td>Sum of all the measurement values obtained during the publication period.</td>
</tr>
<tr>
<td>mean</td>
<td>Arithmetic mean of all the measurement values during publication period. For aggregated metrics, this value is the mean of all the aggregated metrics means.</td>
</tr>
<tr>
<td>min</td>
<td>Minimum of all the measurement values during publication period. For aggregated metrics, this value is the minimum of all the aggregated metrics minimums.</td>
</tr>
<tr>
<td>max</td>
<td>Maximum of all the measurement values during publication period. For aggregated metrics, this value is the maximum of all the aggregated metrics minimums.</td>
</tr>
<tr>
<td>std_dev</td>
<td>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.</td>
</tr>
</tbody>
</table>

Host Metrics

Listing 10.4: Host Types

```c
@appendable @nested
class HostPeriodic {
  @optional StatisticVariable cpu_usage_percentage;
  @optional StatisticVariable free_memory_kb;
  @optional StatisticVariable free_swap_memory_kb;
  int32 uptime_sec;
};
```
### HostConfig

```rust
struct HostConfig {
    BoundedString name;
    uint32 id;
    int64 total_memory_kb;
    int64 total_swap_memory_kb;
    BoundedString target;
};
```

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the host where the service is running.</td>
</tr>
<tr>
<td>id</td>
<td>ID of the host where the service is running.</td>
</tr>
<tr>
<td>total_memory_kb</td>
<td>Total memory in KiloBytes of the host where the service is running.</td>
</tr>
<tr>
<td>total_swap_memory_kb</td>
<td>Total swap memory in KiloBytes of the host where the service is running.</td>
</tr>
</tbody>
</table>

### HostPeriodic

```rust
struct ProcessConfig {
    uint64 id;
};
```

```rust
struct ProcessPeriodic {
    @optional StatisticVariable cpu_usage_percentage;
    @optional StatisticVariable physical_memory_kb;
    @optional StatisticVariable total_memory_kb;
    int32 uptime_sec;
};
```

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_usage_percentage</td>
<td>Statistic variable that provides the global percentage of CPU usage on the</td>
</tr>
<tr>
<td></td>
<td>host where the service is running.</td>
</tr>
<tr>
<td>free_memory_kb</td>
<td>Statistic variable that provides the amount of free memory in KiloBytes of</td>
</tr>
<tr>
<td>free_swap_memory_kb</td>
<td>the host where the service is running.</td>
</tr>
<tr>
<td>uptime_sec</td>
<td>Time in seconds elapsed since the host on which the running service started.</td>
</tr>
</tbody>
</table>

### Process Metrics

#### Listing 10.5: Process Types

```rust
@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 10.12: ProcessConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Identifies the process where the service is running. The meaning of this value is platform dependent.</td>
</tr>
</tbody>
</table>

Table 10.13: ProcessPeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_usage_percentage</td>
<td>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.</td>
</tr>
<tr>
<td>physical_memory_kb</td>
<td>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.</td>
</tr>
<tr>
<td>total_memory_kb</td>
<td>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.</td>
</tr>
<tr>
<td>uptime_sec</td>
<td>Time in seconds elapsed since the running service process started. Availability of this value is platform dependent.</td>
</tr>
</tbody>
</table>

Base Entity Resource Metrics

Listing 10.6: Base Entity Types

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

Table 10.14: EntityConfig

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource_id</td>
<td>String representation of the resource identifier associated with the entity resource.</td>
</tr>
<tr>
<td>configuration</td>
<td>String representation of the XML configuration of the entity resource. The XML contains only children elements that are not entity resources.</td>
</tr>
</tbody>
</table>
Table 10.15: EntityEvent

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>State of the resource entity expressed as an enumeration of type <code>EntityStateKind</code>.</td>
</tr>
</tbody>
</table>

Network Performance Metrics

Listing 10.7: Network Performance Type

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

Table 10.16: NetworkPerformance

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>samples_per_sec</td>
<td>Statistic variable that provides information about the number of samples processed (received or sent) per second.</td>
</tr>
<tr>
<td>bytes_per_sec</td>
<td>Statistic variable that provides information about the number of bytes processed (received or sent) per second.</td>
</tr>
<tr>
<td>latency_millisec</td>
<td>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.</td>
</tr>
</tbody>
</table>

Thread Metrics

Listing 10.8: Thread Metrics Type

```cpp
@mutable @nested
struct ThreadPeriodic {
    uint64 id;
    @optional StatisticVariable cpu_usage_percentage;
};

@mutable @nested
struct ThreadPoolPeriodic {
    @optional sequence<Service::Monitoring::ThreadPeriodic> threads;
};
```
Table 10.17: ThreadPeriodic

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>OS-assigned thread identifier</td>
</tr>
<tr>
<td>cpu_usage_percentage</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 10.5 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 10.7 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 10.7: Plugin object management](image)

Figure 10.7 also shows that there are two mechanisms through which a Service obtains a plugin object: a shared library or the Service API. Both mechanisms are complementary and are described with more detail in the next sections.
10.5.1 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 (Usage).

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 10.18 and Table 10.19 describe the configuration of each different plugin language.
### Table 10.18: Configuration tags for C/C++ plugins.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;class&gt;_plugin</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
</table>
| `<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:**
```xml
<property>
  <value>
    <element>
      <name>myplugin.user_name</name>
    </element>
    <value>myusername</value>
  </value>
</property>
``` | 0..1 |

### Table 10.19: Configuration tags for Java plugins

<table>
<thead>
<tr>
<th>Tags within <code>&lt;java_[class]_plugin&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;class_name&gt;</code></td>
<td>Name of the class that implements the plugin. For example: <code>com.myplugins.CustomPlugin</code> The classpath required to run the Java plugin must be part of the RTI service JVM configuration. See the <code>&lt;jvm&gt;</code> tag within the specific service configuration for additional information on JVM creation and configuration.</td>
<td>1</td>
</tr>
</tbody>
</table>

10.5. Plugin Management
## 10.5.2 Service API

The user provides the plugin object via the Service API, through one of the available `attach_[class]_plugin()` operations. Upon successful return of the operation, the `Service` takes ownership of the plugin object and will delete it on `Service` stop.

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.

### Table 10.19 – continued from previous page

<table>
<thead>
<tr>
<th>Tags within <code>&lt;java_[class]_plugin&gt;</code></th>
<th>Description</th>
<th>Multiplicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;property&gt;</code></td>
<td>A sequence of name-value string pairs that allow you to configure the plugin instance. <strong>Example:</strong></td>
<td>0..1</td>
</tr>
</tbody>
</table>

```xml
<property>
  <value>
    <element>
      <name>myplugin.user_name</name>
      <value>myusername</value>
    </element>
  </value>
</property>
```
Chapter 11

Release Notes

11.1 Supported Platforms

RTI® Recording Service is supported on the platforms listed in Table 11.1.

It can also be deployed as a C++ library linked into your application. This is true for all platforms in Table 11.1.

Table 11.1: Supported Platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux®</td>
<td>All Linux platforms in the RTI Connext DDS Core Libraries Release Notes with the same version number, except NI Linux 3 on Arm® v7.</td>
</tr>
<tr>
<td>macOS®</td>
<td>All macOS platforms in the RTI Connext DDS Core Libraries Release Notes with the same version number.</td>
</tr>
<tr>
<td>QNX®</td>
<td>All QNX Neutrino® 7.0.4 platforms in the RTI Connext DDS Core Libraries Release Notes with the same version number.</td>
</tr>
<tr>
<td>Windows®</td>
<td>All Windows platforms in the RTI Connext DDS Core Libraries Release Notes with the same version number.</td>
</tr>
</tbody>
</table>

Note: POSIX-compliant architectures that end with “FACE_GP” are not supported.

Recording Service is also supported on the platforms in Table 11.2; these are target platforms for which RTI offers custom support. If you are interested in these platforms, please contact your local RTI representative or email sales@rti.com.

Table 11.2: Custom Platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Yocto Project® 2.5 on Arm® v8</td>
</tr>
</tbody>
</table>
11.2 Compatibility

For backward compatibility information between Recording Service 6.1.0 and previous releases, please see the Migration Guide on the RTI Community portal.

11.3 What’s New in 6.1.0

11.3.1 New platforms

- macOS 10.15 (x64)
- Red Hat® Enterprise Linux 7.6 (x64)
- Ubuntu® 18.04 LTS (Arm v7 and v8)
- Ubuntu 20.04 LTS (x64)
- Yocto Project 2.5 (Arm v8) (custom-supported target platform)

For more information on these platforms, see the RTI Connext DDS Core Libraries Platform Notes for this release.

11.3.2 Removed platforms

These platforms are no longer supported:

- macOS 10.12
- SUSE Linux Enterprise Server 11
- Ubuntu 12.04 LTS

11.3.3 New message showing location for output recorded via RTI Launcher

When using RTI Launcher to start Recording Service, it was not clear where to find the resulting recorded database. Now you will see a message that indicates the output directory, if you set the verbosity to level 3 or higher.

11.3.4 Ability to replay by recorded source timestamp

Now you can replay recorded databases using the recorded source timestamp. For details, see Choosing the Sample Order for Replaying Data.
11.3.5 Instance History Replay

Replay Service can now recreate the *state of the world* prior to a given start timestamp. When this new feature is enabled, Replay Service will publish the latest value known for every instance alive, for keyed topics.

This functionality is currently implemented for the builtin SQLite storage. The Storage API has also been extended to support external plugins.

In the builtin SQLite storage, Recording Service builds an internal index for increased performance. The index is built only once, either:

- At recording time
- Offline using the utility rtirecordingindexer
- At replay runtime

The offline and replay runtime modes are applicable only if the index is not already built.

11.3.6 Ability to index SQLite tables offline

This release includes a new command-line utility that will index a database. There is a table indexing mode that will create two SQLite indexes per table, one based on the reception timestamp column and another based on the source timestamp column.

The new script is called rtirecordingindexer.

11.3.7 New debug mode for Replay Service

Replay Service can now be started in debug mode. In this mode, you can set breakpoints in the replay. Debug mode allows you to manage the replay on a deeper level than the standard replay mode. See *Using Debug Mode while Replaying* for details.

To enable debug mode, use the command line option `-debugMode` or the `<debug_mode>` tag, like this:

```xml
<playback>
  <debug_mode>
    <enabled>true</enabled>
  </debug_mode>
</playback>
```

11.3.8 Ability to change speed and jump in time during replay

Remote Administration gives you more control while replaying data. Now you can change the replay speed, or jump forward/backward in time, without having to restart the replay.

For more information, see *Jumping in Time while Replaying* and *Remote API Overview*. 

11.3. What’s New in 6.1.0
11.3.9 Improved automatic recognition of recorded databases by SQLite tools

Recording Service now uses the standard .db file extension by default when creating database files. This change will improve the automatic recognition of these files as SQLite files.

11.3.10 New XML element to wait for acknowledgment

When replaying a large database file, Replay Service may shut down before a subscribing application receives all the samples. This can be solved using the new tag <on_delete_wait_for_ack_timeout>.

11.3.11 Repetitive and misleading message repeated at end of recording (rollover not enabled)

When Recording Service was used with filesets enabled but no rollover, if the end of the recording was reached (meaning all files in the fileset were completely full with data), a message was continuously printed:

Although this message was benign, it could be misleading. There were no actual errors, but the file size limit was reached, so it was no longer possible to write any more data.

This issue has been resolved. Recording Service will now print just one warning that the end of the fileset has been reached. No recording will happen after that, as expected.

11.3.12 Extended -verbosity command-line parameter to specify service and DDS verbosity independently

Recording Service’s -verbosity command-line parameter now allows you to specify separate verbosity levels for service and DDS logs.

For example, you can specify the verbosity as follows:

```
-verbosity WARN:EXCEPTION
```

This will set the verbosity to warning level for service logs and exception level for DDS logs.

For more information about -verbosity, see Recording Service Command-Line Parameters or run rtirecordingservice -help.

11.3.13 Service monitoring now reports CPU usage for individual Session threads

Service monitoring now includes CPU usage for each thread in a Session’s thread pool. A new optional member has been added to the SessionPeriodic type that includes a sequence of CPU usage for each thread in Session’s thread pool.
11.3.14 Added a Service constructor that receives a shutdown hook

*Recording Service* now has the ability to notify the user when a request to terminate the process is generated, either from remote administration or if the replay completes. The notification is provided through a shutdown hook that can be passed to the service constructor.

See *rti::recording::Service* for more information.

11.3.15 New API to execute administration commands directly on a Service

The *rti::recording::Service* class has a new operation that allows you to execute administration commands as direct calls on the service. The operation has the following signature:

```cpp
CommandReply execute_command(const CommandRequest & request);
```

For details, see *Remote Administration*.

11.4 What’s Fixed in 6.1.0

11.4.1 Non-zero throughput monitoring metrics were provided while service was paused

*Recording Service* provided non-zero throughput monitoring metrics while the service was paused. It provided the number of samples being received by the underlying DataReader. This problem has been resolved. Now when the service is paused, the reported throughput value will be zero.

[RTI Issue ID RECORD-1018]

11.4.2 Executable did not log build ID for DDS libraries

The executable version of Recording Service did not log the build ID of DDS libraries, no matter which verbosity level was specified. This was a regression from the previous version, where the build ID was logged for the warning verbosity. This problem has been resolved.

[RTI Issue ID RECORD-1024]

11.4.3 XML Converter, rtixmlconverter, did not show its version properly

The XML Converter tool, *rtixmlconverter*, reported this error if you tried to see its version with *rtixmlconverter -version*:

```
main:!parse_before_common:!parse 'must specify conversion of record or replay...
xml': argument missing
```
The version was only displayed if you also passed other parameters (record/replay and a configuration filename). This problem has been resolved. Now you can enter `rtixmlconverter -version` to see the version number.

[RTI Issue ID RECORD-1043]

### 11.4.4 XMLConverter threw “multiple deserialization modes” warning in some cases

XMLConverter (`rtixmlconverter`) threw the following warning when converting a Recording Service 5.3.1 configuration file:

```
Warning: Recording Service does not currently support multiple deserialization modes in a single recording.
```

This problem has been resolved.

[RTI Issue ID RECORD-1056]

### 11.4.5 Segmentation fault when using non-compliant XML configuration on a QNX system

Using a non-compliant XML configuration with the Recording Service XSD file caused a segmentation fault when trying to process that configuration. This issue, which only occurred on QNX systems, has been resolved.

[RTI Issue ID RECORD-1122]

### 11.4.6 Segmentation fault after discovering application that used UserData QoS Policy

If Recording Service discovered an application that had at least one entity (DomainParticipant, DataReader, or DataWriter) that used the UserData QoS Policy, the behavior was undefined and would likely have resulted in a segmentation fault. This problem has been resolved.

[RTI Issue ID RECORD-1125]

### 11.4.7 Replay Service did not detect termination correctly

Replay Service failed to detect termination correctly under certain conditions. This problem has been resolved.

[RTI Issue ID RECORD-1129]
11.4.8 Windows Debug assertions thrown when running in debug mode

Windows debug assertions were sometimes thrown by the debug version of the Replay and Converter executables. This prevented users from running these services successfully on Windows systems when debugging issues. This problem has been resolved.

[RTI Issue ID RECORD-1136]

11.4.9 Monitoring Config Topic contained incomplete information about Topics created from TopicGroups

Samples sent on the Config monitoring topic did not include information for recording Topic entities that were created from TopicGroups. The missing information included:

- topic_name
- registered_type_name
- participant_name
- topic_group resource ID

This problem has been resolved so the missing information is provided.

[RTI Issue ID RECORD-1146]

11.4.10 Crash if monitoring enabled and TopicGroup matched with Topic name containing '/' character

Recording Service crashed if monitoring was enabled and a TopicGroup matched with a Topic name that contained one or more forward slash ('/') characters. This problem has been resolved.

[RTI Issue ID RECORD-1153]

11.4.11 Replay Monitoring reported increasing current timestamp while service was paused

The current timestamp value provided as part of ServicePeriodic::builtin_sqlite was always increasing, even while the service was paused. However in Replay mode, this value is expected to remain constant while the service is paused. This problem has been resolved.

The following changes in IDL were required:

- Deprecated the fields current_timestamp_sec and current_timestamp_nanosec in SqliteDatabasePeriodic. These values are present but will be always set to zero.

- Added a new field, current_timestamp_nanos under ServicePeriodic, that represents that current timestamp for both Recording and Replay modes, regardless of the storage being used.

[RTI Issue ID RECORD-1170]
11.4.12 RTI Converter did not properly convert Sequence Numbers from Sample-Info field

RTI Converter did not properly process sequence numbers from the SampleInfo field. This caused those fields to always be set to -1 or UINT32_MAX (0xFFFFFFFF). This problem has been resolved.

[RTI Issue ID RECORD-1228]

11.4.13 Recording/Replay Service Crashed when Receiving Wrong Remote Admin Request for Updating Service State

Both Recording Service and Replay Service may have crashed when receiving an UPDATE remote administration request to update the service state (e.g., using resource ID /recording_services/<service name>/state or /replay_services/<servicename>/state).

If the octet body field of the request was not filled in properly (by serializing the CDR representation of the desired state), and instead contained random data or the serialization of a different type of sample, Recording Service and Replay Service could crash.

This problem has been resolved.

[RTI Issue ID RECORD-1249]

11.5 Known Issues

11.5.1 Recording Service may fail when current working directory in c:\Program Files

Recording Service will try to write to its database in the current working directory. If it does not have permissions to write there, it will fail with a confusing error:

```bash
[/recording_services/RecorderService|START|/storage|CREATE]
create_connection:!SQLite - failed to open database file; sqlite returned error: out of memory
[SQLite return code = 7]
[File = metadata]
```

This happens most commonly when running the application from within \Program Files on Windows systems. You can work around this by running Recording Service from a command prompt in a directory where you have write permissions.
Chapter 12

Copyrights

© 2021 Real-Time Innovations, Inc. All rights reserved. Printed in U.S.A. First printing. April 2021.

Trademarks

RTI, Real-Time Innovations, Connext, NDDS, the RTI logo, IRTI and the phrase, “Your Systems. Working as one.” are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished under and subject to the RTI software license agreement. The software may be used or copied only under the terms of the license agreement.

This is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.

The security features of this product include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

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/

© 2021 RTI
Index

C
Configuration name, 156
Configuration variable, 156

S
Service API, 156
Shipped executable, 156

X
XML document, 156