RTI Recording Service

User's Manual

Version 6.0.1

Your systems.
Working as one.

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 . . . 6
3.1.3 Recording Service Command-Line Parameters . . . 6
3.1.4 Controlling Recording Service's Operation Mode . . . 8
Controlling Buffer Size in Buffering Mode . . . 8
Configuring In-Memory Buffer Size . . . 9
3.2 Configuration . . . 10
3.2.1 How to Load the XML Configuration . . . 10
3.2.2 XML Syntax and Validation . . . 11
3.2.3 Builtin Configuration of Recording Service . . . 12
3.2.4 XML Tags for Configuring Recording Service . . . 12
3.2.5 Recording Service Tag . . . 13
Example: Specify a Recording Service Configuration in XML . . . 15
3.2.6 Administration . . . 16
3.2.7 Monitoring . . . 17
3.2.8 Storage . . . 17
SQLite . . . 18
Plugin . . . 24
3.2.9 DomainParticipant . . . 24
3.2.10 Session . . . 26
3.2.11 Topic Group . . . 28
3.2.12 Topic . . . 29
3.2.13 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
3.2.14 Plugins . . . 31
3.2.15 Enabling Distributed Logger . . . 31
3.2.16 Recording Service Builtin Configuration Details . . . 32

3.3 Remote Administration . . . 33
3.3.1 Enabling Remote Administration . . . 33
3.3.2 Available Service Resources . . . 33
Example . . . 34
3.3.3 Remote API Overview . . . 35
3.3.4 Recording Service . . . 35
3.3.5 Storage . . . 36
3.4 Monitoring . . . 37
3.4.1 Overview . . . 37
Enabling Service Monitoring . . . 37
Monitoring Types . . . 38
3.4.2 Monitoring Metrics Reference . . . 38
Service . . . 38
Session . . . 41
TopicGroup . . . 42
Topic . . . 43
3.5 Tutorials . . . 44
3.5.1 Getting Started with Recording Service and Shapes Demo . . . 44
Edit the Configuration . . . 44
Start Shapes Demo . . . 45
Start Recording Service . . . 45
View the Data in Sqlite3 . . . 47
3.5.2 Using Recording Service and Admin Console . . . 47
Configuration . . . 47
Start Recording Service . . . 48
Start Shapes Demo . . . 48
Viewing with Admin Console . . . 48
Administering with Admin Console . . . 52
3.5.3 Using Recording Service as a Library . . . 52
Include files . . . 52
Using the RecordingService class . . . 53
3.5.4 Plugging in Custom Storage . . . 53
Custom Storage API Overview . . . 54
3.5.5 Accessing JSON samples through SQL . . . 55
3.5.6 Controlling Recording Service Remotely from an Application . . . 55
3.5.7 Listing the Timestamp Tags in a Recording . . . 56
3.6 Troubleshooting . . . 56
3.6.1 Verbosity . . . 56

4 Replay Service **58**
4.1 Usage . . . 58
4.1.1 Starting Replay Service . . . 58
4.1.2 Stopping Replay Service . . . 58
4.1.3 Replay Service Command-Line Parameters . . . 58
4.1.4 Replay Service Runtime Behavior . . . 60
4.1.5 Working With Large Data . . . 60
4.2 Configuration . . . 60
4.2.1 How to Load the XML Configuration . . . 61

4.2.2 XML Syntax and Validation . . . 62
4.2.3 XML Tags for Configuring Replay Service . . . 63
4.2.4 Replay Service Tag . . . 64
Example: Specify a Replay Service Configuration in XML . . . 66
4.2.5 Administration . . . 66
4.2.6 Monitoring . . . 69
4.2.7 Storage . . . 70
SQLite . . . 70
Plugin . . . 72
4.2.8 Legacy . . . 72
4.2.9 Domain Mapping . . . 72
4.2.10 DomainParticipant . . . 73
4.2.11 Playback . . . 75
4.2.12 Data Selection . . . 76
4.2.13 Time Range . . . 77
4.2.14 Session . . . 77
4.2.15 Topic Group . . . 79
4.2.16 Topic . . . 79
4.2.17 Plugins . . . 81
4.3 Remote Administration . . . 81
4.3.1 Enabling Remote Administration . . . 81
4.3.2 Available Service Resources . . . 81
Example . . . 82
4.3.3 Remote API Overview . . . 82
4.3.4 Replay Service . . . 82
4.4 Monitoring . . . 83
4.4.1 Overview . . . 83
Enabling Service Monitoring . . . 83
Monitoring Types . . . 83
4.4.2 Monitoring Metrics Reference . . . 84
Service . . . 84
Session . . . 87
TopicGroup . . . 88
Topic . . . 89
4.5 Tutorials . . . 90
4.5.1 Example: Getting Started with Replay and Shapes Demo . . . 90
Edit the Replay Configuration . . . 90
Start Shapes Demo and Subscribe to Squares . . . 91
Start Replay Service . . . 92
4.5.2 Example: Replaying Data at a Different Rate . . . 92
Edit the Replay Configuration . . . 92
Start Shapes Demo . . . 93
Start Replay Service . . . 93
4.5.3 Example: Plugging in Custom Storage . . . 94
Custom Storage API Overview . . . 94
4.5.4 Using Timestamp Tags with Replay Service . . . 96
4.6 Troubleshooting . . . 96
4.6.1 No Input File . . . 96

4.6.2 Table Not Found Errors . . . 97

5 Converter **98**
5.1 Usage . . . 98
5.1.1 Starting Converter . . . 98
5.1.2 Converter Command-Line Parameters . . . 98
5.1.3 Working With Large Data . . . 99
5.2 Converter Configuration . . . 99
5.2.1 How to Load the XML Configuration . . . 100
5.2.2 XML Syntax and Validation . . . 101
5.2.3 Builtin Configuration of Converter . . . 101
5.2.4 XML Tags for Configuring Converter . . . 102
5.2.5 Converter Tag . . . 103
Example: Specify a Configuration in XML . . . 104
5.2.6 Input Storage . . . 104
5.2.7 Output Storage . . . 106
5.2.8 SQLite . . . 106
5.2.9 Fileset . . . 108
5.2.10 Rollover . . . 109
5.2.11 Legacy . . . 110
5.2.12 Domain Mapping . . . 110
5.2.13 Plugin . . . 111
5.2.14 Data Selection . . . 111
5.2.15 Time Range . . . 111
5.2.16 DomainParticipant . . . 112
5.2.17 Session . . . 113
5.2.18 Topic Group . . . 114
5.2.19 Topic . . . 114
5.2.20 Converter's Builtin Configuration Details . . . 115
5.3 Tutorials . . . 116
5.3.1 Using Timestamp Tags with Converter . . . 116
5.4 Troubleshooting . . . 117
5.4.1 Table Not Found Errors . . . 117

6 Software Development Kit **119**

7 Common Infrastructure **121**
7.1 Application Resource Model . . . 121
7.1.1 Example: Simple Resource Model of a Connext DDS Application . . . 122
7.1.2 Resource Identifiers . . . 122
Escaped Identifiers . . . 123
Example: Resource Identifiers of a Generic Connext DDS Application . . . 124
Example: Resource Identifiers Generated from XML Entity Model . . . 124
7.2 Remote Administration Platform . . . 124
7.2.1 Remote Interface . . . 125
Standard Methods . . . 126
Custom Methods . . . 126
7.2.2 Communication . . . 127

Reply Sequence . . . 129
Example: Accessing from Connext DDS Application . . . 130
7.2.3 Common Operations . . . 131
Create Resource . . . 131
Get Resource . . . 132
Update Resource . . . 132
Set Resource State . . . 133
Delete Resource . . . 134
7.3 Monitoring Distribution Platform . . . 134
7.3.1 Distribution Topic Definition . . . 135
Example: Monitoring of Generic Application . . . 136
7.3.2 DDS Entities . . . 138
7.3.3 Monitoring Metrics Publication . . . 138
Configuration Distribution Topic . . . 138
Event Distribution Topic . . . 138
Periodic Distribution Topic . . . 138
7.3.4 Monitoring Metrics Reference . . . 139
Statistic Variable . . . 139
Host Metrics . . . 140
Process Metrics . . . 141
Base Entity Resource Metrics . . . 142
Network Performance Metrics . . . 143
7.4 Plugin Management . . . 144
7.4.1 Shared Library . . . 144
Configuration . . . 145
7.4.2 Service API . . . 147

8 Release Notes 148
8.1 Supported Platforms . . . 148
8.2 Compatibility . . . 149
8.3 What's New in 6.0.1 . . . 149
8.3.1 New platforms . . . 149
8.3.2 Removed platforms . . . 149
8.3.3 New <topic_name> tag to specify DDS topic name . . . 149
8.3.4 Replay fidelity improved with large data . . . 150
8.3.5 Improved performance when creating legacy database stream readers . . . 150
8.3.6 Support for remotely triggered flush of user-data to disk . . . 150
8.4 What's Fixed in 6.0.1 . . . 150
8.4.1 Legacy deserialized database table couldn't be replayed or converted if it contained compact byte sequences or arrays . . . 150
8.4.2 Segmentation fault in Replay Service or Converter if recorded topics had "::" in their names . . . 151
8.4.3 Detection of termination in Converter fixed . . . 151
8.4.4 QoS Topic Filters were not supported . . . 151
8.4.5 Execution directories overwritten incorrectly . . . 152
8.4.6 Segmentation fault in Recording Service if <time_limit> didn't have all its child properties . . . 153

8.4.7 Possible race condition in Replay Service when looping enabled and pause/resume issued . . . 153
8.4.8 Possible segmentation fault when replaying or converting legacy database with compressed octet/char sequences . . . 153
8.4.9 Missing shutdown messages . . . 154
8.4.10 XML attribute values were overwritten if element set with variable . . . 154
8.4.11 Segmentation fault when no valid configuration file found . . . 154
8.4.12 Possible segmentation fault when converting legacy database to CDR_AUTO format . . . 155
8.5 Previous Releases . . . 155
8.5.1 What's New in 6.0.0 . . . 155
Avoiding gaps in Replay when using time ranges . . . 155
8.5.2 What's Fixed in 6.0.0 . . . 155
Pluggable storage for Converter discarded plugin properties specified in XML 155
8.6 Known Issues . . . 156
8.6.1 Recording Service may fail when current working directory in c:\Program Files 156
8.6.2 Segmentation fault after discovering application that uses UserData QoS Policy 156

9 Copyrights 157

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 builtin 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.

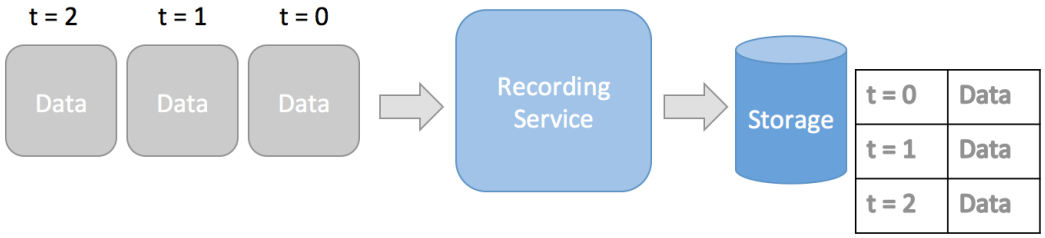


Figure 1.1: *Recording Service* receives DDS samples over the network and records them to storage, based on their timestamps

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.

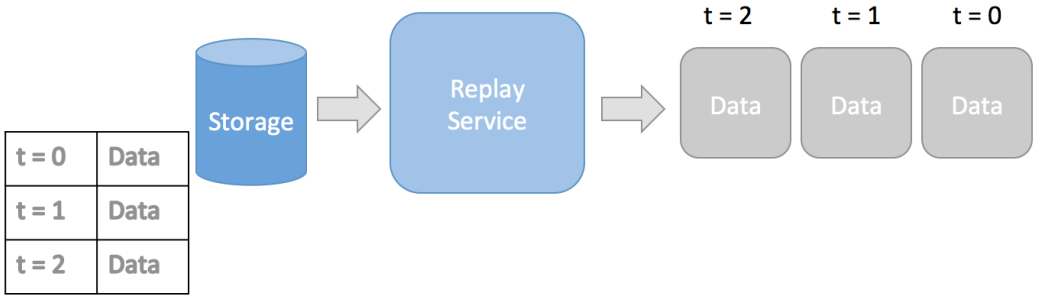


Figure 1.2: *Replay Service* retrieves stored data samples from storage and replays it over DDS, maintaining the original timing of the data

Converter is a tool that can convert between the data formats used by *Recording Service*. Currently, the only builtin 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`
  - UNIX®-based systems, non-root user: `/home/your user name/rti_connext_dds-version`

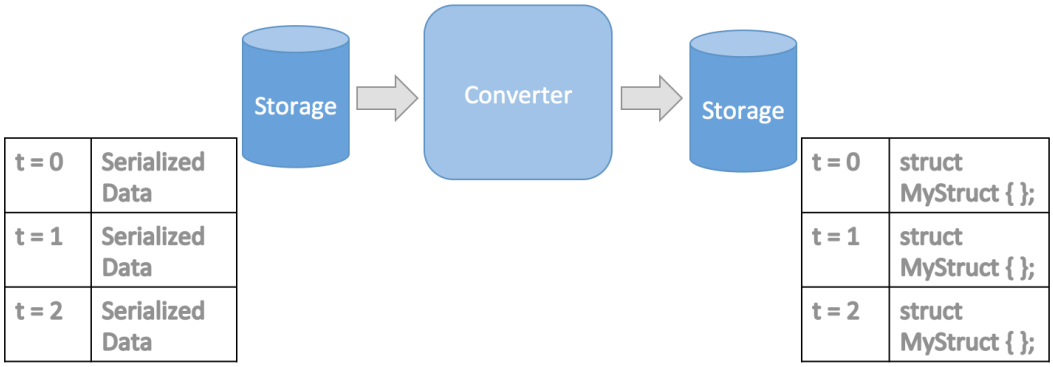


Figure 1.3: *Converter* takes serialized (binary) data and converts it to deserialized JSON data

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

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

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

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

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

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

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

  Default path to the examples:

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

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)

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.

Note: *Recording Service* is pre-loaded with a builtin configuration that has default settings. See Section 3.2.16.

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 3.2.4 for the XML elements that can be overridden with command-line parameters.

Table 3.1: *Recording Service* Command-Line Parameters

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

Continued on next page

Table 3.1 – continued from previous page

Parameter	Description
-remoteMonitoringDomainId <int>	Enables remote monitoring and sets the domain ID for status publication. **Default**: Remote monitoring is not enabled.
-verbosity [0-6]	**RTI Service verbosity** • 0 - silent • 1 - exceptions (*Connext DDS* and *Recording Service*) • 2 - warnings (*Recording Service*) • 3 - information (*Recording Service*) • 4 - warnings (*Connext DDS* and *Recording Service*) • 5 - tracing (*Recording Service*) • 6 - tracing (*Connext DDS* and *Recording Service*) **Default**: 1 (exceptions)
-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 Section 3.2.10 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.

You can also define a flush period (see section Section 3.2.8 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 Section 3.2.8). 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 3.2: *Recording Service* Buffer Size Control

QoS Policy	Value
`History.kind`	KEEP_LAST
`History.depth`	N
(Optional) `ResourceLimits.max_samples`	N
(Optional) `ResourceLimits.max_instances`	N
(Optional) `ResourceLimits.max_samples_per_instance`	N
(Optional) `ResourceLimits.initial_samples`	N
(Optional) `ResourceLimits.initial_instances`	N

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.

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 Configuration

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

3.2.1 How to Load the XML Configuration

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

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

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

`[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 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="MyQosLib">
        <qos_profile name="ReliableQos" >
            <datareader_qos>
                <reliability>
                    <kind>RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
            </datareader_qos>
        </qos_profile>
    </qos_library>
    <!--  A definition of a Recording Service instance to run -->
    <recording_service name="MyRecorderService">
```

(continues on next page)

(continued from previous page)

```
        <!--  Storage settings -->
        <storage>
            <sqlite>
                <storage_format>XCDR_AUTO</storage_format>
                <file>rti_recorder_default</file>
                <file_suffix>dat</file_suffix>
            </sqlite>
        </storage>
        <!--  Top-level domain settings -->
        <domain_participant name="Participant0">
            <domain_id>0</domain_id>
        </domain_participant>
        <!-- Recording Session -->
        <session name="DefaultSession" >
            <!-- Group of topics to record -->
            <topic_group name="RecordAll" participant_ref="Participant0">
                <allow_topic_name_filter>*</allow_topic_name_filter>
                <allow_type_name_filter>*</allow_type_name_filter>
                <!-- DDS input -->
                <datareader_qos base_name="MyQosLib::ReliableQos" />
            </topic_group>
        </session>
    </recording_service>
</dds>
```

3.2.2 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, *Recording Service* relies on an XSD file that describes the format of the XML content. We recommend including a reference to this document in the XML file that contains the service's configuration; this 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/resource/schema/rti_recording_service.xsd`.

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

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

(continues on next page)

(continued from previous page)

```
   <!-- ... -->
</dds>
```

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

3.2.3 Builtin Configuration of Recording Service

Recording Service is pre-configured with a builtin configuration. See Section 3.2.16 for more details.

3.2.4 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.

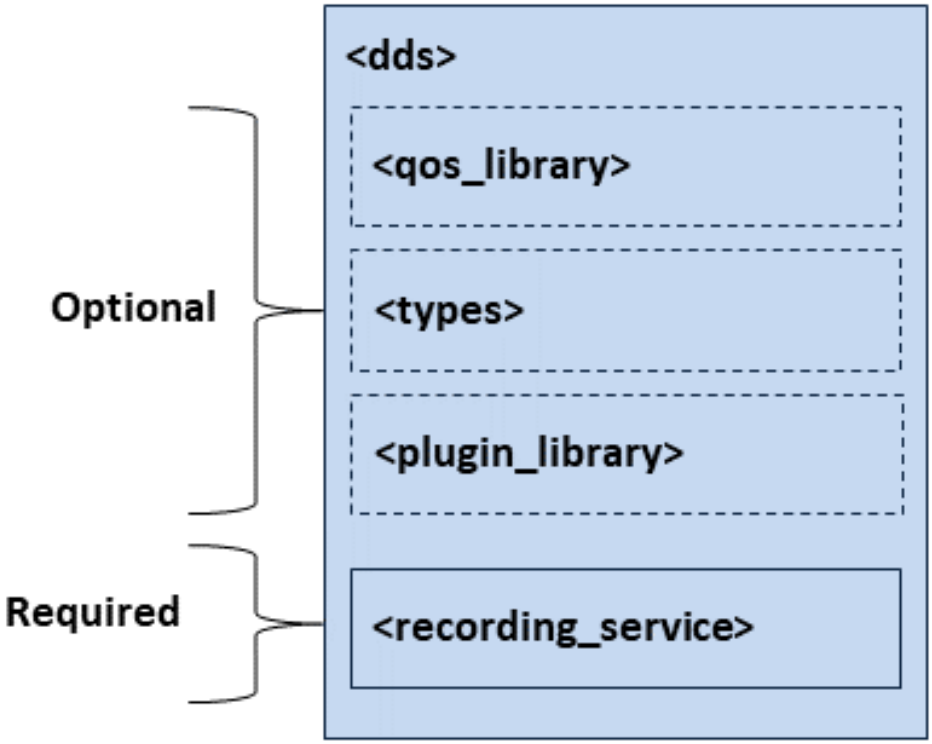


Figure 3.1: Top-level Tags in Recording Service's Configuration File

Table 3.3: Top-level Tags in Recording Service's Configuration File

Tags within <dds>	Description	Multiplicity
<qos_library>	Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for the *Connext DDS* QoS profile file—see Configuring QoS with XML in the RTI Connext DDS Core Libraries User's Manual.	0..*
<types>	Defines types that can be used by *Recording Service*. This 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.	0..*
<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 3.5.4. • Transform data after it is received from *Connext DDS* and before it is stored. For more on using transformations, see Data Transformation, in the RTI Routing Service User's Manual.. See Section 3.2.14	0..*

<recording_service>	**Required.** Specifies a *Recording Service* configuration. See Section 3.2.5.
	Attributes
	• `name`: Uniquely identifies a service configuration. **Required**.
	Example
	`<recording_service name="RecordAll">`
	`<!-- your service settings ... -->`
	`</recording_service>`
		1..*

3.2.5 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.

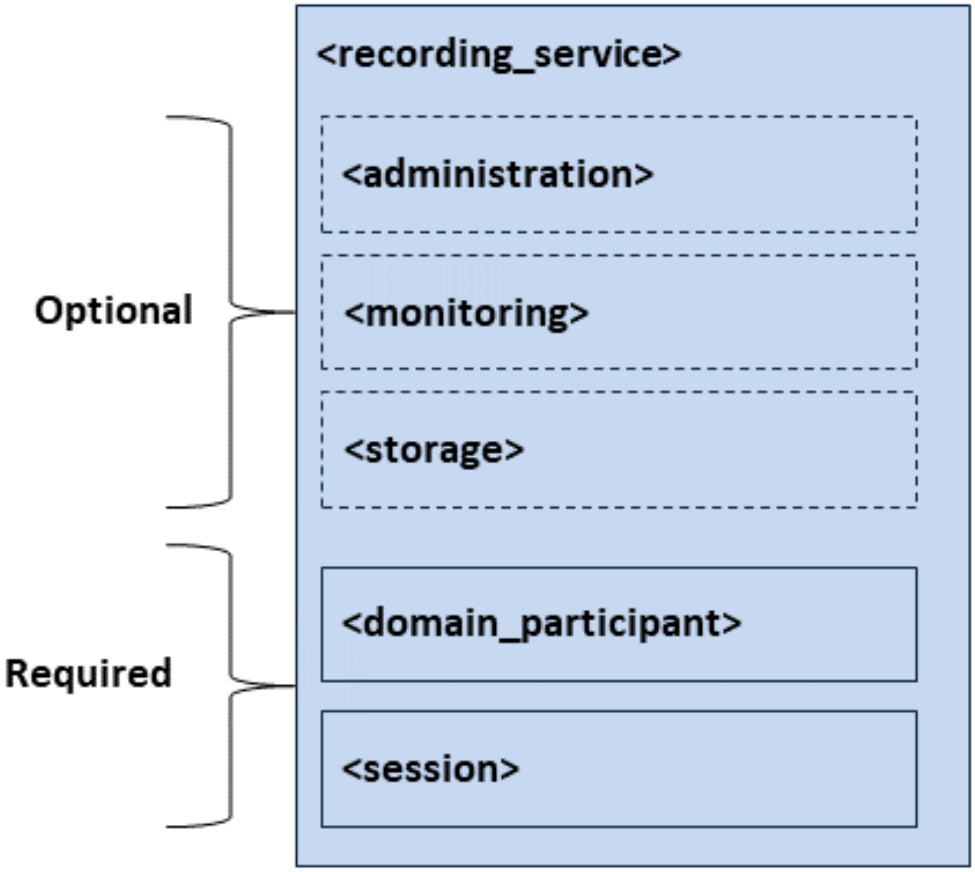


Figure 3.2: Tags used to configure a *Recording Service* instance

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

Tags within <recording_service>	Description	Multiplicity
<administration>	Enables remote administration. When administration is enabled, monitoring is also enabled by default. If no domain ID is specified for monitoring, *Recording Service* will use the same domain as administration by default. See Section 3.2.6.	0..1
<monitoring>	Enables monitoring for the recording service, including statistics. See Section 3.2.7.	0..1
<storage>	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.dat". See Section 3.2.8.	0..1

<domain_participant>	**Required**. Specifies a DomainParticipant to use to record data.
	Attributes
	• `name`: Uniquely defines a DomainParticipant. **Required**.
	Example
	`<domain_participant name="Participant3">`
	`<domain_id>3</domain_id>`
	`<!-- Participant QoS -->`
	`</domain_participant>`
	See Section 3.2.9.
		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
	`<session name="Session" default_participant_ref= ↪"Participant3">`
	`<!-- ... topics / groups of topics to record -->`
	`</session>`
	See Section 3.2.10.
		1..*

Example: Specify a Recording Service Configuration in 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
```

3.2.6 Administration

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

See Section 3.3 for details on using remote administration.

Table 3.5: Administration Tags in Recording Service's Configuration File

Tags within <administration>	Description	Multiplicity
<domain_id>	Domain ID used for remote administration. Also used for monitoring by default.	0..1
<participant_qos>	QoS used by the administration DomainParticipant. If the tag is not defined, *Connext DDS* defaults will be used.	0..1
<publisher_qos>	QoS used by the administration Publisher. If the tag is not defined, *Connext DDS* defaults will be used.	0..1
<subscriber_qos>	QoS used by the administration Subscriber. If the tag is not defined, *Connext DDS* defaults will be used.	0..1
<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*, *Recording Service* will publish its Log messages to *Connext DDS*. See Section 3.2.15.	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.

3.2.7 Monitoring

The `<monitoring>` tag allows you to enable and configure remote monitoring of *Recording Service*. See Section 3.4.

Table 3.6: Monitoring Tags in Recording Service's Configuration File

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

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

3.2.8 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

Tags within <storage>	Description	Multiplicity
<sqlite>	Enables storing data in a SQLite database file. See Section 3.2.8.	0..1
<plugin>	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 7.4 for details on how the options of how to register plugins). See Section 3.2.8 for more about using this tag. See Section 3.5.4 for a tutorial on plugging in custom storage.	0..1

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 3.8: SQLite Tags in Recording Service's Configuration File

Tags within <sqlite>	Description	Multiplicity
<fileset>	Set of files to write to, and parameters for creating files and directories in that set. See Section 3.2.8.	0..1
<file>	File to write to. Default: rti_recorder_default	0..1
<overwrite_policy_kind>	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	0..1

Continued on next page

Table 3.8 – continued from previous page

Tags within <sqlite>	Description	Multiplicity
<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
<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
<flush_period>	Defined as a duration (seconds and nanoseconds), <flush_period> 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 <enable_buffering_mode> tag. Default: no default. If not set, *Recording Service* will work in a purely reactive way.	0..1

Continued on next page

Table 3.8 – continued from previous page

Tags within <sqlite>	Description	Multiplicity
<enable_buffering_mode>	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 continously for remote *flush()* commands to trigger the storage to disk (see section Section 3.3.5 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 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 <flush_period> 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).	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

Tags within <fileset>	Description	Multiplicity
<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

Continued on next page

Table 3.9 – continued from previous page

Tags within <fileset>	Description	Multiplicity

<execution_dir_expression>	When *Recording Service* starts, it will use this expression to create the directory where output files will be stored. Every time *Recording Service* starts, it will evaluate this expression to decide on its output directory. Stopping and restarting remotely will cause *Recording Service* 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:
	• 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:
	`<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).
		0..1

Continued on next page

Table 3.9 – continued from previous page

Tags within <fileset>	Description	Multiplicity

<filename_expression>	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:
	• Autonumeric. **Format: %auto:M-N%.** 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.
	Example:
	`<filename_expression>`
	`  test_files_%auto:0-2%.dat`
	`</filename_expression>`
	This will create files named test_files_0.dat, test_files_1.dat, and test_files_2.dat. 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.dat.
	• 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 continously).
	• Date and time. **Format: %c%.** Date and time representation, locale-dependent. This parameter uses the strftime() parameter %c.
	Note: 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%.dat (auto-numeric starting at zero, unlimited).
		0..1

 Continued on next page

Table 3.9 – continued from previous page

Tags within <fileset>	Description	Multiplicity
<rollover>	Configuration for rolling over the file after a size or time limit is reached. See Section 3.2.8.	0..1

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.)

```
<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%.dat
                </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

Tags within <rollover>	Description	Multiplicity
<enabled>	Whether *Recording Service* will roll over files when a limit is reached. Default: False.	0..1
<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

Tags within <plugin>	Description	Multiplicity
<property>	Name/value pairs of properties to pass to a storage plugin.	0 or 1

3.2.9 DomainParticipant

Table 3.12: DomainParticipant Tags in Recording Service's Configuration File

Tags within <domain_participant>	Description	Multiplicity
<domain_id>	**Required**. DDS domain ID used for recording.	1
<participant_qos>	QoS used by this DomainParticipant. See Configuring QoS with XML, in the RTI Connext DDS Core Libraries User's Manual.	0..1

<memory_management>	Configures certain aspects of how *Connext DDS* allocates internal memory. The configuration is per DomainParticipant and therefore affects all the contained DDS entities.
	Example:
	`<memory_management>`
	`  <sample_buffer_min_size>`
	`  1024`
	`  </sample_buffer_min_size>`
	`  <sample_buffer_trim_to_size>`
	`  true`
	`  </sample_buffer_trim_to_size>`
	`</memory_management>`
	Tags within this tag:
	• <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.
		0..1

Continued on next page

Table 3.12 – continued from previous page

Tags within <domain_participant>	Description	Multiplicity
<register_type>	Registers a type name and associates it with a type representation. When you define a type in the configuration file, you have to register the type in order to use it in 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..*

3.2.10 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 3.13: Session Tags in Recording Service's Configuration File

Tags within <session>	Description	Multiplicity
<subscriber_qos>	Specifies the QoS of DDS subscribers that will be used by the contained `<topic>` and `<topic_group>`. See Configuring QoS with XML, in the RTI Connext DDS Core Libraries User's Manual.	0..1

Continued on next page

Table 3.13 – continued from previous page

Tags within <session>	Description	Multiplicity

<thread_pool>	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 reactiveness and scalability of the running *Recording Service* when there are multiple Topics and Topic Groups associated with the same session.
	Example:
	`<thread_pool>`
	`  <mask>MASK_DEFAULT</mask>`
	`  <priority>THREAD_PRIORITY_DEFAULT</priority>`
	`  <stack_size>`
	`  THREAD_STACK_SIZE_DEFAULT`
	`  </stack_size>`
	`</thread_pool>`
	Default values:
	• **size**: 1
	• **mask**: MASK_DEFAULT
	• **priority**: THREAD_PRIORITY_DEFAULT
	• **stack_size**: THREAD_STACK_SIZE_DEFAULT
		0..1

<topic>	Specifies an individual Topic to record. **Attributes:** • name: The name of the Topic to record. This name is also used when monitoring and administering each Topic. • participant_ref: A DomainParticipant to use when recording this Topic. If the parent `<session>` specifies a default_participant_ref, this attribute is optional. See Section 3.2.12.	0..*
<topic_group>	Specifies a group of Topics to record. **Attributes:** • name: The name of the Topic group. This name is also used when monitoring and administering each Topic group. • participant_ref: Specifies a DomainParticipant to use when recording this topic group. If the parent `<session>` specifies a default_participant_ref, this attribute is optional. See Section 3.2.11.	0..*

3.2.11 Topic Group

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

Table 3.14: Topic Group Tags in Recording Service's Configuration File

Tags within <topic_group>	Description	Multiplicity

<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:
	`<topic_group name="RecordAll">`
	`  <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 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
<datareader_qos>	The DataReader's QoS to use when recording data.	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 < and > as shown in the example below.
	Example:
	`<topic_group name="RecordAll">`
	`  <content_filter>`
	`  <!-- Data will only be recorded when x is greater than 100 -->`
	`  <expression>x > 100</expression>`
	`  </content_filter>`
	`</topic_group>`
		0..1

3.2.12 Topic

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

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

Tags within <topic>	Description	Multiplicity
<topic_name>	The name of the DDS topic to be recorded. If this tag is not present, the name attribute of the *<topic>* 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.	0..1
<registered_type_name>	**Required**. The name of the data type that will be recorded for this topic.	1

<transformation>	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.
	Example:
	`<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>`
		0..1

Continued on next page

Table 3.15 – continued from previous page

Tags within <topic>	Description	Multiplicity
<datareader_qos>	The DataReader QoS to use when recording this data.	0..1

<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 > 100</expression>`
	`  </content_filter>`
	`</topic_group>`
		0..1

3.2.13 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

```
<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">
            <participant_qos>
                <discovery>
                    <initial_peers>
                        <element>shmem://</element>
```

(continues on next page)

(continued from previous page)

```
                    </initial_peers>
                </discovery>
                <transport_builtin>
                    <mask>SHMEM</mask>
                </transport_builtin>
            </participant_qos>
        </qos_profile>
    </qos_library>

    <recording_service name="FlatDataWithZeroCopy">

        <domain_participant name="RecordDomain">
            <domain_id>0</domain_id>
            <participant_qos base_name="MyQosLib::ShmemOnly"/>
            <register_type name="Point" type_ref="Point"/>
        </domain_participant>

        <session default_participant_ref="RecordDomain">
            <topic name="PointTopic">
                <registered_type_name>Point</registered_type_name>
            </topic>
        </session>
    </recording_service>

</dds>
```

3.2.14 Plugins

All the pluggable components specific to *Recording Service* are configured within the `<plugin_library>` tag. Table 3.16 describes the available tags.

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

Table 3.16: Configuration tags for plug-in libraries

Tags within <plugin_library>	Description	Multiplicity
<storage_plugin>	Specifies a C/C++ *Storage* plug-in. See Table 7.16 and Section 3.2.8.	0..*
<transformation_plugin>	Specifies a C/C++ *Transformation* plug-in. See Table 7.16 and Section 3.2.12.	0..*

3.2.15 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 `<adminstration>`. For example:

```
<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.

3.2.16 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.dat
- In the efficient XCDR format

```
<?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>

    <recording_service name="RecorderService">
        <!--  Top-level storage settings -->
        <storage>
            <sqlite>
```

(continues on next page)

(continued from previous page)

```
                <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>
```

3.3 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 7.2. 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.

3.3.1 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 3.2.6) 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.

3.3.2 Available Service Resources

Table 3.17 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 7.1.2 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">...</recording_service>
```

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 3.17: Resources and Their Identifiers in *Recording Service*

Resource	Resource Identifier
Recording Service	/recording_services/(rs)
Storage	/recording_services/(rs)/storage/(st)

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">...</recording_service>
```

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
```

3.3.3 Remote API Overview

Table 3.18: Remote Interface Overview

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

3.3.4 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.

Request Field	Value
`action`	UPDATE
`resource_identifier`	/recording_services/MyRecorder:flush

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 7.2.3).

Valid requested states:

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

- Example

  To pause an instance of *Recording Service* with the name "MyRecorder":

Request Field	Value
`action`	UPDATE
`resource_identifier`	/recording_services/MyRecorder/state
`octet_body`	`to_cdr_buffer(RTI::Service::EntityStateKind::PAUSED)`

3.3.5 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.

Request Field	Value
`action`	UPDATE
`resource_identifier`	/recording_services/MyRecorder/storage/sqlite:rollover

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:

Request Field	Value
`action`	UPDATE
`resource_identifier`	/recording_services/MyRecorder/storage/sqlite:tag_timestamp

`octet_body`	`std::string tag_name_1("/example/test1/tag1");`
	`std::string tag_description_1("start of test");`
	`RTI::RecordingService::DataTagParams data_tag_arguments;`
	`data_tag_arguments.tag_name(tag_name_1);`
	`data_tag_arguments.tag_description(tag_description_1);`
	`data_tag_arguments.timestamp_offset(-123);`
	`std::vector<char> data_tag_arguments_buffer;`
	
	`dds::topic::topic_type_support<RTI::RecordingService::DataTagParams>`
	`::to_cdr_buffer(`
	`data_tag_arguments_buffer,`
	`data_tag_arguments);`

3.4 Monitoring

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

Note: *Recording Service* monitoring is based on the *Monitoring Distribution Platform* described in Section 7.3. We recommend that you read Section 7.3 before using *Recording Service* monitoring.

3.4.1 Overview

Enabling Service Monitoring

By default, monitoring is disabled in *Recording Service*. To enable monitoring you can use the `<monitoring>` tag (see Section 3.2.5) 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 3.19. The complete type relationship is shown in Figure 3.3.

Table 3.19: *Recording Service Keyed Resources*

Keyed Resource Class	Config	Event	Periodic
Service	`ServiceConfig`	`ServiceEvent`	`ServicePeriodic`
Session	`SessionConfig`	`SessionEvent`	`SessionPeriodic`
TopicGroup	`TopicGroupConfig`	`TopicGroupEvent`	`TopicGroupPeriodic`
Topic	`TopicConfig`	`TopicEvent`	`TopicPeriodic`

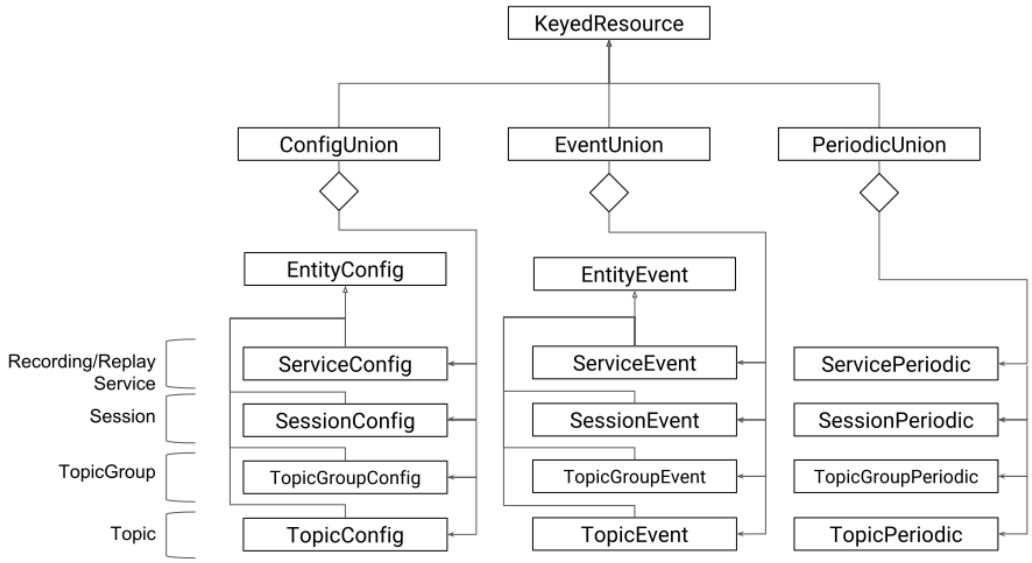


Figure 3.3: Keyed Resource Types for *Recording Service* monitoring

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 3.2.5 for details on configuring the QoS for these entities.

3.4.2 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

```
@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;
    int32 current_timestamp_sec;
    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;
    @optional SqliteDatabasePeriodic builtin_sqlite;
};
```

Table 3.20: `ServiceConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
application_name	Name of the *Recording Service* instance. The application name is provided through: • `appName` command-line option when run as executable. • `ServiceProperty::application_name` field when run as a library.
application_guid	GUID of the *Recording Service* instance. Unique across all service instances.
host	See Table 7.9.
process	See Table 7.11.
builtin_sqlite	See Table 3.21
participants	Sequence of `ParticipantInfo` objects, one for each *DomainParticipant* inside the *Recording Service*. See Table 3.22.

Table 3.21: `SqliteDatabaseConfig`

Field Name	Description
db_directory	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.
execution_directory_expression	The expression used to generate the directory where the database files live. Note: this value is not set when running *Replay Service*. See Section 3.2.8
user_data_file_expression	The expression used to generate the names of the database files. Note: this value is not set when running *Replay Service*. See Section 3.2.8

Table 3.22: `ParticipantInfo`

Field Name	Description
name	Name of the *DomainParticipant* instance, as specified in the `name` attribute of the corresponding configuration tag.

Table 3.23: `ServiceEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.
builtin_sqlite	See Table 3.24

Table 3.24: `SqliteDatabaseEvent`

Field Name	Description
current_db_directory	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.
current_file	Path to the current file where data is being stored to. Note: this value is not set when running *Replay Service*.
rollover_count	The total number of file rollover events that have happened to this moment. Note: this value is not set when running *Replay Service*.

Table 3.25: `ServicePeriodic`

Field Name	Description
host	See Table 7.10.
process	See Table 7.12.
builtin_sqlite	See Table 3.26

Table 3.26: `SqliteDatabasePeriodic`

Field Name	Description
current_file_size	The size in bytes of the current file where data is being stored. Note: this value is not set when running *Replay Service*.
current_timestamp_sec	This time represents the seconds *Recording Service* has been running.
current_timestamp_nanosec	This time represents the nanoseconds *Recording Service* has been running.

Session

Listing 3.2: *Session* Types

```
@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;
};
```

Table 3.27: `SessionConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
default_participant_name	The name of the default participant configuration.

Table 3.28: `SessionEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.

Table 3.29: `SessionPeriodic`

Field Name	Description
network_performance	Provides network performance metric as an aggregation of the same metric across the contained *Topics* and *TopicGroups*. See Section 7.3.4.

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 {
    @optional Service::Monitoring::NetworkPerformance network_performance;
    int64 topic_count;
};
```

Table 3.30: `TopicGroupConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
participant_name	Name of the *DomainParticipant* from which the *Topic* is created.

Table 3.31: `TopicGroupEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.

Table 3.32: `TopicGroupPeriodic`

Field Name	Description
network_performance	Provides network performance metric as an aggregation of the same metric across the contained *Topics*. See Section 7.3.4.
topic_count	Current number of *Topics* created from this *TopicGroup*.

Topic

Listing 3.4: *Topic* Types

```
@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 3.33: `TopicConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
topic_name	Topic name as specified in the configuration.
registered_type_name	Topic registered type name as specified in the configuration.
paritcipant_name	Name of the *DomainParticipant* from which the *Topic* is created.
topic_group	GUID of the *TopicGroup* from which this *Topic* was created. This field is set to zero for standalone *Topics*.

Table 3.34: `TopicEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.

Table 3.35: `TopicPeriodic`

Field Name	Description
network_performance	Provides network performance metric as an aggregation of the same metric across the contained *Topics*. See Section 7.3.4.
matched_status	Provides information about the matched endpoints associated with this *Topic*.

3.5 Tutorials

3.5.1 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.

```
<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.dat</filename_expression>
            </fileset>
        </sqlite>
    </storage>

    <!--  Top-level domain settings -->
    <domain_participant name="Participant0">
        <domain_id>0</domain_id>
    </domain_participant>
```

(continues on next page)

(continued from previous page)

```
    <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>
</recording_service>
```

This configuration is recording all topics in domain 0 into a database file named rti_recorder_default_json.dat. 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.

```
<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.dat</filename_expression>
            </fileset>
        </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>Square</allow_topic_name_filter>
            <deny_topic_name_filter>rti/*</deny_topic_name_filter>
        </topic_group>
    </session>
</recording_service>
```

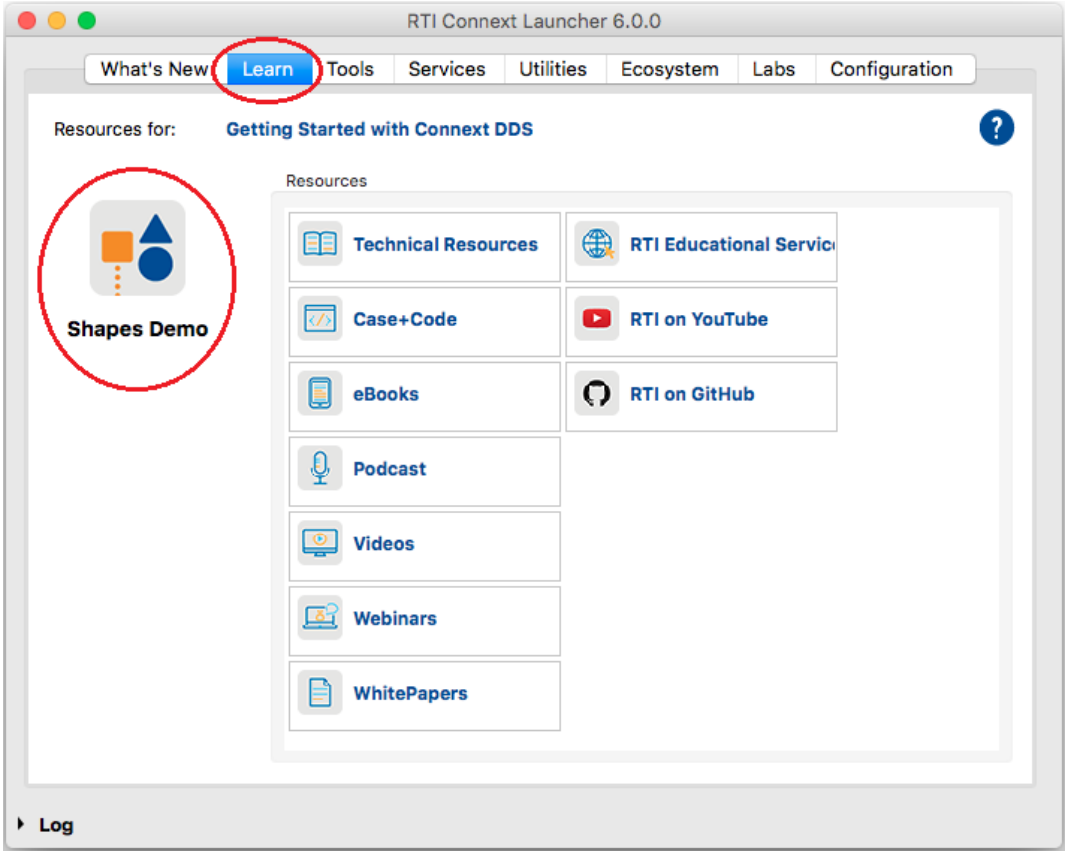
Start Shapes Demo

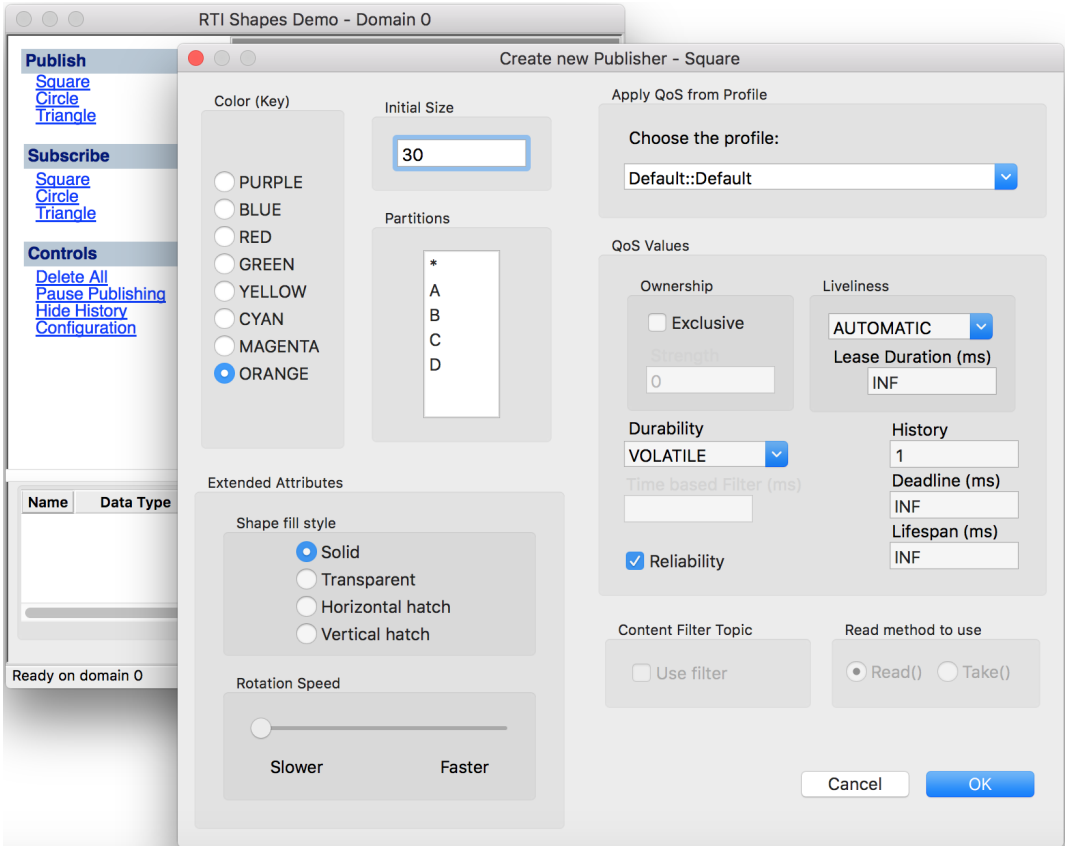
Start *Shapes Demo* from the *Launcher* tool.

In *Shapes Demo*, start two publishers: One Square publisher and one Circle publisher.

Start Recording Service

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


RTI Connext Launcher 6.0.0
What's New
Learn
Tools
Services
Utilities
Ecosystem
Labs
Configuration
Resources for:
Getting Started with Connext DDS
Shapes Demo
Resources
Technical Resources
RTI Educational Servic
Case+Code
RTI on YouTube
eBooks
RTI on GitHub
Podcast
Videos
Webinars
WhitePapers
Log


RTI Shapes Demo - Domain 0
Publish
Square
Circle
Triangle
Subscribe
Square
Circle
Triangle
Controls
Delete All
Pause Publishing
Hide History
Configuration
Name
Data Type
Ready on domain 0
Create new Publisher - Square
Color (Key)
PURPLE
BLUE
RED
GREEN
YELLOW
CYAN
MAGENTA
ORANGE
Initial Size
30
Partitions
*
A
B
C
D
Apply QoS from Profile
Choose the profile:
Default::Default
QoS Values
Ownership
Exclusive
Strength
0
Liveliness
AUTOMATIC
Lease Duration (ms)
INF
Durability
VOLATILE
Time based Filter (ms)
History
1
Deadline (ms)
INF
Lifespan (ms)
INF
Reliability
Extended Attributes
Shape fill style
Solid
Transparent
Horizontal hatch
Vertical hatch
Rotation Speed
Slower
Faster
Content Filter Topic
Use filter
Read method to use
Read()
Take()
Cancel
OK

```
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. (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 rti_recorder_default_json.dat
```

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}
```

3.5.2 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 3.5.1.

Viewing with Admin Console

Use the *Launcher* tool 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, **Recording 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. For example, *RecordAll@Square*.

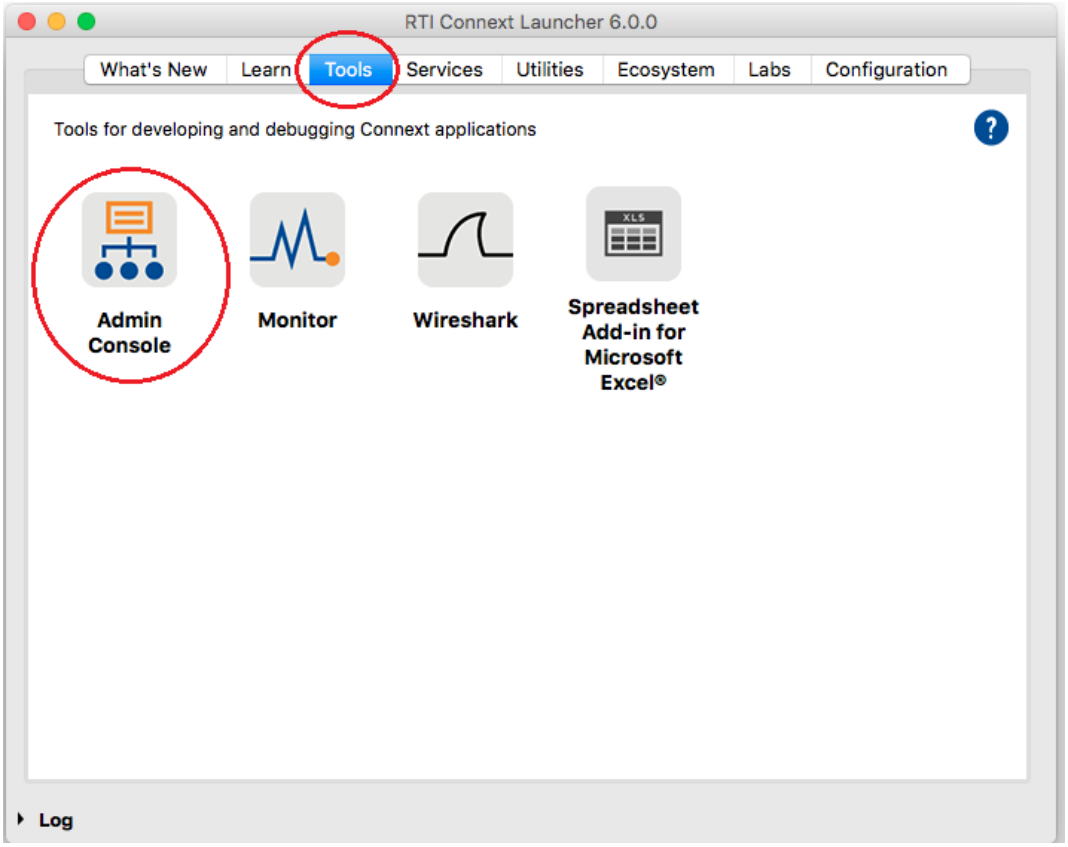
The second tab, **Recording Configuration**, displays the configuration that was used to configure the running Recording Service instance.

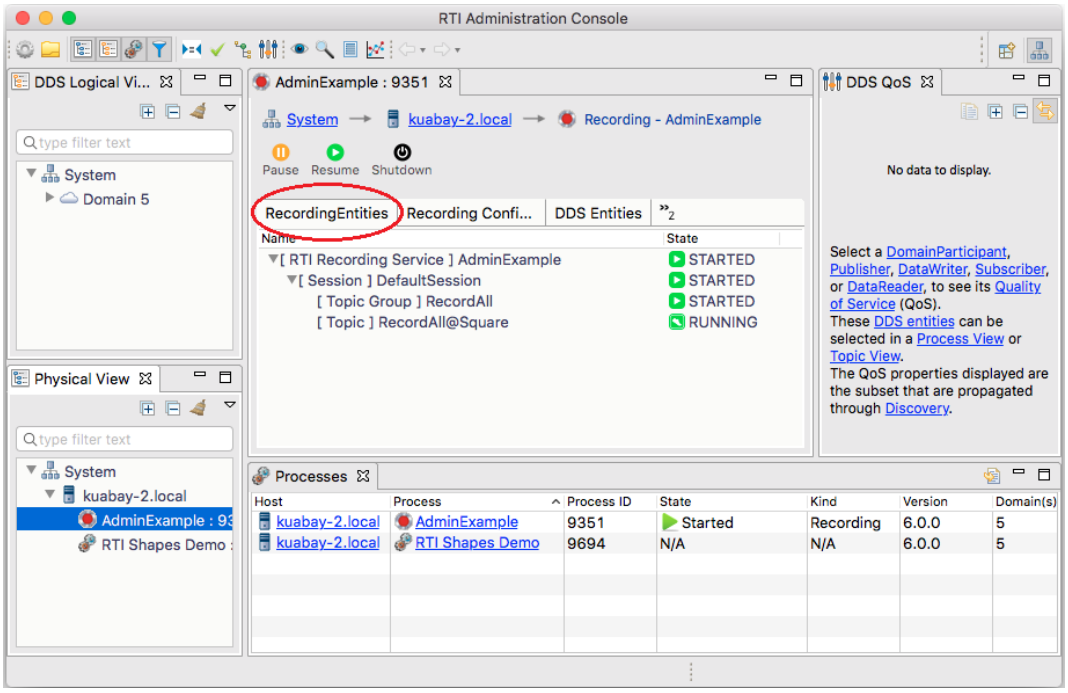
The third tab, **DDS Entities**, displays the DDS entities that have been created by the Recording Service. If you click on individual DataReaders, you can view their QoS in the DDS QoS view.

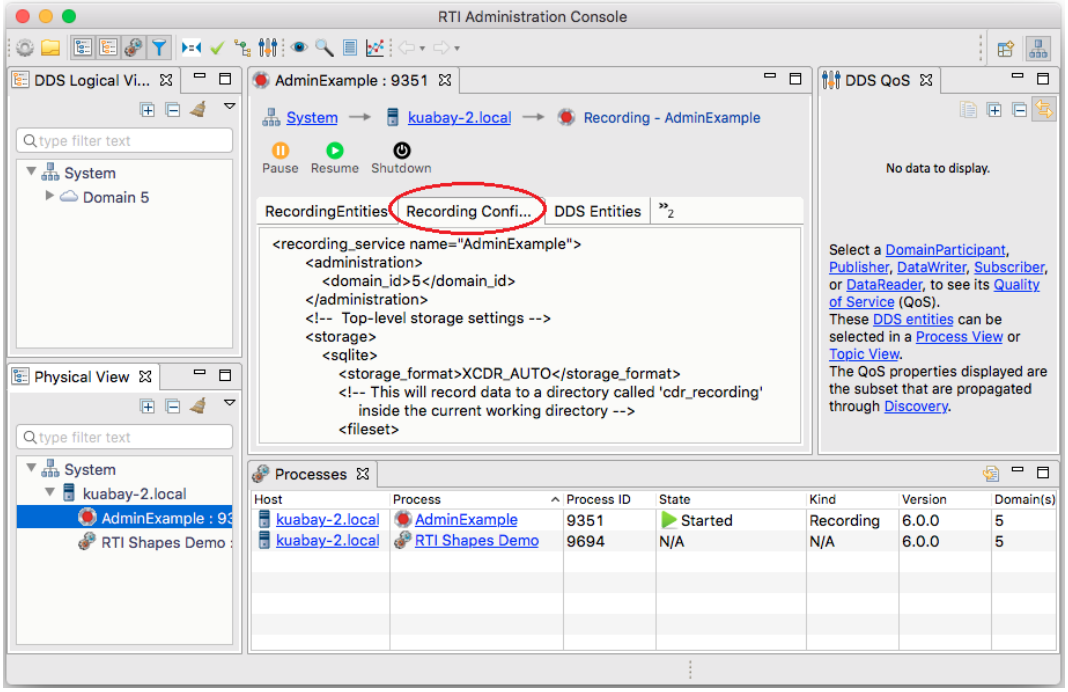
The fourth tab, **Recording Information**, shows details about the recording, such as:

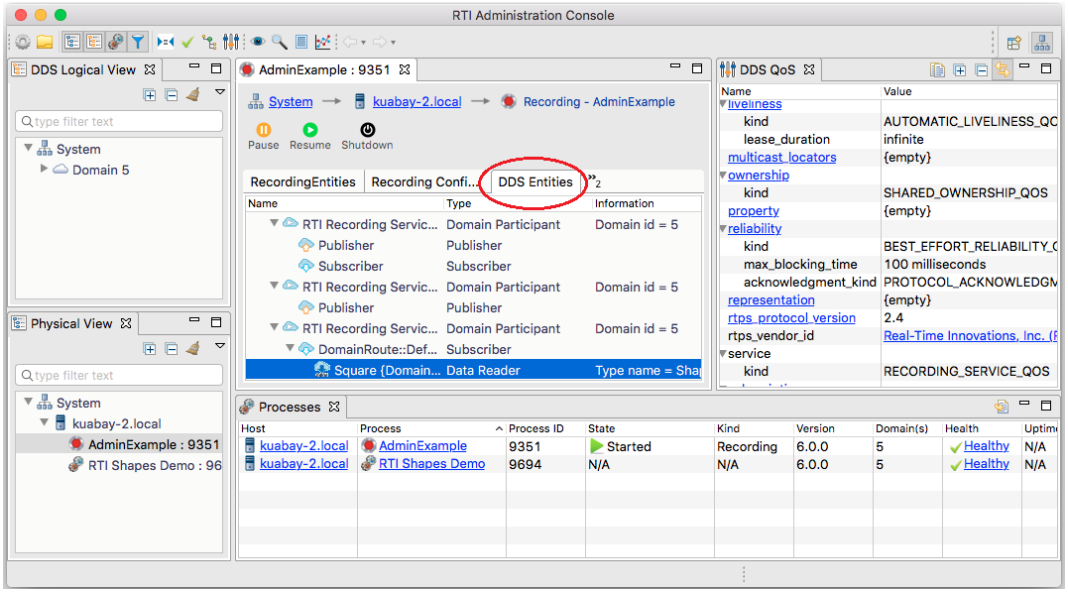
- The *Connext DDS* version number
- The name of the database file it is recording (if you are using the builtin SQLite storage)
- The current size of the database file you are recording (if you are using the builtin SQLite storage)

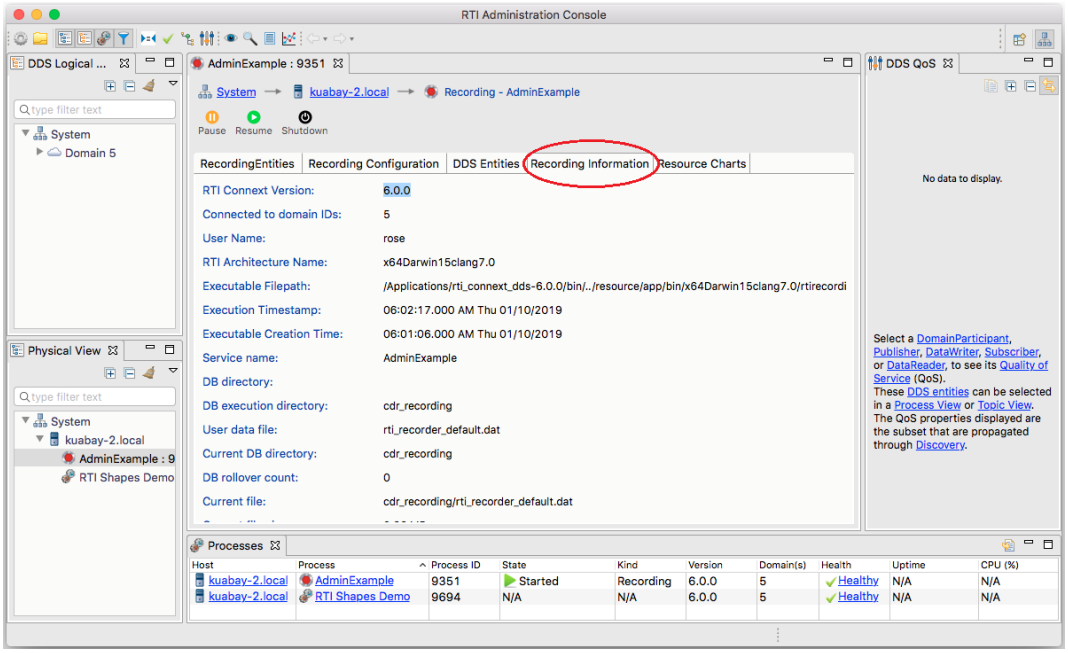
The final tab, **Resource Charts**, allows you to monitor *Recording Service's* CPU and memory usage.

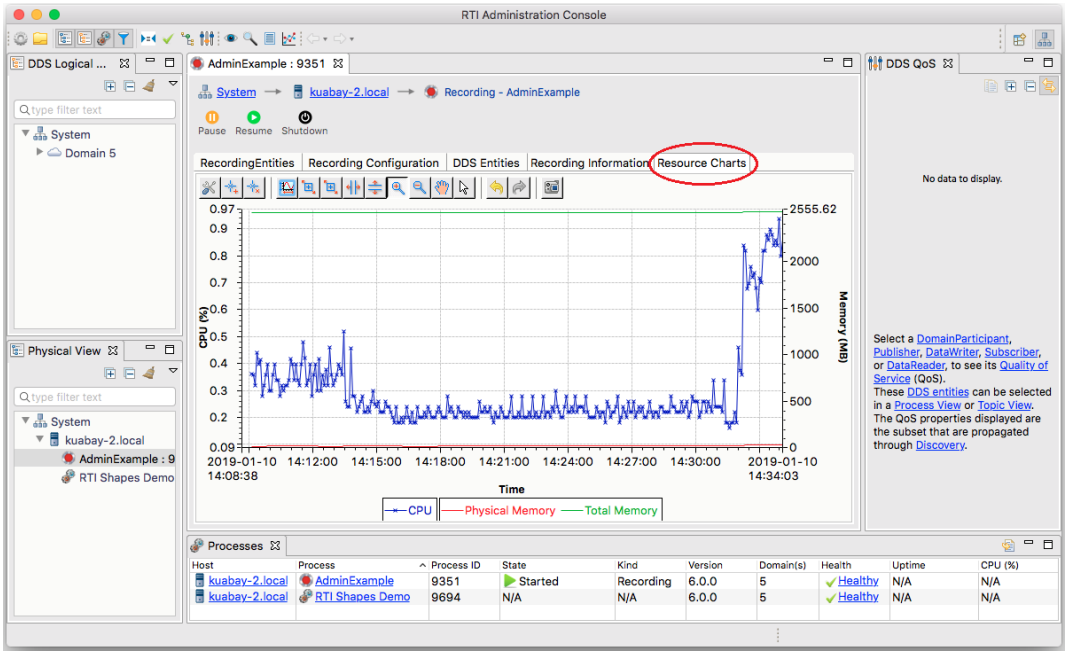

RTI Connext Launcher 6.0.0
What's New
Learn
Tools
Services
Utilities
Ecosystem
Labs
Configuration
Tools for developing and debugging Connext applications
Admin Console
Monitor
Wireshark
Spreadsheet Add-in for Microsoft Excel®
Log


RTI Administration Console
DDS Logical Vi...
type filter text
System
Domain 5
Physical View
type filter text
System
kuabay-2.local
AdminExample : 93
RTI Shapes Demo
AdminExample : 9351
System
kuabay-2.local
Recording - AdminExample
Pause
Resume
Shutdown
RecordingEntities
Recording Confi...
DDS Entities
Name
State
[RTI Recording Service] AdminExample
STARTED
[Session] DefaultSession
STARTED
[Topic Group] RecordAll
STARTED
[Topic] RecordAll@Square
RUNNING
DDS QoS
No data to display.
Select a DomainParticipant, Publisher, DataWriter, Subscriber, or DataReader, to see its Quality of Service (QoS).
These DDS entities can be selected in a Process View or Topic View.
The QoS properties displayed are the subset that are propagated through Discovery.
Processes
Host
Process
Process ID
State
Kind
Version
Domain(s)
kuabay-2.local
AdminExample
9351
Started
Recording
6.0.0
5
kuabay-2.local
RTI Shapes Demo
9694
N/A
N/A
6.0.0
5


RTI Administration Console
DDS Logical Vi...
type filter text
System
Domain 5
Physical View
type filter text
System
kuabay-2.local
AdminExample : 93
RTI Shapes Demo :
AdminExample : 9351
System
kuabay-2.local
Recording - AdminExample
Pause
Resume
Shutdown
RecordingEntities
Recording Confi...
DDS Entities
<recording_service name="AdminExample">
<administration>
<domain_id>5</domain_id>
</administration>
<!-- Top-level storage settings -->
<storage>
<sqlite>
<storage_format>XCDR_AUTO</storage_format>
<!-- This will record data to a directory called 'cdr_recording'
inside the current working directory -->
<fileset>
DDS QoS
No data to display.
Select a DomainParticipant, Publisher, DataWriter, Subscriber, or DataReader, to see its Quality of Service (QoS).
These DDS entities can be selected in a Process View or Topic View.
The QoS properties displayed are the subset that are propagated through Discovery.
Processes
Host
Process
Process ID
State
Kind
Version
Domain(s)
kuabay-2.local
AdminExample
9351
Started
Recording
6.0.0
5
kuabay-2.local
RTI Shapes Demo
9694
N/A
N/A
6.0.0
5


RTI Administration Console
DDS Logical View
type filter text
System
Domain 5
Physical View
type filter text
System
kuabay-2.local
AdminExample : 9351
RTI Shapes Demo : 96
AdminExample : 9351
System
kuabay-2.local
Recording - AdminExample
Pause
Resume
Shutdown
RecordingEntities
Recording Confi...
DDS Entities
Name
Type
Information
RTI Recording Servic...
Domain Participant
Domain id = 5
Publisher
Publisher
Subscriber
Subscriber
RTI Recording Servic...
Domain Participant
Domain id = 5
Publisher
Publisher
RTI Recording Servic...
Domain Participant
Domain id = 5
DomainRoute::Def...
Subscriber
Square {Domain...
Data Reader
Type name = Sha
DDS QoS
Name
Value
liveliness
kind
AUTOMATIC_LIVELINESS_QC
lease_duration
infinite
multicast_locators
{empty}
ownership
kind
SHARED_OWNERSHIP_QOS
property
{empty}
reliability
kind
BEST_EFFORT_RELIABILITY_(
max_blocking_time
100 milliseconds
acknowledgment_kind
PROTOCOL_ACKNOWLEDGM
representation
{empty}
rtps_protocol_version
2.4
rtps_vendor_id
Real-Time Innovations, Inc. (F
service
kind
RECORDING_SERVICE_QOS
Processes
Host
Process
Process ID
State
Kind
Version
Domain(s)
Health
kuabay-2.local
AdminExample
9351
Started
Recording
6.0.0
5
Healthy
N/A
kuabay-2.local
RTI Shapes Demo
9694
N/A
N/A
6.0.0
5
Healthy
N/A

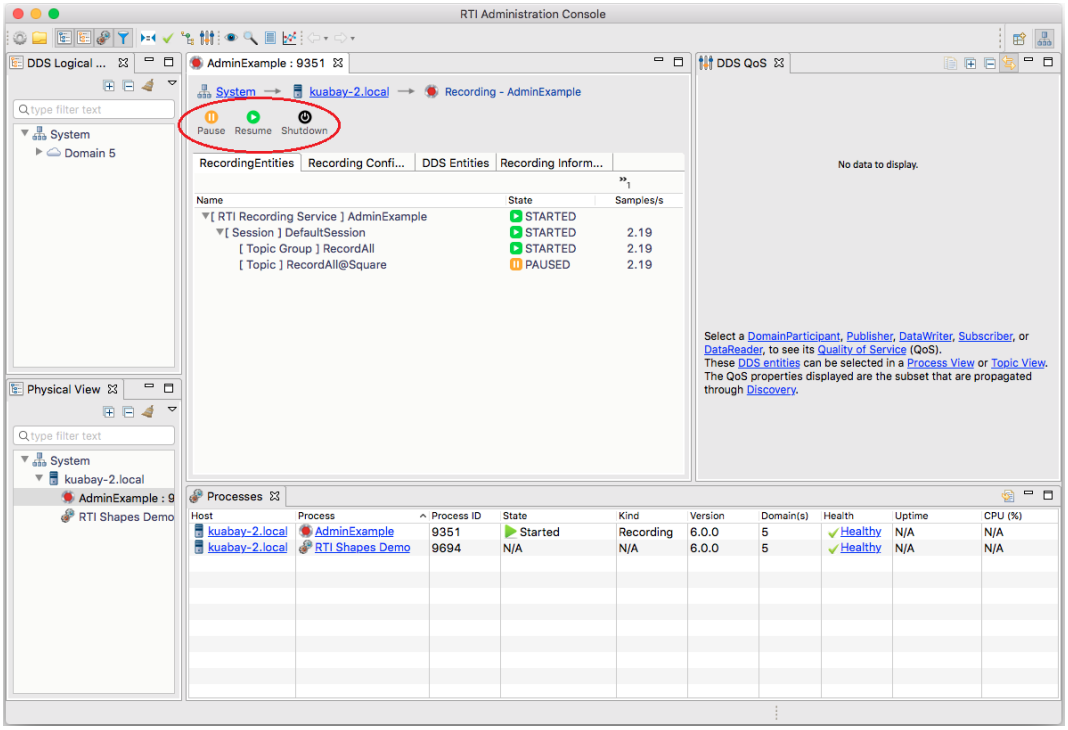

RTI Administration Console
DDS Logical ...
AdminExample : 9351
DDS QoS
System
Domain 5
System → kuabay-2.local → Recording - AdminExample
Pause
Resume
Shutdown
RecordingEntities
Recording Configuration
DDS Entities
Recording Information
Resource Charts
RTI Connext Version: 6.0.0
Connected to domain IDs: 5
User Name: rose
RTI Architecture Name: x64Darwin15clang7.0
Executable Filepath: /Applications/rti_connext_dds-6.0.0/bin/../resource/app/bin/x64Darwin15clang7.0/rtirecordi
Execution Timestamp: 06:02:17.000 AM Thu 01/10/2019
Executable Creation Time: 06:01:06.000 AM Thu 01/10/2019
Service name: AdminExample
DB directory:
DB execution directory: cdr_recording
User data file: rti_recorder_default.dat
Current DB directory: cdr_recording
DB rollover count: 0
Current file: cdr_recording/rti_recorder_default.dat
No data to display.
Select a DomainParticipant, Publisher, DataWriter, Subscriber, or DataReader, to see its Quality of Service (QoS).
These DDS entities can be selected in a Process View or Topic View.
The QoS properties displayed are the subset that are propagated through Discovery.
Physical View
kuabay-2.local
AdminExample : 9
RTI Shapes Demo
Processes
Host
Process
Process ID
State
Kind
Version
Domain(s)
Health
Uptime
CPU (%)
kuabay-2.local AdminExample 9351 Started Recording 6.0.0 5 Healthy N/A N/A
kuabay-2.local RTI Shapes Demo 9694 N/A N/A 6.0.0 5 Healthy N/A N/A


RTI Administration Console
DDS Logical ...
AdminExample : 9351
DDS QoS
System
Domain 5
System → kuabay-2.local → Recording - AdminExample
Pause
Resume
Shutdown
RecordingEntities
Recording Configuration
DDS Entities
Recording Information
Resource Charts
CPU (%)
Memory (MB)
2555.62
2019-01-10 14:08:38
2019-01-10 14:34:03
Time
CPU
Physical Memory
Total Memory
No data to display.
Select a DomainParticipant, Publisher, DataWriter, Subscriber, or DataReader, to see its Quality of Service (QoS).
These DDS entities can be selected in a Process View or Topic View.
The QoS properties displayed are the subset that are propagated through Discovery.
Physical View
kuabay-2.local
AdminExample : 9
RTI Shapes Demo
Processes
Host
Process
Process ID
State
Kind
Version
Domain(s)
Health
Uptime
CPU (%)
kuabay-2.local AdminExample 9351 Started Recording 6.0.0 5 Healthy N/A N/A
kuabay-2.local RTI Shapes Demo 9694 N/A N/A 6.0.0 5 Healthy N/A N/A

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.
- Shut down: This shuts down the *Recording Service* application. To restart the application, you must re-run from *Launcher* or the command line.



3.5.3 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.

```
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();
```

3.5.4 Plugging in Custom Storage

Note: The APIs for custom storage are not available in the Basic bundle.

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 StorageStreamWriters.
- **StorageWriter:**
  - A function for creating StorageStreamWriters 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 StorageStreamWriters for the builtin 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 builtin topic.
  - A function for deleting user-data StorageStreamWriters. *Recording Service* expects this function to be able to work with streams writers created for user-data samples only.
  - Three functions to delete the StorageStreamWriters representing the DDS builtin 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.
- **StorageStreamWriter:**
  - 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 builtin discovery topics: *StorageParticipantWriter*, *StoragePublicationWriter* and *StorageSubscriptionWriter*. 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 located here:

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

3.5.5 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:

```
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
```

3.5.6 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.

3.5.7 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:

```
<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. 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 4.5.4. They can also be used with *Converter*, see Section 5.3.1.

3.6 Troubleshooting

3.6.1 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=Triangle] type_name=ShapeType
[/recording_services/RecorderService/domain_participants/Participant0|STREAM_DISCOVERED␣
↪name=Triangle|../../sessions/DefaultSession/topics/RecordAll@Triangle|CREATE]
[/recording_services/RecorderService/domain_participants/Participant0|STREAM_DISCOVERED␣
↪name=Triangle|../../sessions/DefaultSession/topics/RecordAll@Triangle|ENABLE]
[/recording_services/RecorderService/sessions/DefaultSession/topics/
↪RecordAll@Triangle|START]
[/recording_services/RecorderService/sessions/DefaultSession/topics/
↪RecordAll@Triangle|RUN]
```

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]
PRESPsService_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.

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

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 4.2.3 for the XML elements that can be overridden with command-line parameters.

Table 4.1: Replay Service Command-Line Parameters

Parameter	Description
-appName <string>	Application name used to identify this execution for remote administration, and to name the *Connext DDS* participant.
-cfgFile <string>	Semicolon-separated list of configuration file paths. **Default**: Unspecified
-cfgName	Configuration name Used to find a `<replay_service>` matching tag in the configuration file.
-domainIdBase <int>	This value is added to the domain IDs in the `<domain_participant>` 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. **Default**: 0
-D<name>=<value>	Defines a variable that can be used as an alternate replacement for XML environment variables, specified in the form $(VAR_NAME). Note that definitions in the environment take precedence over these definitions.
-help	Shows this help.
-heapSnapshotDir	Output directory where the heap monitoring snapshots are dumped. The filename format is: RTI_heap_<appName>_<processId>_<index>.log
-heapSnapshotPeriod <sec>	Period at which heap monitoring snapshots are dumped. Enables heap monitoring if > 0. **Default**: 0 (disabled)
-logFormat <format>	A mask to configure the format of the log messages for both *Replay 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.

Continued on next page

Table 4.1 – continued from previous page

Parameter	Description
-verbosity [0-6]	**RTI Service verbosity** • 0 - silent • 1 - exceptions (*Connext DDS* and *Replay Service*) • 2 - warnings (*Replay Service*) • 3 - information (*Replay Service*) • 4 - warnings (*Connext DDS* and *Replay Service*) • 5 - tracing (*Replay Service*) • 6 - tracing (*Connext DDS* and *Replay Service*) **Default**: 1 (exceptions)
-version	Prints the program version and exits.

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)
```

4.2 Configuration

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

4.2.1 How to Load the XML Configuration

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

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

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

`[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 version="1.0" encoding="UTF-8"?>
<!-- All of our files start with a dds tag -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="../schema/rti_replay_service.xsd">
    <!-- Qos Library -->
    <qos_library name="MyQosLib">
        <qos_profile name="DurableQos">
            <datawriter_qos>
                <reliability>
                    <kind>RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
            </datawriter_qos>
        </qos_profile>
    </qos_library>

    <!-- Replay service instance -->
    <replay_service name="MyReplayService">
        <!-- Input file to use when replaying -->
        <storage>
            <sqlite>
                <database_dir>.</database_dir>
            </sqlite>
        </storage>

        <!-- Select the begin and end time of the data to replay -->
        <data_selection>
            <time_range>
                <begin_time>
```

(continues on next page)

(continued from previous page)

```
                    <sec>200</sec>
                    <nanosec>2000000</nanosec>
                </begin_time>
            </time_range>
        </data_selection>

        <!-- Domain Participant for replaying data -->
        <domain_participant name="Participant0">
            <domain_id>0</domain_id>
        </domain_participant>

         <!-- The session that is replaying the data.  Contains configuration
              for publishers, threading, group of topics to replay.  -->
        <session name="Session" default_participant_ref="Participant0">
            <!-- Topics to replay in this session -->
            <topic_group name="RecordAll">
                <!-- Topics to replay -->
                <allow_topic_name_filter>*</allow_topic_name_filter>
            </topic_group>
        </session>
    </replay_service>
</dds>
```

4.2.2 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, *Replay Service* relies on an XSD file that describes the format of the XML content. We recommend including a reference to this document in the XML file that contains the service's configuration; this 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/resource/schema/rti_replay_service.xsd`. (See Section 1.3 for the path to NDDSHOME.)

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

```
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation=
↪"../schema/rti_recording_service.xsd">
    <!-- ... -->
</dds>
```

4.2.3 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.

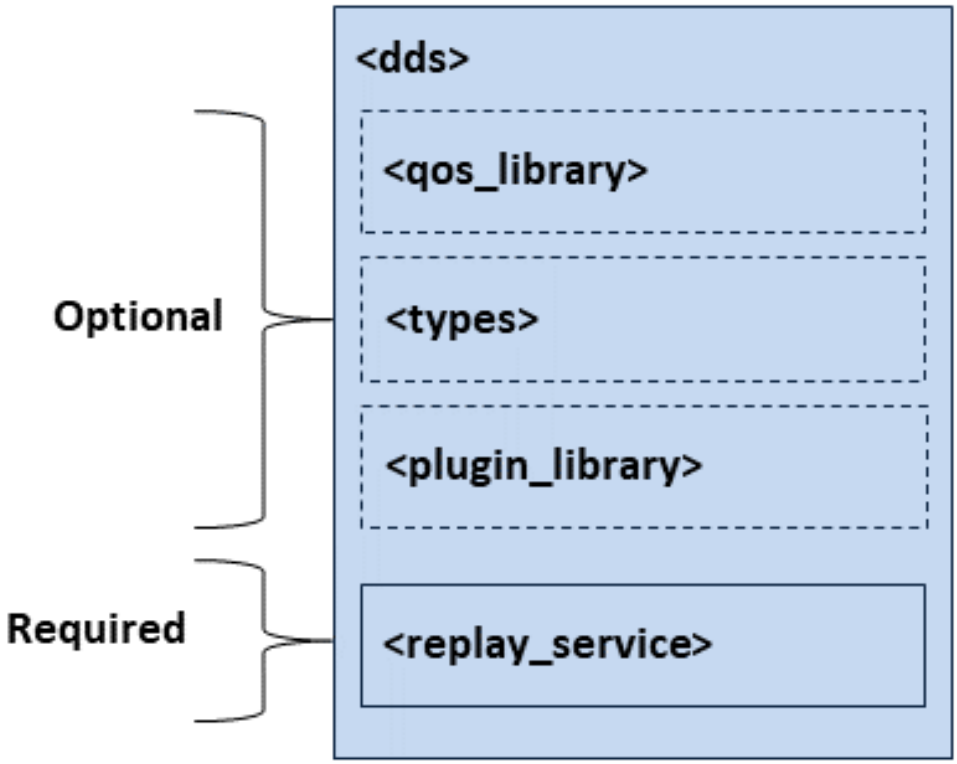


Figure 4.1: Top-level Tags in the *Replay* Configuration File

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

Tags within <dds>	Description	Multiplicity
<qos_library>	Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for a *Connext DDS* QoS profile file — see Configuring QoS with XML, in the RTI Connext DDS Core Libraries User's Manual.	0..*
<types>	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.	0..*

Continued on next page

Table 4.2 – continued from previous page

Tags within <dds>	Description	Multiplicity
<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 4.5.3. • 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 3.2.14	0..*

<replay_service>	**Required.** Specifies a *Replay Service* configuration. See Section 4.2.4.
	Attributes
	• `name`: uniquely identifies a service configuration. **Required**.
	Example
	`<replay_service name="ReplayAll">`
	`<!-- your service settings ... -->`
	`</replay_service>`
		1..*

4.2.4 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

Tags within <replay_service>	Description	Multiplicity
<administration>	Enables remote administration. When administration is enabled, monitoring is also enabled by default. See Section 4.2.5.	0..1
<monitoring>	Enables monitoring for *Replay Service*, including statistics. See Section 4.2.6.	0..1
<storage>	Describes how the data will be loaded from storage. See Section 4.2.7. If this is not specified, data will be loaded from the current working directory.	0..1

Continued on next page

Table 4.3 – continued from previous page

Tags within <replay_service>	Description	Multiplicity
<playback>	Specifies the timing rules for how data is played back. See Section 4.2.11.	0..1
<data_selection>	Enables selection of a subset of data to replay. Supports selecting replay data by time (or tagged time). See Section 4.2.12.	0..1

<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 4.2.10.
	Attributes:
	• `name`: Uniquely defines a DomainParticipant. **Required**.
	Example
	`<domain_participant name="Participant3">`
	`<domain_id>3</domain_id>`
	`<!-- Participant QoS -->`
	`</domain_participant>`
		1..*

Continued on next page

Table 4.3 – continued from previous page

Tags within <replay_service>	Description	Multiplicity
<session>	**Required**. Active component of *Replay Service* for replaying data. Contains one or more threads that can be used for replay. See Section 4.2.14. **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** `<session name="Session" default_participant_ref="Participant3">   <!-- ... topics / groups of topics to record --> </session>`	1..*

Example: Specify a Replay Service Configuration in 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.

4.2.5 Administration

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

See Section 4.3 for details on using remote administration.

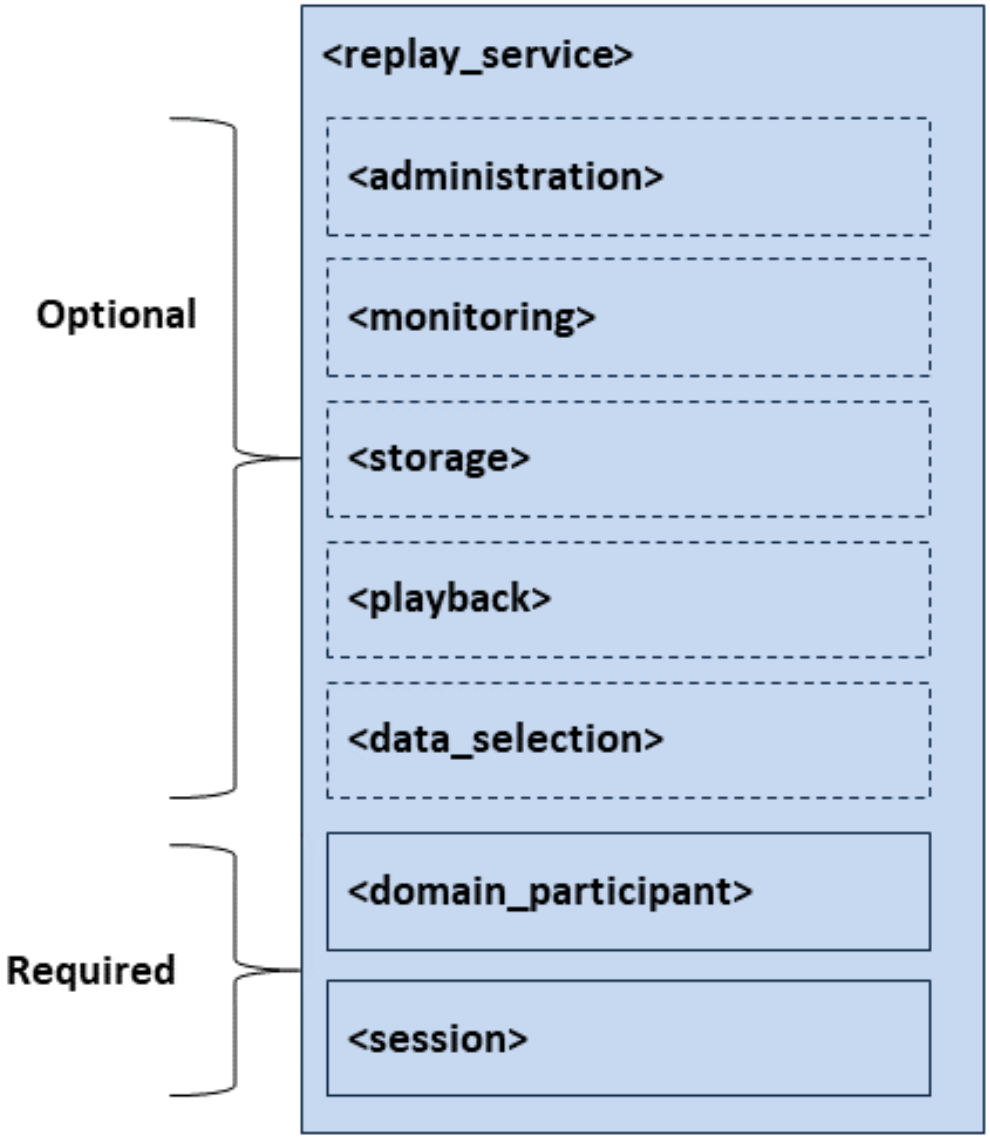


Figure 4.2: Tags used to configure a *Replay Service* instance

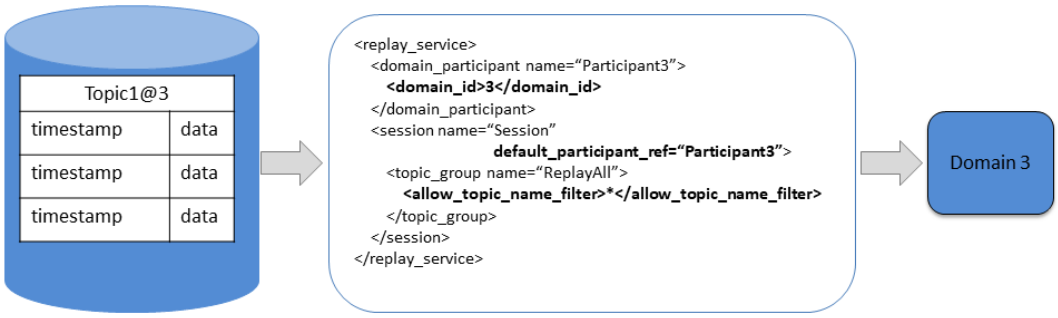


Figure 4.3: The domain ID specified inside a DomainParticipant, in combination with the Topic name, is used to load data from recorded tables. In this case, the DomainParticipant's domain ID is set to 3. The `topic_group` specifies all topics.

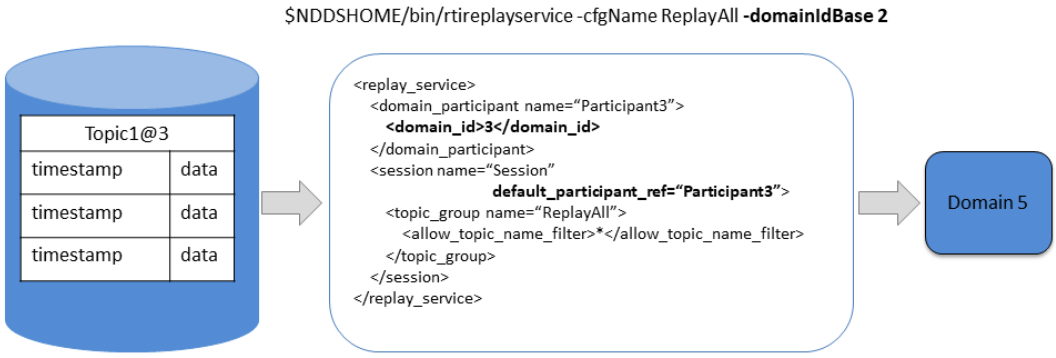


Figure 4.4: To replay data in a different domain than where it was recorded, use the `-domainIdBase` command-line parameter to specify a base ID for the replay domain. Here, by specifying a domainIdBase of 2, all Topics will be replayed in their original domain, plus 2.

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

Tags within <administration>	Description	Multiplicity
<domain_id>	Domain ID used for remote administration. Also used for monitoring by default.	0..1
<participant_qos>	QoS used by the administration DomainParticipant. If the tag is not defined, *Connext DDS* defaults will be used.	0..1
<publisher_qos>	QoS used by the administration Publisher. If the tag is not defined, *Connext DDS* defaults will be used.	0..1
<subscriber_qos>	QoS used by the administration Subscriber. If the tag is not defined, *Connext DDS* defaults will be used.	0..1
<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 3.2.15.	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.

4.2.6 Monitoring

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

Table 4.5: Monitoring Tags in Replay Service's Configuration File

Tags within <monitoring>	Description	Multiplicity
<enabled>	Whether to enable monitoring of the service. Default: Disabled unless administration is enabled	0..1

Continued on next page

Table 4.5 – continued from previous page

Tags within <monitoring>	Description	Multiplicity
<domain_id>	Domain ID used for monitoring. Default: Use the domain ID specified for monitoring	0..1
<datawriter_qos>	QoS used by monitoring DataWriter(s)	0..1
<publisher_qos>	QoS used by monitoring Publisher(s)	0..1
<participant_qos>	QoS used by monitoring DomainParticipant	0..1
<statistics_sampling_period>	How frequently to sample the service's statistics, using the tags <sec> or <nanosec>. For example, <sec>1</sec> samples the service's statistics every second.	0..1
<status_publication_period>	How frequently to publish the service status, using the tags <sec> or <nanosec>. For example, <sec>1</sec> publishes the service's status every second.	0..1

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

4.2.7 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 4.6: Storage Tags in Replay Service's Configuration File

Tags within <storage>	Description	Multiplicity
<sqlite>	Enables replaying data from an SQLite database file See Section 4.2.7.	0..1
<plugin>	Enables storing data in an external library that you specify. Attributes • `plugin_name`: Name of the plugin that creates a storage plugin object. This name shall refer to a registered storage plugin. See Section 7.4 for details on how to register plugins). See Section 4.2.7 for more about using this tag. See Section 4.5.3 for a tutorial on plugging in custom storage.	0..1

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

Tags within <sqlite>	Description	Multiplicity
<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

Continued on next page

Table 4.7 – continued from previous page

Tags within <sqlite>	Description	Multiplicity

<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>`
	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;`
		0..1

Plugin

Table 4.8: Storage plugin Tag in the Configuration File

Tags within <plugin>	Description	Multiplicity
<property>	Name/value pairs of properties to pass to a storage plugin.	0 or 1

4.2.8 Legacy

Table 4.9: Legacy Tags in the Configuration File

Tags within <legacy>	Description	Multiplicity
<file_path>	Path to the legacy format recorded file. File set and version properties will be obtained automatically from the file itself.	0..1
<domain_mapping>	This tag allows you to link legacy domain names with domain IDs. See Section 4.2.9.	0..1

4.2.9 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

Tags within <domain_mapping>	Description	Multiplicity
<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..*

4.2.10 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 4.11: DomainParticipant Tags in Replay Service's Configuration File

Tags within <domain_participant>	Description	Multiplicity
<domain_id>	The domain ID of tables loaded from the database. This is used (in conjunction with the `-domainIdBase` command-line parameter) to choose which domain to replay into.	1
<participant_qos>	QoS used by this DomainParticipant.	0..1

Continued on next page

Table 4.11 – continued from previous page

Tags within <domain_participant>	Description	Multiplicity

<memory_management>	Configures certain aspects of how *Connext DDS* allocates internal memory. The configuration is per DomainParticipant and therefore affects all the contained DDS entities.
	Example:
	`<memory_management>`
	`  <sample_buffer_min_size>`
	`  1024`
	`  </sample_buffer_min_size>`
	`  <sample_buffer_trim_to_size>`
	`  true`
	`  </sample_buffer_trim_to_size>`
	`</memory_management>`
	This tag includes the following tags:
	• <sample_buffer_min_size>: For all 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.
		0..1

Continued on next page

Table 4.11 – continued from previous page

Tags within <domain_participant>	Description	Multiplicity
<register_type>	Registers a type name and associates it with a type representation. When you define a type in the configuration file, you have to register the type in order to use it in 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..*

4.2.11 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

Tags within <playback>	Description	Multiplicity
<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. Default: Replay will start immediately	0..1

Continued on next page

Table 4.12 – continued from previous page

Tags within <playback>	Description	Multiplicity
<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 4.2.12) 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

4.2.12 Data Selection

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

Table 4.13: Data Selection Tags in Replay Service's Configuration File

Tags within <data_selection>	Description	Multiplicity
<time_range>	Select data to replay from the database, based on start and end times. See Section 4.2.13.	0..1

4.2.13 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 3.5.7).

Table 4.14: Time Range Tags in Replay Service's Configuration File

Tags within <time_range>	Description	Multiplicity
<begin_time>	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.	0..1
<begin_tag>	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 3.5.6. Default: Start at beginning of the file.	0..1
<end_time>	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.	0..1
<end_tag>	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 3.5.6. Default: Stop at end of file.	0..1

4.2.14 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 4.15: Session Tags in Replay Service's Configuration File

Tags within <session>	Description	Multiplicity
<publisher_qos>	Specifies the QoS of the Publisher that will be used by the contained `<topic>` and `<topic_group>`. For information on configuring Pubscriber QoS with XML, see Configuring QoS with XML, in the RTI Connext DDS Core Libraries User's Manual.	0..1

Continued on next page

Table 4.15 – continued from previous page

Tags within <session>	Description	Multiplicity

<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:
	`<thread_pool>`
	`  <mask>MASK_DEFAULT</mask>`
	`  <priority>THREAD_PRIORITY_DEFAULT`
	`↪</priority>`
	`  <stack_size>`
	`  THREAD_STACK_SIZE_DEFAULT`
	`  </stack_size>`
	`</thread_pool>`
	Default values:
	• **size**: 1
	• **mask**: MASK_DEFAULT
	• **priority**: THREAD_PRIORITY_DEFAULT
	• **stack_size**: THREAD_STACK_SIZE_DEFAULT
		0..1

<topic>	Specifies an individual Topic to replay. See Section 4.2.16. **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.	0..*
<topic_group>	Specifies a group of Topics to replay. See Section 4.2.15. **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.	0..*

4.2.15 Topic Group

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

Table 4.16: Topic Group Tags in Replay Service's Configuration File

Tags within <topic_group>	Description	Multiplicity

<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:
	`<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
<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

4.2.16 Topic

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

Table 4.17: Topic Tags in Replay Service's Configuration File

Tags within <topic>	Description	Multiplicity
<topic_name>	The name of the DDS topic to be replayed. If this tag is not present, the name attribute of the *<topic>* 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 Replay to fail when found in the topic name attribute.	0..1
<registered_type_name>	The name of the data type that will be replayed for this topic. **Required**.	1

<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:
	`<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

4.2.17 Plugins

All the pluggable components specific to *Recording Service* are configured within the `<plugin_library>` tag. Table 4.18 describes the available tags.

Plug-ins are categorized and configured based on the source language. *Replay Service* supports C/C++ plugins.

Table 4.18: Configuration tags for plugin libraries

Tags within <plugin_library>	Description	Multiplicity
<storage_plugin>	Specifies a C/C++ *Storage* plugin. See Table 7.16 and Section 3.2.8.	0..*
<transformation_plugin>	Specifies a C/C++ *Transformation* plugin. See Table 7.16 and Section 3.2.12.	0..*

4.3 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 7.2. 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.

4.3.1 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 4.2) 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.

4.3.2 Available Service Resources

Table 4.19 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 7.1.2 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 4.19: Resources and Their Identifiers in *Routing Service*

Resource	Resource Identifier
Replay Service	/replay_service/(rs)

Example

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

Replay Service

Entity with name "MyReplay":

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

Resource identifier:

```
/replay_services/MyReplay
```

4.3.3 Remote API Overview

Table 4.20: Remote Interface Overview

Resource	Operation	Description
Replay-Service	*DELETE /replay_services/(rs)*	Shuts down a running *Replay Service*
	UPDATE /replay_services/(rs)/state	Sets a *Replay Service* state

4.3.4 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 7.2.3).

Valid requested states:

- `STARTED`

- STOPPED
- PAUSED
- RUNNING
- Example

  To pause a replay service with the name "MyReplay":

Request Field	Value
command_action	UPDATE
resource_identifier	/replay_services/MyReplay/state
octet_body	to_cdr_buffer(RTI::Service::EntityStateKind::PAUSED)

4.4 Monitoring

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

Note: *Recording Service* monitoring is based on the *Monitoring Distribution Platform* described in Section 7.3. We recommend that you read Section 7.3 before using *Recording Service* monitoring.

4.4.1 Overview

Enabling Service Monitoring

By default, monitoring is disabled in *Recording Service*. To enable monitoring you can use the <monitoring> tag (see Section 3.2.5) 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 4.21. The complete type relationship is shown in Figure 4.5.

Table 4.21: *Recording Service Keyed Resources*

Keyed Resource Class	Config	Event	Periodic
Service	ServiceConfig	ServiceEvent	ServicePeriodic
Session	SessionConfig	SessionEvent	SessionPeriodic
TopicGroup	TopicGroupConfig	TopicGroupEvent	TopicGroupPeriodic
Topic	TopicConfig	TopicEvent	TopicPeriodic

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

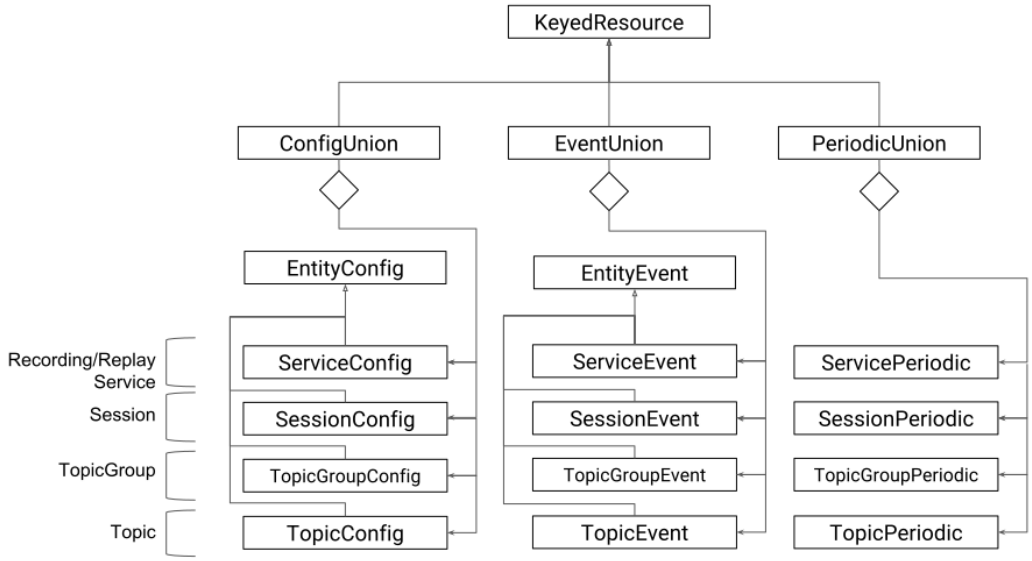


Figure 4.5: Keyed Resource Types for *Recording Service* monitoring

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 3.2.5 for details on configuring the QoS for these entities.

4.4.2 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

```
@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;
};
```

(continues on next page)

(continued from previous page)

```
@mutable @nested
struct SqliteDatabasePeriodic {
    @optional Service::FilePath current_file;
    @optional uint64 current_file_size;
    int32 current_timestamp_sec;
    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;
    @optional SqliteDatabasePeriodic builtin_sqlite;
};
```

Table 4.22: `ServiceConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
application_name	Name of the *Recording Service* instance. The application name is provided through: • `appName` command-line option when run as executable. • `ServiceProperty::application_name` field when run as a library.
application_guid	GUID of the *Recording Service* instance. Unique across all service instances.
host	See Table 7.9.
process	See Table 7.11.
builtin_sqlite	See Table 4.23
participants	Sequence of `ParticipantInfo` objects, one for each *DomainParticipant* inside the *Recording Service*. See Table 4.24.

Table 4.23: `SqliteDatabaseConfig`

Field Name	Description
db_directory	Path to the base directory where the database files live.
execution_directory_expression	This value is not set when running *Replay Service*. See Section 3.2.8
user_data_file_expression	This value is not set when running *Replay Service*.

Table 4.24: `ParticipantInfo`

Field Name	Description
name	Name of the *DomainParticipant* instance, as specified in the `name` attribute of the corresponding configuration tag.

Table 4.25: `ServiceEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.
builtin_sqlite	See Table 4.26

Table 4.26: `SqliteDatabaseEvent`

Field Name	Description
current_db_directory	Path to the current directory where files are being replayed from.
current_file	This value is not set when running *Replay Service*.
rollover_count	This value is not set when running *Replay Service*.

Table 4.27: `ServicePeriodic`

Field Name	Description
host	See Table 7.10.
process	See Table 7.12.
builtin_sqlite	See Table 4.28

Table 4.28: `SqliteDatabasePeriodic`

Field Name	Description
current_file_size	This value is not set when running *Replay Service*.
current_timestamp_sec	The seconds part of the timestamp at which data is being replayed, relative to recorded time.
current_timestamp_nanosec	The nanoseconds part of the timestamp at which data is being replayed, relative to recorded time.

Session

Listing 4.2: *Session* Types

```
@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;
};
```

Table 4.29: `SessionConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
default_participant_name	The name of the default participant configuration.

Table 4.30: `SessionEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.

Table 4.31: `SessionPeriodic`

Field Name	Description
network_performance	Provides network performance metric as an aggregation of the same metric across the contained *Topics* and *TopicGroups*. See Section 7.3.4.

TopicGroup

Listing 4.3: *TopicGroup* Types

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

(continues on next page)

(continued from previous page)

```
        //to avoid unused variable warnings
        int32 _dummy;
    };
    @mutable @nested
    struct TopicGroupPeriodic {
        @optional Service::Monitoring::NetworkPerformance network_performance;
        int64 topic_count;
    };
```

Table 4.32: `TopicGroupConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
participant_name	Name of the *DomainParticipant* from which the *Topic* is created.

Table 4.33: `TopicGroupEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.

Table 4.34: `TopicGroupPeriodic`

Field Name	Description
network_performance	Provides network performance metric as an aggregation of the same metric across the contained *Topics*. See Section 7.3.4.
topic_count	Current number of *Topics* created from this *TopicGroup*.

Topic

Listing 4.4: *Topic* Types

```
    @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;
    };
```

(continues on next page)

(continued from previous page)

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

Table 4.35: `TopicConfig`

Field Name	Description
Inherited fields from `EntityConfig`	See Table 7.13.
topic_name	Topic name as specified in the configuration.
registered_type_name	Topic registered type name as specified in the configuration.
paritcipant_name	Name of the *DomainParticipant* from which the *Topic* is created.
topic_group	GUID of the *TopicGroup* from which this *Topic* was created. This field is set to zero for standalone *Topics*.

Table 4.36: `TopicEvent`

Field Name	Description
Inherited fields from `EntityEvent`	See Table 7.14.

Table 4.37: `TopicPeriodic`

Field Name	Description
network_performance	Provides network performance metric as an aggregation of the same metric across the contained *Topics*. See Section 7.3.4.
matched_status	Provides information about the matched endpoints associated with this *Topic*.

4.5 Tutorials

4.5.1 Example: Getting Started with Replay and Shapes Demo

Start by recording Square data, as described in Section 3.5.1.

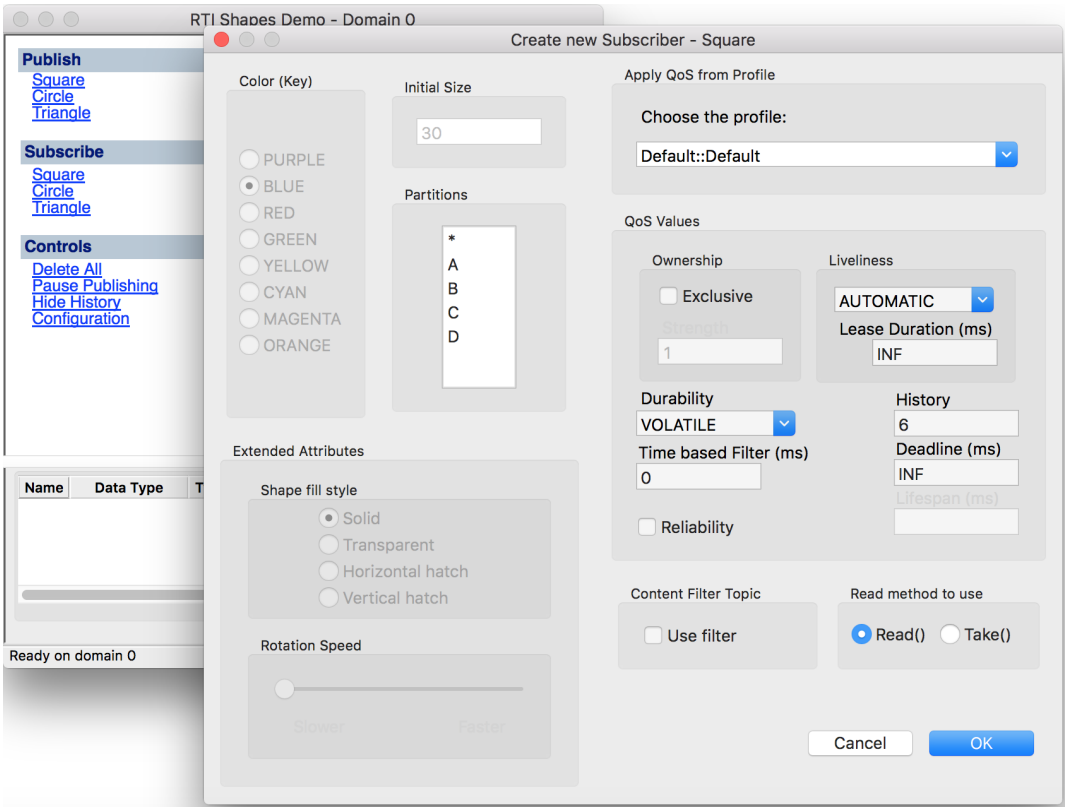
Edit the Replay Configuration

In your <RTI_WORKSPACE>/user_config/recording_service directory, edit the USER_REPLAY_SERVICE.xml file. Change the storage_format to be JSON_SQLITE.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- All of our files start with a dds tag -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="http://community.rti.com/schema/6.0.0/rti_replay_
↪service.xsd">

    <replay_service name="UserReplayService">

        <storage>
            <sqlite>
                <storage_format>JSON_SQLITE</storage_format>
                <!-- No <database_dir> specified means current directory -->
            </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>
        </playback-->

        <!-- Configuration for the service:
        DomainParticipant configuration, storage configuration -->
        <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>
    </replay_service>
</dds>
```

Start Shapes Demo and Subscribe to Squares

Start *Shapes Demo* from the *Launcher* tool and create a Square subscriber as described in Section 3.5.1.



Start Replay Service

Start *Replay Service*:

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

You should see Square data replayed in *Shapes Demo*.

4.5.2 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 3.5.1.

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 version="1.0" encoding="UTF-8"?>

<!-- All of our files start with a dds tag -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="https://community.rti.com/schema/6.0.0/rti_replay_
↪service.xsd">
```

(continues on next page)

(continued from previous page)

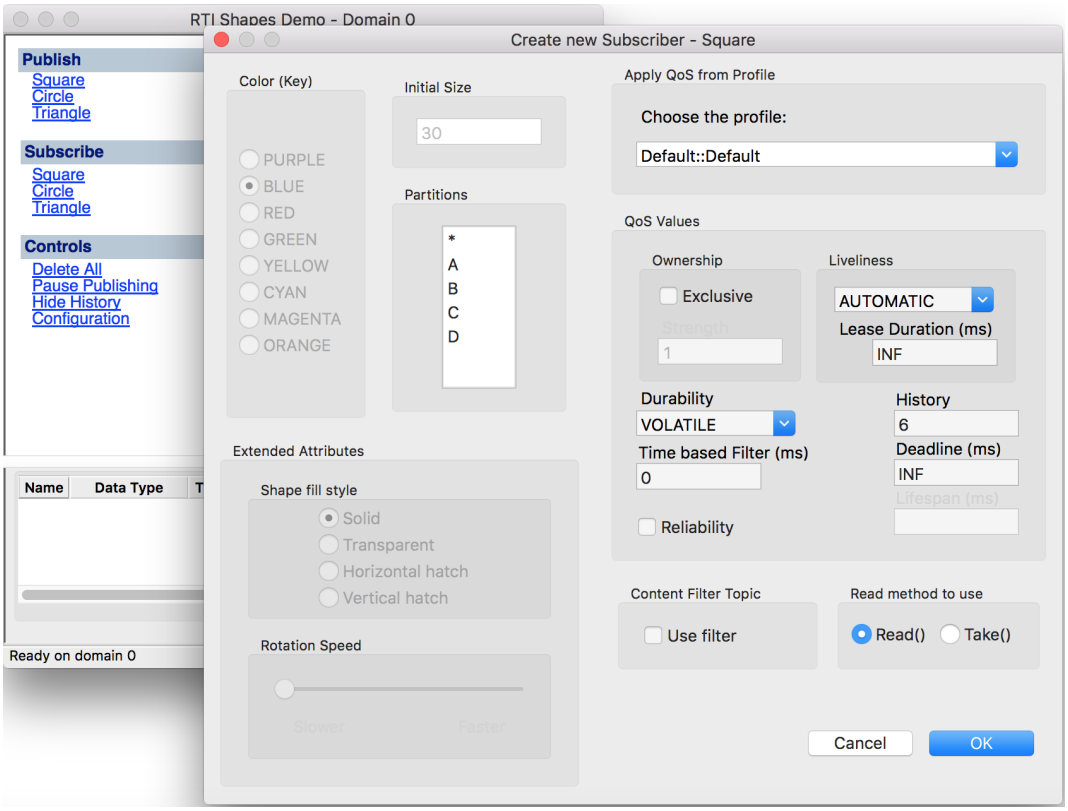
```
  <replay_service name="UserReplayServiceJson">
      <!-- This will look for files in the directory 'json_recording' in the
           current working directory. This is integrated with the Recording
           Service configuration 'UserRecorderServiceJson' in the file
           USER_RECORDING_SERVICE.xml -->
      <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>
  </replay_service>
</dds>
```

Start Shapes Demo

Start *Shapes Demo* from the *Launcher* tool and 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.

4.5.3 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.

Note: The APIs for custom storage are not available in the Basic bundle.

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 retreive 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 time-stamp 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 and a builtin one is provided based on `dds::core::xtypes::DynamicData`.

More detailed API documentation is located in:

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

4.5.4 Using Timestamp Tags with Replay Service

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 3.5.6.

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:

```
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
```

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 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 Troubleshooting

4.6.1 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] ROUTERService_start:!create domain route
[/replay_services/default|START] RTI_RoutingService_start:!start routing service
main:!!start RoutingService error
```

4.6.2 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*.

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.)

```
rticonverter [options]
```

To start *Converter* with a default configuration, enter:

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

Note: *Converter* is pre-loaded with a builtin configuration that has default settings. See Section 5.2.20.

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 5.1: Converter Command-Line Parameters

Parameter	Description
-cfgFile <string>	Semicolon-separated list of configuration file paths. **Default:** Unspecified
-cfgName	Configuration name. This name is used to find a matching `<converter>` tag in the configuration file.
-D<name>=<value>	Defines a variable that can be used as an alternate replacement for XML environment variables, specified in the form $(VAR_NAME). Note that definitions in the environment take precedence over these definitions.
-help	Shows this help.
-verbosity [0-6]	**RTI Service verbosity** • 0 - silent • 1 - exceptions (*Connext DDS* and *Converter*) • 2 - warnings (*Converter*) • 3 - information (*Converter*) • 4 - warnings (*Connext DDS* and *Converter*) • 5 - tracing (*Converter*) • 6 - tracing (Connext DDS* and *Converter*) **Default**: 1 (exceptions)
-version	Prints the program version and exits.

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 version="1.0" encoding="UTF-8"?>

<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="https://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.dat</filename_
↪expression>
                </fileset>
            </sqlite>
        </output_storage>

        <!--  Domain selection: assume 0 by default.
```

(continues on next page)

(continued from previous page)

```
            Converter is not a DDS application, so this won't create a
            DDS Domain Participant, but allows you to select which domains
            to convert -->
        <domain_participant name="Domain0">
            <domain_id>0</domain_id>
        </domain_participant>

        <session name="DefaultSession"
                 default_participant_ref="Domain0">
            <!-- Topics to convert in this session -->
            <topic_group name="DefaultTopicGroup">
                <!-- Rules describing the topics to convert -->
                <allow_topic_name_filter>*</allow_topic_name_filter>
                <deny_topic_name_filter>rti/*</deny_topic_name_filter>
            </topic_group>
        </session>

    </converter>
</dds>
```

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:

```
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation=
↪"../schema/rti_converter.xsd">
    <!-- ... -->
</dds>
```

5.2.3 Builtin Configuration of Converter

Converter comes pre-configured with a configuration that converts a file called rti_recorder_default.dat from XCDR format to a file called rti_recorder_default_converted.dat in JSON_SQLITE

format. See Section 5.2.20 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.

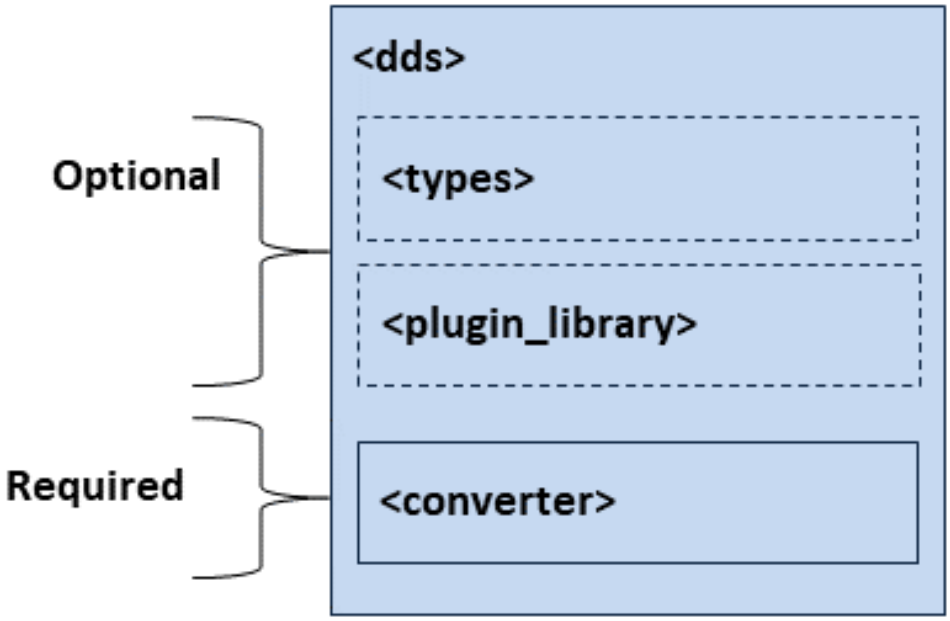


Figure 5.1: Top-level Tags in the *Converter* Configuration File

Table 5.2: Top-Level Tags in Converter's Configuration file

Tags within <dds>	Description	Multiplicity
<types>	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.	0..*
<plugin_library>	Contains a list of storage libraries that you may have implemented to support custom databases. For more information, see Section 3.2.14 and Section 4.2.17.	0..*

<converter>	**Required**. Specifies a *Converter* configuration.
	Attributes
	• `name`: uniquely identifies a *Converter* configuration. **Required**.
	Example
	`<converter name="ConvertAll">`
	`</converter>`
	See Section 5.2.5.
		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

Tags within <converter>	Description	Multiplicity
<input_storage>	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.	0..1
<output_storage>	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.	0..1

Continued on next page

Table 5.3 – continued from previous page

Tags within <converter>	Description	Multiplicity
<data_selection>	Enables selection of a subset of data to convert. Supports selecting data for conversion by time (or tagged time). See Section 5.2.14.	0..1
<session>	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 5.2.17. **Attributes:** • `name`: Uniquely defines a session. **Required**. • `default_participant_ref`: Refers to a `domain_participant` tag, that specifies the domain ID to use from the database when converting this Topic. Children can override this by specifying their own participant.	0..*
<domain_participant>	**Required.** 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 5.2.16. **Attributes:** • `name`: Uniquely defines a DomainParticipant. **Required**.	1..*

Example: Specify a Configuration in 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
```

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.

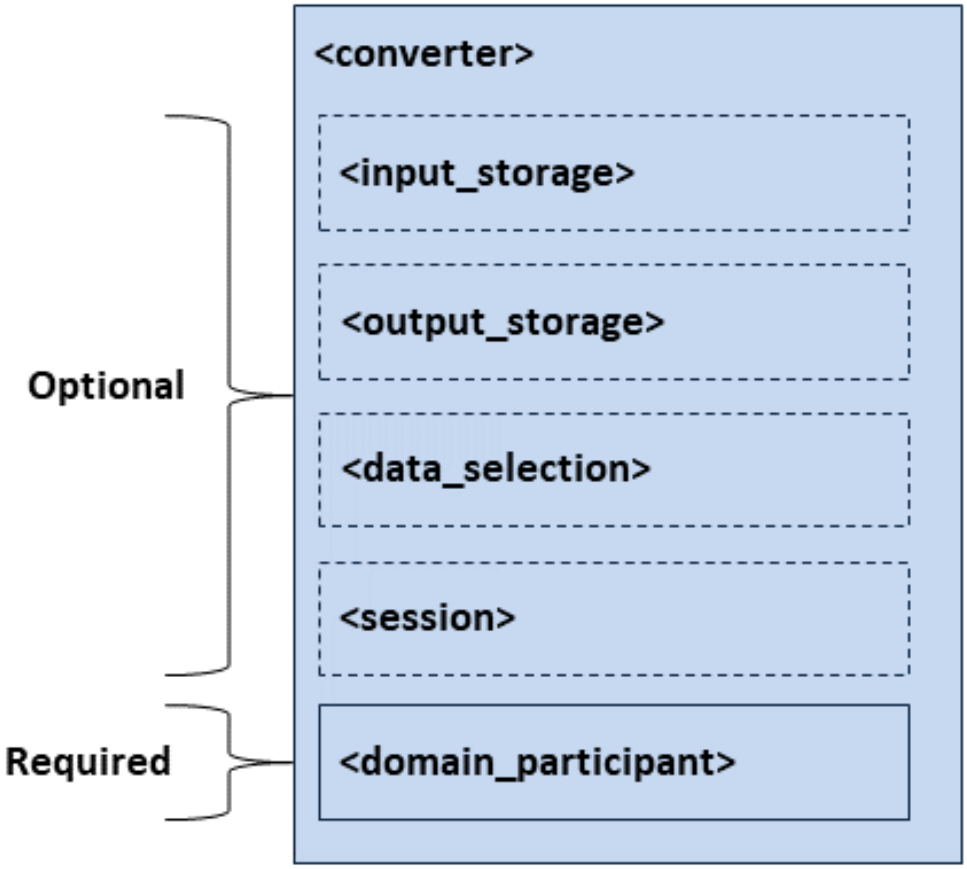


Figure 5.2: Converter Tags in the *Converter* Configuration File

Table 5.4: Input Storage Tags in Converter's Configuration File

Tags within <input_storage>	Description	Multiplicity
<sqlite>	Enables converting data from an SQLite database file. See Section 5.2.8.	0..1
<legacy>	Enables converting data from a legacy version of *Recording Service* to a database usable by this version. See Section 5.2.11.	0..1
<plugin>	Enables converting data from storage using an external library that you specify. See Section 5.2.13.	0..1
<max_samples_per_read>	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 *Converter*, 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.	0..1

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 or implementing your own storage plugin.

Table 5.5: Output Storage Tags in Converter's Configuration File

Tags within <output_storage>	Description	Multiplicity
<sqlite>	Enables writing converted data into an SQLite database file. See Section 5.2.8.	0..1
<plugin>	Enables writing converted data into storage of your choosing, using a plugin library that you specify. See Section 5.2.13.	0..1

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

Tags within <sqlite>	Description	Multiplicity
<fileset> (output only)	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 5.2.9.	0..1
<file> (output only)	Name of file to write converted database file(s) to. Used when creating only a single output file. Default: rti_recorder_default_converted	0..1
<file_suffix> (output only)	Suffix of the output file. Used when creating only a single output file. Default: dat	0..1
<database_dir> (input only)	Directory containing file(s) to convert. Default: Current working directory.	0..1
<storage_format>	Specifies what format the data is stored in. The options are: • [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. • 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.	0..1
<overwrite_policy_kind>	Defines what to do when database files are already present in the current Converter execution directory. **There are two options:** • OVERWRITE: delete old files and replace them with newly created ones. • DO_NOT_TOUCH: do not delete any old files and just exit. Default: OVERWRITE	0..1

Continued on next page

Table 5.6 – continued from previous page

Tags within <sqlite>	Description	Multiplicity

<sql_initialization_string>	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.
	Example:
	`<sql_initialization_string>`
	`DROP INDEX IF EXISTS pingtopic_0_by_src_timestamp;`
	`CREATE INDEX pingtopic_0_by_src_timestamp ON 'PingTopic@0' (SampleInfo_source_timestamp);`
	`</sql_initialization_string>`
	Default: PRAGMA SYNCHRONOUS = OFF; PRAGMA JOURNAL_MODE = MEMORY;
		0..1

5.2.9 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 5.7: Fileset Tags in Converter's Configuration File

Tags within <fileset>	Description	Multiplicity
<workspace_dir>	**Required**. Base directory where *Converter* should output files. Must be different from the input `database_dir` directory. Default: The current working directory	1

Continued on next page

Table 5.7 – continued from previous page

Tags within <fileset>	Description	Multiplicity

<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%.dat`
	`</filename_expression>`
	This will create a series of files starting with test_files_0.dat, 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 continously).
	• Date and time. **Format: %c%.** Date and time representation, locale-dependent. This parameter uses the strftime() parameter %c.
	Default: rti_recorder_default_%auto:0%.dat (auto-numeric starting at zero, unlimited)
		1

<rollover>	Configuration for rolling over the file after a size limit is reached. See Section 5.2.10.	0..1

5.2.10 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 5.8: Rollover in Converter's Configuration File

Tags within <rollover>	Description	Multiplicity
<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

5.2.11 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 5.9: Legacy in the Configuration File

Tags within <legacy>	Description	Multiplicity
<file_path>	The path to a legacy file to be converted. File set and version properties will be obtained automatically from the file itself.	0..1
<domain_mapping>	This tag allows you to link legacy domain names with domain IDs. See Section 5.2.12.	0..1

5.2.12 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 5.10: Domain Mapping in the Configuration File

Tags within <domain_mapping>	Description	Multiplicity
<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..*

5.2.13 Plugin

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

Table 5.11: Storage plugin Tag in the Configuration File

Tags within <plugin>	Description	Multiplicity
<property>	Name/value pairs of properties to pass to a storage plugin.	0 or 1

5.2.14 Data Selection

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

Table 5.12: Data Selection Tags in Converter's Configuration File

Tags within <data_selection>	Description	Multiplicity
<time_range>	Select data to convert from the database, based on start and end times. See Section 5.2.15.	0..1

5.2.15 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 3.5.7).

Table 5.13: Time Range Tags in Converter's Configuration File

Tags within <time_range>	Description	Multiplicity
<begin_time>	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.	0..1
<begin_tag>	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 3.5.6. Default: Start at beginning of the file.	0..1
<end_time>	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.	0..1
<end_tag>	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 3.5.6. Default: Stop at end of file.	0..1

5.2.16 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 5.14: DomainParticipant Tags in Converter's Configuration File

Tags within <domain_participant>	Description	Multiplicity
<domain_id>	The domain ID of tables loaded from the database. This is used to determine which domain IDs to convert.	1
<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..*

5.2.17 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 5.15: Session Tags in Converter's Configuration File

Tags within <session>	Description	Multiplicity
<thread_pool>	Specifies the number of threads used by this session, and the settings used when creating them.	0..1
<topic>	Specifies an individual Topic to convert. See Section 5.2.19. **Attributes:** • `name`: The name of the Topic to convert. • `participant_ref`: Refers to a `domain_participant` tag, that specifies the domain ID to use from the database when converting this Topic. If the parent `<session>` specifies a `default_participant_ref`, this attribute is optional.	0..*

Continued on next page

Table 5.15 – continued from previous page

Tags within <session>	Description	Multiplicity
<topic_group>	Specifies a group of Topics to convert. See Section 5.2.18. **Attributes:** • `name`: The name of the topic group. • `participant_ref`: Refers to a `domain_participant` tag, that specifies the domain ID to use from the database when converting this Topic. If the parent `<session>` specifies a `default_participant_ref`, this attribute is optional.	0..*

5.2.18 Topic Group

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

Table 5.16: Topic Group Tags in Converter's Configuration File

Tags within <topic_group>	Description	Multiplicity

<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

5.2.19 Topic

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

Table 5.17: Topic Tags in Converter's Configuration File

Tags within <topic>	Description	Multiplicity
<topic_name>	The name of the DDS topic to be converted. If this tag is not present, the name attribute of the *<topic>* 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 Converter to fail when found in the topic name attribute.	0..1
<registered_type_name>	The name of the data type that will be converted for this topic. **Required**.	1

<transformation>	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.
	Example:
	`<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>`
		0..1

5.2.20 Converter's Builtin Configuration Details

Converter comes with a builtin 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.dat' as the input file

and 'rti_recorder_default_converted.dat' 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 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.dat</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 3.5.6.

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 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:

```
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
```

Then you can have a `<data_selection>` tag in your XML for *Converter*, after the `<output_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>
```

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

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 builtin SQLite® storage plugin that is part of the service library.
- *RTI Recording Service* Transformation API: *Transformation*s are data-oriented pluggable components that allow you to perform conversions of the representation and content of the data that goes through *Recording Service*. This module offers a set of pluggable APIs to develop a custom *Transformation*s, which you can use through shared libraries or through the Service API.

Table 6.1 shows which modules are available for each API, along with links to the API documentation.

Table 6.1: API Documentation for the SDK

Language API	Available Modules
RTI Recording Service C API	• Storage • Transformation (see RTI Routing Service C API)
RTI Recording Service C++ API	• Service • Storage • Transformation (see RTI Routing Service C++ API)

Chapter 7

Common Infrastructure

7.1 Application Resource Model

RTI services are described through a *hierarchical application resource model*. In this model, an application is composed of a set of *Resources*, each representing a particular component within the application. *Resources* have a parent-child relationship. Figure 7.1 shows a general view of this concept.

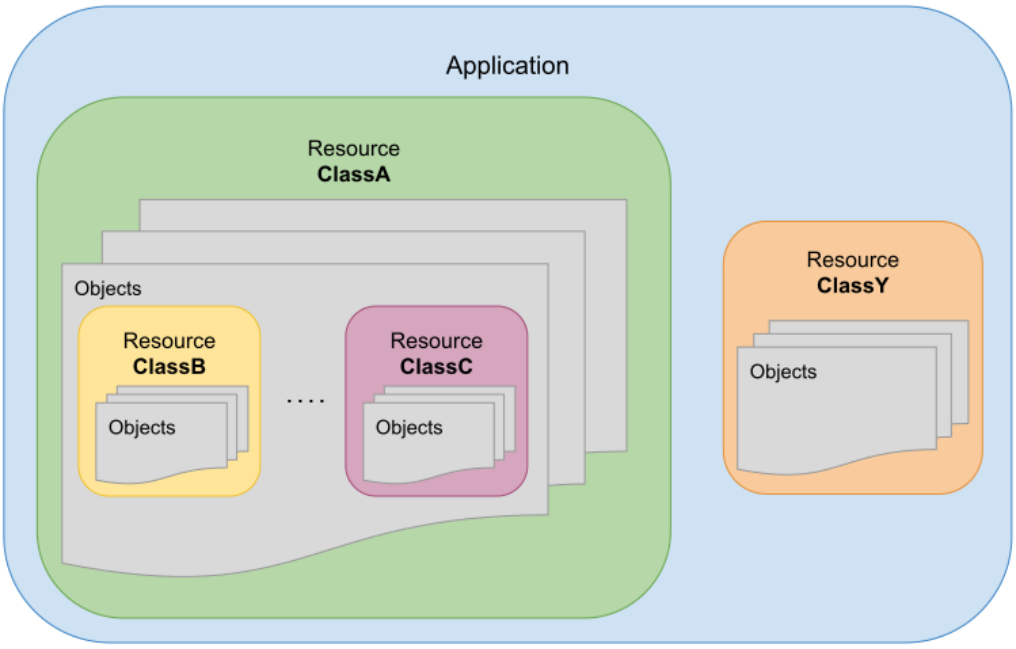


Figure 7.1: Application modeled as a set of related Resources

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

- **Simple**–Represents a single object.

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

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

7.1.1 Example: Simple Resource Model of a Connext DDS Application

Figure 7.2 depicts a UML class diagram to provide a generic resource model for *Connext DDS* applications.

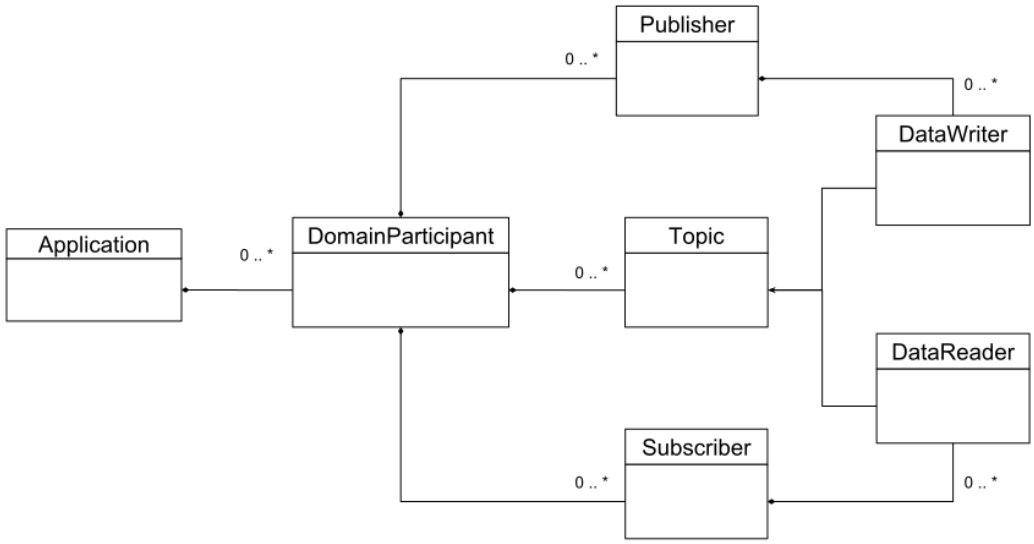


Figure 7.2: Connext DDS application resource model

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.

7.1.2 Resource Identifiers

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

$$/resource_id_1/resource_id_2.../resource_id_N$$

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

1. Fixed and specified by the resource model of the parent Resource class.
2. Given by the user of the application. This is the case where the parent resource is a collection in which the user can insert objects, providing a name for each of them.

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

$$\boxed{/\text{collection_id}_1/\text{resource_id}_1/\text{resource_id}_2}$$

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

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

The resource identifier format follows these conventions:

- The first character is `/`, which represents the root resource and parent of all the available resources across the applications.
- A collection identifier is defined in lower `snake_case`, and it is always specified by the resource class.
- A simple resource identifier is defined in `camelCase` (lower and upper) and may be specified by both the resource class or the user.

Escaped Identifiers

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

$$\boxed{/\text{"escaped_identifier"}}$$

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

For example, the following full resource path:

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

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

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

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

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

becomes

```
/routing_service/"A/B"

/routing_service/"A:B"
```

in the resource identifier.

Example: Resource Identifiers of a Generic Connext DDS Application

Consider the *Connext DDS* application resource model in Section 7.1.1. The following resource identifier addresses a concrete *DomainParticipant* named "MyParticipant" in a given application:

```
/domain_participants/MyParticipant
```

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

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

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

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

Example: Resource Identifiers Generated from XML Entity Model

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

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

The resulting generated resource identifiers will look as follows:

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

7.2 Remote Administration Platform

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

Note: Remote administration of RTI services requires an understanding of the *application resource*

model. We recommend that you read *Application Resource Model* (Section 7.1) before continuing with this section.

The *RTI Remote Administration Platform* addresses two areas:

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

The combination of these two areas provides the general view of the *RTI Remote Administration Platform*, as shown in Figure 7.3. 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*.

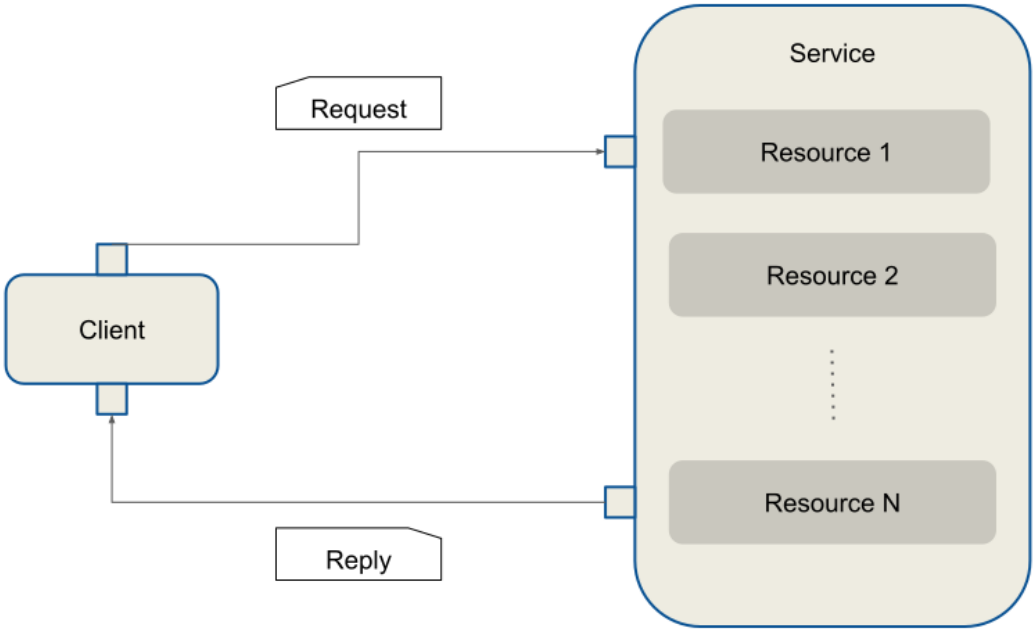


Figure 7.3: General View of the *RTI Remote Administration Platform* Architecture

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

7.2.1 Remote Interface

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

Table 7.1: REST Interface

Element	Description
REST Request	**[method] + [resource_identifier] + [body]** • `method`: Specifies the action to be performed on a service resource. There is only a small subset of methods, known as *standard methods* (see Section 7.2.1). • `resource_identifier`: Addresses a concrete service resource. Each concrete service has its own set of resources (see Section 7.1.2). • `body`: Optional request data that contains necessary information to complete the operation.
REST Reply	**[return code] + [body]** • `return code`: Integer indicating the result of the operation. • `body`: Optional reply data that contains information associated with the processing of the request.

Standard Methods

The *RTI Remote Administration Platform* defines the methods listed in Table 7.2.

Table 7.2: Standard Methods

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

Custom Methods

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

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

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

It is up to each service implementation to define which custom methods are available and on what

resources they apply. Custom methods follow these conventions:

- They are invoked through the `UPDATE` standard method.
- They are named using lower snake_case.
- They may use the request body and reply body if necessary.

Example: Database Rollover

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

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

7.2.2 Communication

The information exchange between client and server is based on the DDS request-reply pattern, as shown in Figure 7.4. The client maps to a *Requester*, whereas the server maps to a *Replier*.

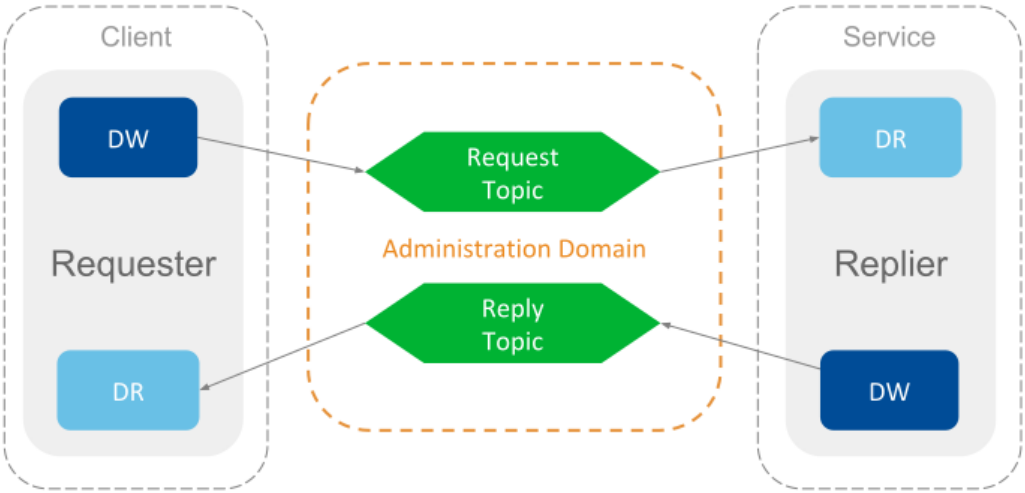


Figure 7.4: Communication in *RTI Remote Administration Platform* is Based on DDS Request-Reply

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

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

The definition of these topics is shown in Table 7.3:

Table 7.3: Remote Administration *Topics*

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

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

Listing 7.1: `CommandRequest` Type

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

Table 7.4: `CommandRequest`

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

Listing 7.2: `CommandReply` Type

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

Table 7.5: `CommandReply`

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

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

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

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

Reply Sequence

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

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

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

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

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

Example: Accessing from Connext DDS Application

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

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

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

int main(int, char *[]) {

    try {

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

    return 0;
}
```

7.2.3 Common Operations

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

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

The following sections describe each of these common operations.

Create Resource

`CREATE [resource_identifier]`

Creates a resource object from its configuration in XML representation.

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

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

Request body

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

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

- Example `file://` request body:

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

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

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

Get Resource

`GET [resource_identifier]`

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

Request body

- Empty.

Reply body

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

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

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.

Update Resource

`UPDATE [resource_identifier]`

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

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

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

Request body

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

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

- Example `file://` request body:

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

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

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

Set Resource State

`UPDATE [resource_identifier]/state`

Sends a state change request to the specified resource object.

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

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

Request body

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

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

- The specified resource identifier does not exist.
- The target request is invalid.

- The resource object reported an error while performing the state transition.

Delete Resource

`DELETE [resource_identifier]`

Deletes the specified resource object.

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

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

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

Request body

- Empty.

Reply body

- Empty.

Return codes

The operation may return a reply with error if:

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

7.3 Monitoring Distribution Platform

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

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

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

These three *Topic*s are shared across all services for the distribution of the monitoring information. Table 7.6 provides a summary of these topics.

Table 7.6: Monitoring Distribution *Topic*s

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

Figure 7.5 shows the mapping of the monitoring information into the distribution *Topic*s. A distribution *Topic* **is keyed** on service resources categorized as *keyed Resources*. These are resources whose related monitoring information is provided as an instance on the distribution *Topic*.

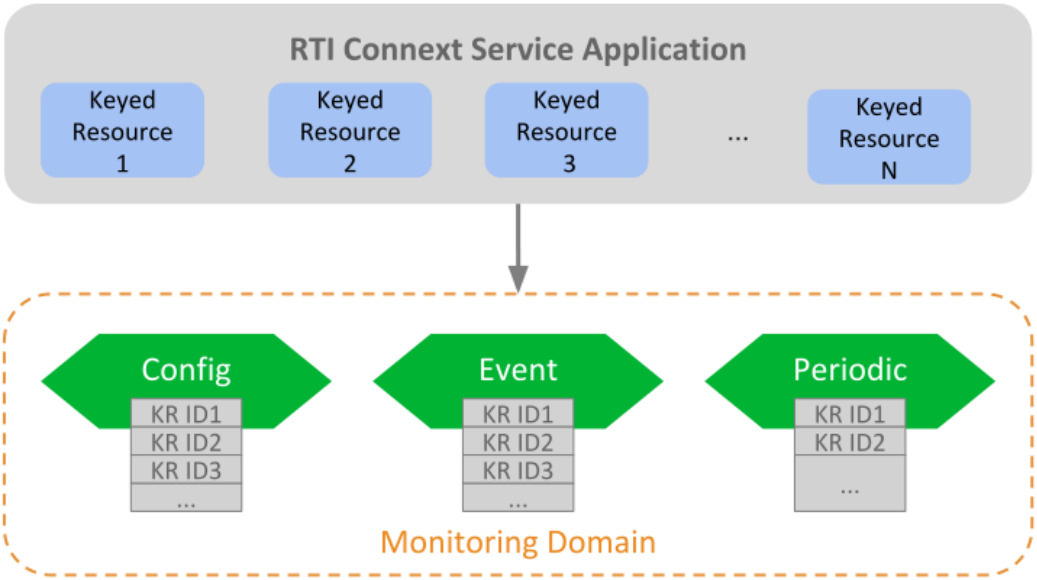


Figure 7.5: Monitoring Distribution *Topic*s of *RTI* Services

7.3.1 Distribution Topic Definition

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

The definition of a distribution *Topic* is shown in Figure 7.6.

Keyed Resource Base Type Fields

This is the base type of all distribution *Topic*s and consists of two fields:

- `object_guid`: Key field. It represents a 16-byte sequence that uniquely identifies a *Keyed Resource* across all the available services in the monitoring domain. Hence, the associated instance handle key hash will be the same for all distribution *Topic*s, allowing easy correlation

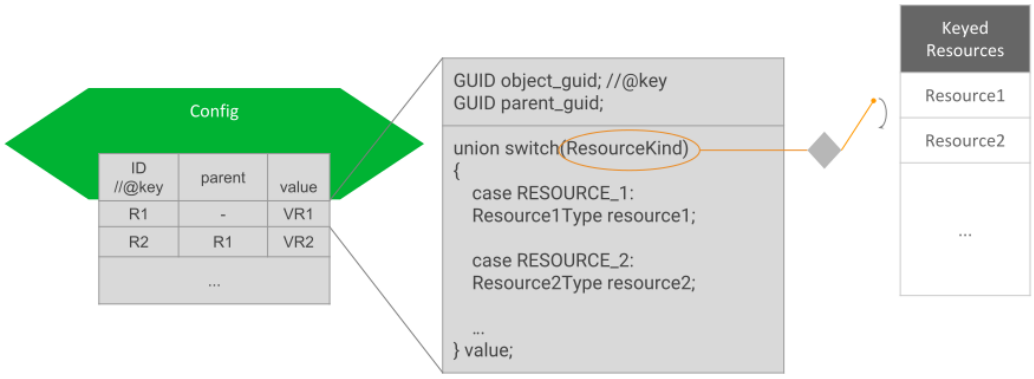


Figure 7.6: Monitoring Distribution *Topic* Definition

of a resource. It will also facilitate, as we will discuss later, easy instance data manipulation in a *DataReader*.

- `parent_guid`: It contains the object GUID of the parent resource. This field will be set to all zeros if the object is a top-level resource thus with no parent.

This base type, `KeyedResource`, is defined in `[NDDSHOME]/resource/idl/ServiceCommon.idl`.

Resource-Specific Type Fields

This is the type that conveys monitoring information for a concrete resource object. Since a distribution *Topic* is responsible for providing information about different resource classes, the resource-specific type consists of a single field that is a **Union of all the possible representations** for the keyed resources that provide that on the topic.

As expected, there must be consistency between the two parts of the distribution topic type. That is, a sample for a concrete resource object must contain the resource-specific union discriminator corresponding to the resource object's class.

Example: Monitoring of Generic Application

Assume a generic application that provides monitoring information about the modes of transports `Car`, `Boat` and `Plane`. Each mode is mapped to a keyed resource, each with a custom type that contains metrics specific to each class.

The monitoring distribution *Topic* top-level type, `TransportModeDistribution`, would be defined as follows, using IDL v4 notation:

```
#include "ServiceCommon.idl"

@nested
struct CarType {
    float speed;
    String color;
    String plate_number;
```

(continues on next page)

(continued from previous page)

```
};

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

@nested
struct PlaneType {
    float ground_speed;
    int32 air_track;
};

enum TransportModeKind {
    CAR_TRANSPORT_MODE,
    BOAT_TRANSPORT_MODE,
    PLANE_TRANSPORT_MODE
};

@nested
union TransportModeUnion switch (TransportModeKind) {
    case CAR_TRANSPORT_MODE:
    CarType car;

    case BOAT_TRANSPORT_MODE:
    BoatType boat;

    case PLANE_TRANSPORT_MODE:
    PlaneType plane;
}

struct TransportModeDistribution : KeyedResource {
    TransportModeUnion value;

};
```

Assume now that in the monitoring domain there are three resource objects, one for each resource class: a `Car` object 'CarA', a `Boat` object 'Boat1', and a `Plane` object 'PlaneX'. They all have unique resource GUIDs and each object represents an instance in the distribution *Topic*. The table shows the example of potential sample values:

Table 7.7: Samples in TransportModeDistribution *Topic*

	CarA	Boat1	PlaneX
`object_guid`	0x0C	0xAB	0xf2
`parent_guid`	0x00	0x00	0x00
`value` discriminator	CAR_TRANSPORT_MODE	BOAT_TRANSPORT_MODE	PLANE_TRANSPORT_MODE

7.3.2 DDS Entities

RTI services allow you to distribute monitoring information in any domain. For that, they create the following DDS entities:

- A *DomainParticipant* on the monitoring domain.
- A single *Publisher* for all *DataWriter*s.
- A *DataWriter* for each distribution *Topic*.

A service will create these entities with default QoS or otherwise the corresponding service user's manual will specify the actual values. Services allow you to customize the QoS of the DDS entities, typically in the service monitoring configuration under the `<monitoring>` tag. You will need to refer to each service's user's manual.

7.3.3 Monitoring Metrics Publication

How services publish monitoring samples depends on the distribution *Topic*.

Configuration Distribution Topic

There are two events that cause the publication of samples in this topic:

- As soon as a *Resource* object is created. This event generates the first sample in the *Topic* for the resource object just created. Since these first samples are published as resources are created, it is guaranteed to be in hierarchical order; that is, the sample for a parent *Resource* is published before its children. When *Resources* are created depends on the service. Typically, *Resources* are created on service startup. Other cases include manual creation (e.g., through remote administration) or external event-driven creation (e.g., discovery of matching streams, in the case of *AutoRoute* in *Routing Service*).
- On *Resource* object update. This event occurs when the properties of the object change due to a set or update operation (e.g., through remote administration).

Event Distribution Topic

Services publish samples in this *Topic* in reaction to an internal event, such as a *Resource* state change. Which events and their associated information and when they occur is highly dependent on concrete service implementations.

Periodic Distribution Topic

Samples in this *Topic* are published periodically, according to a fixed configurable period. The metrics provided in this *Topic* are generated in two different ways:

- As a snapshot of the current value, taken at the publication time (e.g., current number of matching *DataReaders*). This represents a simple case and the metric is typically represented with an adequate primitive member.
- As a *statistic variable* generated from a set of discreet measurements, obtained periodically. This represents a *continous* flow of metrics, represented with the `StatisticVariable` type (see Section 7.3.4).

There are two activities involved in the generation of the statistic variables: Calculation and Publication. All the configuration elements for these activities are available under the `<monitoring>` tag.

Calculation

The instrumented service periodically performs measurements on the metric. This activity is also known as *sampling* (don't confuse with data samples). The frequency of the measurements can be configured with the tag `<statistics_sampling_period>`. As a general recommendation, the sampling period should be a few times smaller than the publication period. A small sampling period provides more accurate statistics generation at the expense of increasing memory and CPU consumption.

Publication

The service periodically publishes a data sample containing a snapshot of the statistics generated during the calculation phase. The publication period can be configured with the tag `<statistics_publication_period>`.The value of a statistic variable corresponds to the time window of a publication period.

7.3.4 Monitoring Metrics Reference

This section describes the types used as common metrics across services. All the type definitions listed here are in `[NDDSHOME]/resource/idl/ServiceCommon.idl`.

Statistic Variable

Listing 7.3: Statistics

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

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

Table 7.8: `StatisticMetrics`

Field Name	Description
period_ms	Period in milliseconds at which the metrics are published.
count	Sum of all the measurement values obtained during the publication period.
mean	Arithmetic mean of all the measurement values during publication period. For aggregated metrics, this value is the mean of all the aggregated metrics means.
min	Minimum of all the measurement values during publication period. For aggregated metrics, this value is the minimum of all the aggregated metrics minimums.
max	Maximum of all the measurement values during publication period. For aggregated metrics, this value is the maximum of all the aggregated metrics minimums.
std_dev	Standard deviation of all the measurement values during publication period. For aggregated metrics, this value is the standard deviation of all the aggregated metrics minimums.

Host Metrics

Listing 7.4: Host Types

```
@appendable @nested
struct HostPeriodic {
    @optional StatisticVariable cpu_usage_percentage;
    @optional StatisticVariable free_memory_kb;
    @optional StatisticVariable free_swap_memory_kb;
    int32 uptime_sec;
};
```

(continues on next page)

(continued from previous page)

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

Table 7.9: `HostConfig`

Field Name	Description
name	Name of the host where the service is running.
id	ID of the host where the service is running.
total_memory_kb	Total memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.
total_swap_memory_kb	Total swap memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.

Table 7.10: `HostPeriodic`

Field Name	Description
cpu_usage_percentage	Statistic variable that provides the global percentage of CPU usage on the host where the service is running. Availability of this value is platform dependent.
free_memory_kb	Statistic variable that provides the amount of free memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.
free_wap_memory_kb	Statistic variable that provides the amount of free swap memory in KiloBytes of the host where the service is running. Availability of this value is platform dependent.
uptime_sec	Time in seconds elapsed since the host on which the running service started. Availability of this value is platform dependent.

Process Metrics

Listing 7.5: Process Types

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

(continues on next page)

(continued from previous page)

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

Table 7.11: `ProcessConfig`

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

Table 7.12: `ProcessPeriodic`

Field Name	Description
cpu_usage_percentage	Statistic variable that provides the percentage of CPU usage of the process where the service is running. The field `count` of the variable contains the total CPU time that the processor spent during the publication period. Availability of this value is platform dependent.
physical_memory_kb	Statistic variable that provides the physical memory utilization in KiloBytes of the process where the service is running. Availability of this value is platform dependent.
total_memory_kb	Statistic variable that provides the virtual memory utilization in KiloBytes of the process where the service is running. Availability of this value is platform dependent.
uptime_sec	Time in seconds elapsed since the running service process started. Availability of this value is platform dependent.

Base Entity Resource Metrics

Listing 7.6: Base Entity Types

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

Table 7.13: `EntityConfig`

Field Name	Description
resource_id	String representation of the resource identifier associated with the entity resource.
configuration	String representation of the XML configuration of the entity resource. The XML contains only children elements that are not entity resources.

Table 7.14: `EntityEvent`

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

Network Performance Metrics

Listing 7.7: Network Performance Type

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

Table 7.15: `NetworkPerformance`

Field Name	Description
samples_per_sec	Statistic variable that provides information about the number of samples processed (received or sent) per second.
bytes_per_sec	Statistic variable that provides information about the number of bytes processed (received or sent) per second.
latency_millisec	Statistic variable that provides information about the latency in milliseconds for the data processed. The latency in a refers to the total time elapsed during the associated processing of the data, which depends on the type of application.

7.4 Plugin Management

Some RTI services allows for custom behavior through the use of *pluggable* components or *plugins* . The type of plugins is described in Section 6. A plugin is represented as a top-level service-owned object whose main role is a factory of other pluggable components, which themselves are responsible for providing the user-defined behavior.

Figure 7.7 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 7.7 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.

7.4.1 Shared Library

A plugin object is instantiated through a *create function*, which is included and addressable as part of a shared library. This function is also known as the *entry point* and each RTI service defines the signature for each plugin class. This method requires specifying the path to the shared library and the name of the entry point (see Section 7.4.1). The *Service* loads the library the first time an instance of the plugin is needed (lazy initialization) and looks up the specified entry point symbol in the loaded library. The *Service* will always delete the plugin on *Service* stop.

This is the only method suitable when a RTI service is deployed through an already linked executable, such as the shipped command-line executable (Section 3.1).

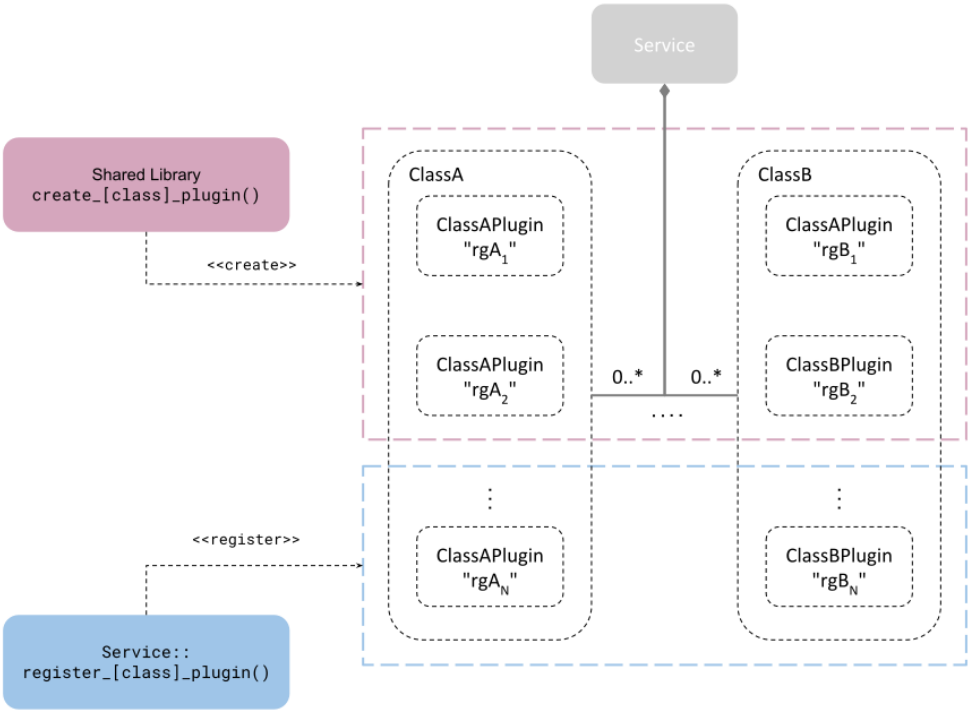


Figure 7.7: Plugin object management

The plugin lifecycle is as follows:

1. After start, *Service* creates a plugin object for each registered plugin in the configuration. The plugin object is instantiated through the shared library entry point, specified in the configuration.
2. *Service* calls operations on the plugin objects as needed and keeps them alive while the *Service* remains started.
3. During stop, the *Service* deletes each plugin object by calling the class abstract deleter.

Configuration

An RTI service configures the pluggable components within the `<plugin_library>` tag. RTI services that support plugins will define a set of tags within in the form:

- `<[class]_plugin>` for C/C++ plugins
- `<java_[class]_plugin>` for Java plugins

where `[class]` refers to the name of the plugin class. For example, in *Routing Service* an available tag is `<adapter_plugin>`.

The definition of these tags is the same regardless of the plugin class and is described in the tables below.

Table 7.16 and Table 7.17 describe the configuration of each different plugin language.

Table 7.16: Configuration tags for C/C++ plugins.

Tags within <[class]_plugin>	Description	Multiplicity
<dll>	Shared library containing the implementation of the adapter plugin. This tag may specify the exact path (absolute or relative) of the file (for example, **lib/libmyplugin.so**) or a general name (no file extension). If no extension is provided, the path will be completed based on the running platform. For example, assuming a value for this tag of **dir/myplugin**: • UNIX-based systems: **dir/libmyplugin.so** • Windows systems: **dir/myplugin.dll** If the library specified in this tag cannot be loaded (because the environment library path is not pointing to the path where the library is located), *Routing Service* will look for the library in the following locations, in this order: • [**plugin_search_path**]: Provided as part of the service parameters (see Section 3.1) • [**executable_dir**]: Directory where the executable lives	1
<create_function>	Entry point. This tag must contain the name of the function used to create the plugin instance. The function symbol must be present in the shared library specified in `<dll>`	1

<property>	A sequence of name-value string pairs that allow you to configure the plugin instance.
	Example:
	`<property>`
	`  <value>`
	`  <element>`
	`  <name>myplugin.user_name</name>`
	`  <value>myusername</value>`
	`  </element>`
	`  </value>`
	`</property>`
		0..1

Table 7.17: Configuration tags for Java plugins

Tags within <java_[class]_plugin>	Description	Multi-plicity
<class_name>	Name of the class that implements the plugin. For example: com.myplugins.CustomPlugin The classpath required to run the Java plugin must be part of the RTI service JVM configuration. See the `<jvm>` tag within the specific service configuration for additional information on JVM creation and configuration.	1

<property>	A sequence of name-value string pairs that allow you to configure the plugin instance.
	Example:
	`<property>`
	`  <value>`
	`  <element>`
	`  <name>myplugin.user_name</name>`
	`  <value>myusername</value>`
	`  </element>`
	`  </value>`
	`</property>`
		0..1

7.4.2 Service API

The user provides the plugin object via the Service API, through one of the available `attach_[class]_plugin()` operations. Upon successful return of the operation, the *Service* takes ownership of the plugin object and will delete it on *Service* stop.

Plugin lifecycle goes as follows:

1. User instantiates plugin objects and provides them to the *Service* through the `attach_[class]_plugin()` operation. This is allowed only before the *Service* starts.
2. After start, the *Service* becomes the owner of the registered plugin objects, calls operations on the plugin objects as needed, and keeps them alive while the *Service* remains started.
3. On stop, the *Service* deletes each registered plugin object by calling the class abstract deleter.

Chapter 8

Release Notes

8.1 Supported Platforms

RTI® Recording Service is supported on the platforms listed in Table 8.1.

It can also be deployed as a C++ library linked into your application. This is true for all platforms in Table 8.1.

Table 8.1: Supported Platforms in 6.0.1

Platform	Operating System
Linux	All Linux platforms in the *RTI Connext DDS Core Libraries Release Notes* with the same version number, except NI Linux 3 on Arm® v7 and Wind River® Linux 7 on x64.
macOS®	All macOS platforms in the *RTI Connext DDS Core Libraries Release Notes* with the same version number.
QNX®	All QNX Neutrino® 7.0 platforms in the *RTI Connext DDS Core Libraries Release Notes* with the same version number.
Windows®	All Windows® platforms in the *RTI Connext DDS Core Libraries Release Notes* with the same version number.

Recording Service is also supported on the platforms in Table 8.2; 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 8.2: Custom Platforms

Platform	Operating System
Linux	Yocto Project 2.5 on Arm v7

8.2 Compatibility

For backward compatibility information between *Recording Service* 6.0.1 and previous releases, please see the *Migration Guide* on the RTI Community portal.

8.3 What's New in 6.0.1

8.3.1 New platforms

Recording Service, *Replay Service*, and *Converter* now include support for these platforms:

- macOS 10.14 (x64)
- Ubuntu 16.04 LTS (Arm v8 x64)
- QNX 7.0 (x64 and Arm v8 x64)
- Red Hat® Enterprise Linux 8 (x64)
- Windows 10 (x86, x64) with Visual Studio® 2019
- Windows Server 2016 (x86, x64) with Visual Studio 2019
- Yocto Project 2.5 (Arm v7) (custom target platform)

For more information on these platforms, see the *RTI Connext DDS Core Libraries Platform Notes* for this release.

8.3.2 Removed platforms

These platforms are no longer supported:

- macOS 10.11
- Windows 7
- Windows Server 2008 R2

8.3.3 New <topic_name> tag to specify DDS topic name

In the previous version, the `<topic>` tag used the `name` attribute to refer to both its own name and the DDS topic to be recorded, replayed, or converted. This can be problematic when the topic name contains special characters that cannot be used in an XML attribute. It can also be problematic when the name uses a character or string also used by *Recording Service*, *Replay Service*, or *Converter* for other purposes. For example, the `::` string is the separator for administration and monitoring names within the tool.

The new `<topic_name>` tag, which is not required but encouraged, will be used to obtain the DDS topic name if present, instead of using the `<topic>` tag's `name` attribute. You can fall back to the previous behavior by not specifying `<topic_name>`, in which case the `name` attribute will be used instead.

8.3.4 Replay fidelity improved with large data

Replay Service's performance has been improved when working with large data. Reading the samples to be replayed has been decoupled from the process of writing the samples to DDS. This change provides better fidelity by reducing the delay in writing.

8.3.5 Improved performance when creating legacy database stream readers

This release provides better performance when creating legacy database stream readers in *Replay Service* or *Converter* (that is, when using *Replay Service* or *Converter* with the `<legacy>` storage plugin).

8.3.6 Support for remotely triggered flush of user-data to disk

This release incorporates a new operation mode in *Recording Service*, called *buffering mode*. It also includes a new remote administration operation, *flush*.

Buffering mode allows you to start *Recording Service* in a listening-only mode, where the service just buffers data but doesn't actually write it to disk automatically.

By using the new remote *flush* operation, you can remotely control exactly when the data is written to permanent storage. Upon receiving this remote command, the service will command all the Topics and Topic Groups to store all the buffered data.

For more information about these features, see the User's Manual, section 3.1.4, Controlling Recording Service's Operation Mode.

8.4 What's Fixed in 6.0.1

8.4.1 Legacy deserialized database table couldn't be replayed or converted if it contained compact byte sequences or arrays

Replaying or converting a legacy (version 5.3.1 or older) database stored in deserialized format failed if the database was recorded with octet/char sequences/arrays stored as blobs (that is, in compact mode). For example:

```
struct LargeByteSeqType {
  sequence<octet, 65530> large_byte_seq;
  ...;
};
```

If the database stored `large_byte_seq` as a BLOB column, then *Replay Service* or *Converter* failed and complained with the following message:

```
exception:[DRT_DynamicType_expand_sequence_v3@3379]:Sequence length
greater than built-in max
```

This problem has been resolved.

[RTI Issue ID RECORD-957]

8.4.2 Segmentation fault in Replay Service or Converter if recorded topics had "::" in their names

This issue happened when a DDS topic contained `::` as part of its name. *Replay Service* or *Converter* would hit a segmentation fault when trying to process that topic's table. This issue has been resolved.

[RTI Issue ID RECORD-959]

8.4.3 Detection of termination in Converter fixed

When running *Converter*, it was possible that some topics weren't converted before the application terminated. This was due to a poor termination detection schema. This problem has been resolved.

[RTI Issue ID RECORD-970]

8.4.4 QoS Topic Filters were not supported

Recording Service entities previously ignored QoS Topic Filters. This problem has been resolved. The entities listed below will obtain the QoS within the selected profile having a matching topic filter based on the actual name of the Topic they are associated with.

- Topic (both *Recording Service* and *Replay Service*)
- TopicGroup's (both *Recording Service* and *Replay Service*)
- Remote Administration Replier
- Monitoring DataWriters

For example, consider the following QoS profile, which uses topic filters.

```
<qos_library name="QosLib">
    <qos_profile name="QosProfile"">
        <datawriter_qos>...</datawriter_qos>
        <datawriter_qos topic_filter="MyTopic">
                ...
        </datawriter_qos>
        <datareader_qos topic_filter="MyTopic">
                 ...
        </datareader_qos>
        <datawriter_qos topic_filter="rti/service/admin/command_reply">
                 ...
        </datawriter_qos>
        <datareader_qos topic_filter="rti/service/admin/command_request">
                 ...
        </datareader_qos>
        <datawriter_qos topic_filter="rti/service/monitoring/periodic">
                 ...
        </datawriter_qos>
    </qos_profile>
</qos_library>
```

And the following *Recording Service* configuration:

```
<recording_service name="MyRecorder">
    <administration>
        <writer_qos  base_name="QosLib::QosProfile" />
        <reader_qos base_name="QosLib::QosProfile" />
    </administration>
    <monitoring>
        <writer_qos  base_name="QosLib::QosProfile" />
    </monitoring>

    ....
           <topic>
               <topic_name>MyTopic>
               <reader_qos  base_name="QosLib::QosProfile" />
           </topic_route>

           <topic_group>
               <reader_qos  base_name="QosLib::QosProfile" />
           </topic_group >
    ...

</recording_service>
```

The corresponding DDS entities below will be created with the QoS that matches the topic filter based on the topic name:

DDS Entity	QoS from QosLib::QosProfile	Topic filter
Topic's DataReader	reader QoS	MyTopic
TopicGroup's DataReader for topic name=MyTopic	reader QoS	MyTopic
TopicGroup's DataReader for topic name=Other	reader QoS	NULL
Administration Replier's DataReader	reader QoS	rti/service/admin/command_request
Administration Replier's DataWriter	writer QoS	rti/service/admin/command_reply
Monitoring Periodic DataWriter	writer QoS	rti/service/monitoring/periodic
Monitoring Config and Event DataWriter	writer QoS	NULL

[RTI Issue ID RECORD-1008]

8.4.5 Execution directories overwritten incorrectly

Using the `<execution_dir_expression>` tag (under `<fileset>`) with the autonumeric format (for example, `%auto:0-9%`) did not always give the expected results. *Recording Service* always began writing in the first directory (0), instead of looking for the first *unused* directory within the specified range (0-9).

This problem has been resolved. Now *Recording Service* will start writing in the first available execution directory, unless the `<overwrite_policy_kind>` tag (under `sqlite>`) is set to `DO_NOT_TOUCH`.

[RTI Issue ID RECORD-1010]

8.4.6 Segmentation fault in Recording Service if <time_limit> didn't have all its child properties

When using *Recording Service's* `<time_limit>` tag, if any of the child tags were missing, the service would crash. For example, the following XML code is missing the <days> tag, which caused *Recording Service* to crash.

```
<fileset>
    <workspace_dir>output_dir</workspace_dir>
    <execution_dir_expression></execution_dir_expression>
    <filename_expression>file_rollover_%auto:0-9%.dat
    </filename_expression>
    <rollover>
        <enabled>true</enabled>
        <time_limit>
            <hours>0</hours>
            <minutes>0</minutes>
            <seconds>5</seconds>
        </time_limit>
  </rollover>
</fileset>
```

This problem has been resolved.

[RTI Issue ID RECORD-1030]

8.4.7 Possible race condition in Replay Service when looping enabled and pause/resume issued

Sending remote pause/resume commands to *Replay Service* may have caused a race condition if looping was enabled. When this condition occurred, *Replay Service* stopped sending samples because of a corruption in the internal timers. This problem has been resolved.

[RTI Issue ID RECORD-1036]

8.4.8 Possible segmentation fault when replaying or converting legacy database with compressed octet/char sequences

Replaying or converting a legacy database with a table containing a sequence of bytes (chars or octets) stored as a blob (that is, compressed mode) may have caused a segmentation fault. The error occurred when the application tried to set the value of the sequence in the Dynamic Data object. This problem has been resolved.

[RTI Issue ID RECORD-1037]

8.4.9 Missing shutdown messages

Recording Service did not show any messages when shutting down. This problem has been resolved. Now *Recording Service* shows the following message when it begins its shutdown process:

```
Stopping RTI Recording Service
```

Once *Recording Service* has stopped, it will show the next message:

```
RTI Recording Service stopped
```

[RTI Issue ID RECORD-1054]

8.4.10 XML attribute values were overwritten if element set with variable

If an XML element used a variable to specify its value, any contained attributes were incorrectly overwritten with the value of that variable.

For example, consider this XML:

```
<file_size_limit unit="MEGABYTES">$(MAX_SIZE)</file_size_limit>
```

The `unit` attribute was overwritten with the value of the `MAX_SIZE` environment variable. So if MAX_SIZE = 100, the result was:

```
<file_size_limit unit="100">100</file_size_limit>
```

This incorrect behavior applied to any XML element. This problem has been resolved.

[RTI Issue ID RECORD-1061]

8.4.11 Segmentation fault when no valid configuration file found

If *Recording Service* could not find a valid configuration file to load, this caused a segmentation fault. This may have occurred when:

- The default `RTI_RECORDING_SERVICE.xml` file could not be found. This can happen if it has accidentally been renamed or the *Recording Service* executable has been moved.
- The default user configuration `USER_RECORDING_SERVICE.xml` could not be found in the current working directory.
- No other means to obtain the XML configuration were provided, such as using the `-cfgFile` command-line option, or specifying the configuration file or strings property if using *Recording Service* as a library.

This problem has been resolved. Now you will see an error message but no segmentation fault.

[RTI Issue ID RECORD-1078]

8.4.12 Possible segmentation fault when converting legacy database to CDR_AUTO format

When converting a legacy (pre-6.0.0) database to the new database format, if the exported sample format used was XCDR_AUTO, there may have been cases, depending on the types and actual samples exported, where *Converter* threw a segmentation fault when shutting down. This problem has been resolved.

[RTI Issue ID RECORD-1115]

8.5 Previous Releases

8.5.1 What's New in 6.0.0

Recording Service, Replay Service, and Converter are completely new in this release. They have been entirely rewritten, with these goals in mind:

- Improving performance
- Simplifying configuration
- Becoming more consistent with other infrastructure services
- Supporting new use cases

There are a few limitations compared to the prior version:

- Does not support Recording Console, however, *RTI Admin Console* provides some of the functionality.
- Does not yet support replaying in the original partition.
- Does not include a remote shell application, but that is provided as an example.

Avoiding gaps in Replay when using time ranges

Replay Service used to extend the start timestamp to include relevant discovery data in the time range. While this allowed for the discovery of topics whose discovery events may have been out of the specified time range, it could also cause gaps when the discovery data and target data to be replayed were separated in time. To avoid these gaps, *Replay Service* no longer internally modifies the start timestamp defined by the user. All discovery data prior to the defined start timestamp will be discovered instantaneously.

8.5.2 What's Fixed in 6.0.0

Pluggable storage for Converter discarded plugin properties specified in XML

When using the pluggable storage APIs with *Converter*, the properties passed to the plugin in the input_storage or output_storage tags (as shown below) were discarded. This problem has been resolved. Now the properties are passed to the plugin.

```
<input_storage>
    <plugin plugin_name="StorageLib::StoragePlugin">
```

(continues on next page)

(continued from previous page)

```
        <property>
             <value>
                <element>
                    <name>MyProperty</name>
                    <value>MyPropertyValue</value>
                </element>
            </value>
        </property>
    </plugin>
</input_storage>
```

[RTI Issue ID RECNEXTGEN-412]

8.6 Known Issues

8.6.1 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:

```
[/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 `c:\\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.

8.6.2 Segmentation fault after discovering application that uses UserData QoS Policy

If *Recording Service* discovers an application that has at least one entity (*DomainParticipant*, *DataReader*, or *DataWriter*) that uses the UserData QoS Policy, the behavior is undefined and will likely result in a segmentation fault.

[RTI Issue ID RECORD-1125]

Chapter 9

Copyrights

© 2019 Real-Time Innovations, Inc. All rights reserved. Printed in U.S.A. First printing. January 2019.

Trademarks

RTI, Real-Time Innovations, Connext, NDDS, the RTI logo, 1RTI 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/).

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/

© 2019 RTI