RTI Routing Service

Generated by Doxygen 1.9.3

Chapter 1

RTI Routing Service

RTI Routing Service∗ supports the integration and scaling of disparate and geographically dispersed systems. Using off-the-shelf or custom developed plugins, *Routing Service* supports the definition of data transformations and adapters that can interface to multiple protocols in addition to DDS, such as MQTT or legacy code written to the network socket API.

You can learn how to use *Routing Service* through the `RTI Routing Service User's Manual`.

A reference to the API can be found here:

- **RTI Routing Service Adapter API** (p. 13)
- **RTI Routing Service Processor API** (p. 19)
- **RTI Routing Library API** (p. 24)
- **RTI Routing Service Transformation API** (p. 26)

1.1 Feedback and Support for this Release

For more information, visit our knowledge base (`https://community.rti.com/kb`) to see sample code, general information on RTI Connext, performance information, troubleshooting tips, and technical details.

By its very nature, the knowledge base is continuously evolving and improving. We hope that you will find it helpful. If there are questions that you would like to see addressed or comments you would like to share, please send email to `support@rti.com`. We can only guarantee a response for customers with a current maintenance contract or subscription. To purchase a maintenance contract or subscription, contact your local RTI representative (see `http://www.rti.com/company/contact.html`), send an email request to `sales@rti.com`, or call +1 (408) 990-7400.

Please do not hesitate to contact RTI with questions or comments about this release. We welcome any input on how to improve RTI Routing Service∗ and *RTI Connext DDS* to suit your needs.

Chapter 2

Multi-threading safety

Global rti::routing::adapter::AdapterPlugin::create_connection (p. 30) (StreamReaderListener (p. 108) *input_stream_discovery_listener, StreamReaderListener (p. 108) *output_stream_discovery_listener, const PropertySet &properties)=0

Safe

Global rti::routing::adapter::AdapterPlugin::delete_connection (p. 30) (Connection (p. 31) *connection)=0

Safe

Global rti::routing::adapter::Connection::create_session (p. 32) (const PropertySet &properties)

Safe

Global rti::routing::adapter::Connection::create_stream_reader (p. 35) (Session (p. 99) *session, const StreamInfo (p. 99) &stream_info, const PropertySet &properties, StreamReaderListener (p. 108) *listener)

Partially Safe. This operation is never called concurrently for the same **Connection** (p. 31), even if multiple outputs are configured in different Sessions. It can only be called concurrently for different Connections.

Global rti::routing::adapter::Connection::create_stream_writer (p. 33) (Session (p. 99) *session, const StreamInfo (p. 99) &stream_info, const PropertySet &properties)

Partially Safe. This operation is never called concurrently for the same **Connection** (p. 31), even if multiple outputs are configured in different Sessions. It can only be called concurrently for different Connections.

Global rti::routing::adapter::Connection::delete_session (p. 33) (Session (p. 99) *session)

Safe

Class rti::routing::adapter::DiscoveryStreamReader (p. 37)

Safe All the operations on a **DiscoveryStreamReader** (p. 37) always serialized, no matter from which **Connection** (p. 31) it belongs.

Class rti::routing::adapter::StreamReader (p. 102)

Partially Safe All operations on a concrete **StreamReader** (p. 102) objects are safe and always serialized on a given **Session** (p. 99). Operations on different **StreamReader** (p. 102) objects can be called concurrently if the **StreamReader** (p. 102) objects belong to different Sessions.

Class rti::routing::adapter::StreamWriter (p. 109)

Partially Safe All operations on a concrete **StreamWriter** (p. 109) objects are safe and always serialized on a given **Session** (p. 99). Operations on different **StreamWriter** (p. 109) objects can be called concurrently if the **StreamWriter** (p. 109) objects belong to different Sessions.

Class rti::routing::processor::Processor (p. 56)

Partially Safe All operations on a concrete **Processor** (p. 56) object are safe and always serialized on a given Session. Operations on different **Processor** (p. 56) objects may be called concurrently if they belong to different Routes.

Global rti::routing::processor::ProcessorPlugin::create_processor (p. 64) (Route (p. 67) &route, const rti::routing::PropertySet &properties)=0

Safe

Global rti::routing::processor::ProcessorPlugin::delete_processor (p. 65) (Route (p. 67) &route, Processor (p. 56) *processor)=0

Safe

Global rti::routing::Service::Service (p. 84) (const ServiceProperty (p. 88) &property)

On non-Linux, non-Windows systems (i.e. VxWorks): UNSAFE for multiple threads to simultaneously make the FIRST call to any of the **Service** (p. 83) constructors. Subsequent calls are thread safe. On Windows and Linux systems, these calls are thread-safe.

Global rti::routing::Service::Service (p. 84) (const ServiceProperty (p. 88) &property, HookFunc &shutdown_hook)

On non-Linux, non-Windows systems (i.e. VxWorks): UNSAFE for multiple threads to simultaneously make the FIRST call to any of the **Service** (p. 83) constructors. Subsequent calls are thread safe. On Windows and Linux systems, these calls are thread-safe.

Class rti::routing::transf::Transformation (p. 111)

Partially Safe All operations on a concrete **Transformation** (p. 111) object are safe and always serialized on a given Output. Operations on different Transformations objects may be called concurrently if they belong to different Outputs from different Routes.

Global rti::routing::transf::TransformationPlugin::create_transformation (p. 114) (const rti::routing::TypeInfo (p. 137) &input_type_info, const rti::routing::TypeInfo (p. 137) &output_type_info, const rti::routing::PropertySet &properties)=0

Safe

Global rti::routing::transf::TransformationPlugin::delete_transformation (p. 114) (Transformation (p. 111) *transformation)=0

Safe

Class rti::routing::UpdatableEntity (p. 143)

Safe The **UpdatableEntity::update** (p. 143) operation is called only for one entity at a time. Additionally, no other operations can be made concurrently on a pluggable entity during the **UpdatableEntity::update** (p. 143) call.

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Module Documentation

7.1 RTI Routing Service Adapter API

This module describes the C++ Adapter API.

Data Structures

- class **rti::routing::adapter::AdapterPlugin**

  The top-level plug-in class.
- class **rti::routing::adapter::Connection**

  *A **Connection** (p. 31) provides access to a data domain (such as a DDS domain or a JMS network provider).*
- class **StreamReaderListener**

  *Listener representation used by **StreamReader** (p. 102) to notify RTI Routing **Service** (p. 83) when new data is available.*
- class **rti::routing::adapter::DiscoveryStreamReader**

  *Definition of a special **StreamReader** (p. 102) to read information about data streams.*
- class **rti::routing::adapter::Session**
- class **rti::routing::adapter::StreamReader**

  Provides a way to read samples of a specific type from a data domain. In the XML configuration file, StreamReaders are associated with the tag <input> within <route> and <auto_route>.
- class **rti::routing::adapter::TStreamReader< Data, Info >**

  *A wrapper implementation of a **StreamReader** (p. 102) that provides a strongly-typed interface through template parameters for data and info representation.*
- class **rti::routing::adapter::NoOpStreamReader< Data, Info >**

  *An empty implementation of the **TStreamReader** (p. 115) interface.*
- class **rti::routing::adapter::SelectorState**

  *Defines a set of attributes that can be used to read a subset of data from **StreamReader** (p. 102).*
- class **rti::routing::adapter::StreamWriter**

  Provides a way to write samples of a specific type in a data domain.
- class **rti::routing::adapter::TStreamWriter< Data, Info >**

  *A wrapper implementation of a **StreamWriter** (p. 109) that provides a strongly-typed interface through template parameters for data and info representation.*

Macros

- #define **RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DECL**(ADAPTER_CLASS)

  Utility macro that declares a native extern function that can be used to load an AdapterPlugin through a shared library.
- #define **RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DEF**(ADAPTER_CLASS)

  Utility macro that implements the AdapterPlugin entry point declared with RTI_ADAPTER_PLUGIN_CREATE_↩ FUNCTION_DECL.

Typedefs

- typedef **NoOpStreamReader**< dds::core::xtypes::DynamicData, dds::sub::SampleInfo > **rti::routing↩ ::adapter::DynamicDataStreamReader**

  *Convenient definition of typed **StreamReader*** (p. *102) that requires dds::core::xtypes::DynamicData for data samples and dds::sub::SampleInfo for info samples.*
- typedef **TStreamWriter**< dds::core::xtypes::DynamicData, dds::sub::SampleInfo > **rti::routing::adapter::↩ DynamicDataStreamWriter**

  *Convenient definition of typed **StreamWriter*** (p. *109) that requires dds::core::xtypes::DynamicData for data samples and dds::sub::SampleInfo for info samples.*

7.1.1 Detailed Description

This module describes the C++ Adapter API.

Adapters are pluggable components that allow *RTI Routing Service* to consume and produce data for different data domains (e.g., Connext DDS, MQTT, Socket, etc.).

By default, *Routing Service* is distributed with a builtin DDS adapter. Any other adapter plugins must be provided as external components registered through the XML configuration or through the Library API.

The following figure shows an overview of the *Routing Service* adapter.

Input adapters are used to collect data samples from different data domains, such as DDS or MQTT. The input samples are processed by the Routing Service engine and are passed along to output adapters through the Processor, applying any Transformation beforehand if present.

For additional details about Adapter configuration see the `RTI Routing Service User's Manual`.

7.1.2 Development Requirements

	Linux/macOS Systems	Windows Systems
Shared Libraries	libnddscpp2.so	nddscpp2.dll
	librtirsinfrastructure.so	rtirsinfrastructure.dll
	libnddsc.so	nddsc.dll
	libnddscore.so	nddscore.dll
Headers	**rti/routing/adapter/AdapterPlugin.hpp** (p. 145)	

7.1.3 Architecture

The Adapter architecture is shown in the class diagram below.

The sequence diagram in this figure shows when the different adapter entities are created.

- An Adapter object is created as soon as RTI Routing Service starts and finds out about the plug-ins that will be used by underlying DomainRoute connections.
- A Connection object is created when the DomainRoute (<domain_route>) that contain it is enabled.
- A Session object is created when the associated service Session (<session>) is enabled.
- An inputs StreamReader is created if the Route is enabled and the creation mode condition associated with the <input> tag becomes true.
- An outputs StreamWriter is created if the route is enabled and the creation mode condition associated with the <output> tag becomes true.

7.1.3.1 Stream Discovery

A Route cannot forward data until the type representations (e.g., TypeCode) associated with the input and output streams are available.

If a Route refers to types that are not defined in the configuration file, RTI Routing Service has to discover their type representation (e.g., TypeCode) before creating the StreamReader and StreamWriter. The adapter discovery API is used to provide stream and type information in a data domain to Routing Service∗.

The discovery API consists of the following methods:

- **rti::routing::adapter::Connection::input_stream_discovery_reader** (p. 36)
- **rti::routing::adapter::Connection::output_stream_discovery_reader** (p. 36)

These methods provide access to StreamReader used to discover streams in the data domain associated with a connection.

The input stream discovery StreamReader provides information about input streams. An input stream is a stream from which an input's StreamReader reads data. Notification of disposed scenarios, where an input stream disappears, are also made using the input stream discovery StreamReader.

In the builtin DDS adapter, the input stream discovery StreamReader is associated with the publication builtin Data↩ Reader of the DomainParticipant.

The output stream discovery StreamReader provides information about output streams. An output stream is a stream to which an output's StreamWriter can write data. Notification of disposed scenarios, where an output stream disappears, are also made using the output stream discovery StreamReader.

In the builtin DDS adapter, the output stream discovery StreamReader is associated with the subscription builtin Data↩ Reader of the DomainParticipant.

The samples provided by the stream discovery StreamReaders have the type **rti::routing::StreamInfo** (p. 99).

7.1.4 Macro Definition Documentation

7.1.4.1 RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DECL

```
#define RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DECL(
               ADAPTER_CLASS )
```

Value:

```
extern "C" RTI_USER_DLL_EXPORT struct RTI_RoutingServiceAdapterPluginExt * \
    ADAPTER_CLASS ## _create_adapter_plugin(\
        const struct RTI_RoutingServiceProperties *, \
        RTI_RoutingServiceEnvironment *); \
```

Utility macro that declares a native extern function that can be used to load an AdapterPlugin through a shared library.

The prototype of the function is given by RTI_RoutingServiceAdapterPlugin_CreateFcn.

To register an AdapterPlugin plugin with RTI Routing Service, you must use the tag <adapter_plugin> within <plugin↩ _library>. For example:

```
<dds>
    ...
     <plugin_library name="MyPluginLib">
         <adapter_plugin name="MyAdapterPlugin">
             <dll>mycadapter</dll>
             <create_function>
                 MyAdapterPlugin_create
             </create_function>
         </adapter_plugin>
        ...
     </plugin_library>
     ...
     <routing_service>
     ...
     </routing_service>
     ...
</dds>
```

Once an AdapterPlugin is registered, a Connection can refer to it as follows:

For example:

```
<dds>
    <routing_service name="example">
        <domain_route name="myadapter_to_dds">
            <connection name="MyConnection" plugin_name="MyAdapter">
                 <property>
                     <value>
                         <element>
                             <name>my_property_name</name>
                             <value>my_property_value</value>
                         </element>
                     </value>
                 </property>
            </connection>
         ...
        </domain_route>
     </routing_service>
</dds>
```

For additional information on configuring adapters, see the `RTI Routing Service User's Manual`.

Parameters

ADAPTER_CLASS	Class name of an AdapterPlugin implementation

7.1.4.2 RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DEF

```
#define RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DEF(
               ADAPTER_CLASS )
```

Value:

```
struct RTI_RoutingServiceAdapterPluginExt * ADAPTER_CLASS ## _create_adapter_plugin( \
        const struct RTI_RoutingServiceProperties * native_properties, \
        RTI_RoutingServiceEnvironment *environment) \
{ \
    PropertySet properties; \
    rti::routing::PropertyAdapter::add_properties_from_native(\
            properties,\
            native_properties); \
    try { \
        return rti::routing::adapter::detail::AdapterForwarder::create_plugin(new ADAPTER_CLASS(properties)); \
    } catch (const std::exception& ex) {\
        RTI_RoutingServiceEnvironment_set_error(\
                environment,\
                "%s",\
                ex.what());\
    } catch (...) {}\
    \
    return NULL; \
}
```

Utility macro that implements the AdapterPlugin entry point declared with RTI_ADAPTER_PLUGIN_CREATE_↩ FUNCTION_DECL.

7.1.5 Typedef Documentation

7.1.5.1 DynamicDataStreamReader

```
typedef NoOpStreamReader<dds::core::xtypes::DynamicData, dds::sub::SampleInfo> rti::routing↩
::adapter::DynamicDataStreamReader
```

Convenient definition of typed **StreamReader** (p. 102) that requires dds::core::xtypes::DynamicData for data samples and dds::sub::SampleInfo for info samples.

7.1.5.2 DynamicDataStreamWriter

```
typedef TStreamWriter<dds::core::xtypes::DynamicData, dds::sub::SampleInfo> rti::routing↩
::adapter::DynamicDataStreamWriter
```

Convenient definition of typed **StreamWriter** (p. 109) that requires dds::core::xtypes::DynamicData for data samples and dds::sub::SampleInfo for info samples.

7.2 RTI Routing Service

RTI Routing Service.

Modules

- **RTI Routing Service Adapter API**

  This module describes the C++ Adapter API.
- **RTI Routing Service Infrastructure**

  This module contains common definitions used across other functional modules.
- **RTI Routing Service Processor API**

  This module describes the Processor API.
- **RTI Routing Library API**

  *RTI Routing **Service** (p. 83) can be deployed as a native library linked into your application in select architectures.*
- **RTI Routing Service Transformation API**

  This module describes the Transformation API.

7.2.1 Detailed Description

RTI Routing Service.

7.3 RTI Routing Service Infrastructure

This module contains common definitions used across other functional modules.

Modules

- **Standard Type Representation Kinds**

  Standard type Representation kinds.
- **Standard Data Representation Kinds**

  Standard data representation kinds.

Data Structures

- class **rti::routing::StreamInfo**

  *Definition of the stream information that RTI Routing **Service** (p. 83) needs to manage user data streams.*
- class **rti::routing::TypeRepresentationKind_def**

  The definition of the rti::routing::TypeRepresentationKind.
- class **TypeRepresentationKind**

  valid type representations that RTI Routing Service accepts.
- class **rti::routing::TypeInfo**

  *Definition of the type information associated with a RTI Routing **Service** (p. 83) stream.*
- class **rti::routing::UpdatableEntity**

  Defines a common interface for all the pluggable entities that can be updated at runtime.

Typedefs

- typedef std::map< std::string, std::string > **rti::routing::PropertySet**

  The definition of a pluggable entitys configuration properties.

7.3.1 Detailed Description

This module contains common definitions used across other functional modules.

7.3.2 Typedef Documentation

7.3.2.1 PropertySet

`typedef std::map<std::string,std::string>` **`rti::routing::PropertySet`**

The definition of a pluggable entitys configuration properties.

Configuration properties for an entity are obtained as a result of parsing the <property> corresponding to each pluggable entity. The result is a map keyed by property name, represented as std::string, whose values are the property values, also represented as std::string. Hence, a property name shall be unique for each set of entity properties.

7.4 Standard Type Representation Kinds

Standard type Representation kinds.

Standard type Representation kinds.

7.5 Standard Data Representation Kinds

Standard data representation kinds.

Standard data representation kinds.

7.6 RTI Routing Service Processor API

This module describes the Processor API.

Data Structures

- class **rti::routing::processor::Input**

  *Generic Representation of a **Route** (p. 67)'s input.*
- class **rti::routing::processor::TypedInput< Data, Info >**

  *Representation of an **Input** (p. 39) whose data representation is `DataRep`, whose info representation is `InfoRep`.*
- class **rti::routing::processor::Selector< Data, Info >**

  An element that allows reading data that meet a set of specified attributes.
- class **rti::routing::processor::LoanedSamples< T, U >**

  *Provides temporary access to a collection of samples (data and info) from a **TypedInput** (p. 126).*
- class **rti::routing::processor::Output**

  *Generic Representation of a **Route** (p. 67)'s output.*
- class **rti::routing::processor::TypedOutput< Data, Info >**

  *Representation of an **Output** (p. 55) whose data representation is `DataRep`, whose info representation is `InfoRep`.*
- class **rti::routing::processor::Processor**

  ***Processor** (p. 56) interface definition. Provides a way to process **Route** (p. 67) events and control the data forwarding process.*
- class **rti::routing::processor::NoOpProcessor**

  *An empty implementation of the pure virtual interface **rti::routing::processor::Processor** (p. 56).*
- class **rti::routing::processor::ProcessorPlugin**

  The top-level plug-in class.
- class **rti::routing::processor::Query**

  *Encapsulates a content query to select data from a **rti::routing::adapter::StreamReader** (p. 102).*
- class **rti::routing::processor::Route**

  *Representation of the **Route** (p. 67) object that owns a **Processor** (p. 56).*
- class **rti::routing::processor::SampleIterator< T, U >**

  A random-access iterator of LoanedSample.

Macros

- #define **RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DECL**(PROCESSOR_CLASS)

  Utility macro that declares a native extern function that can be used to load a ProcessorPlugin through a shared library.
- #define **RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DEF**(PROCESSOR_CLASS)

  Utility macro that implements the ProcessorPlugin entry point declared with RTI_PROCESSOR_PLUGIN_CREATE_↩ FUNCTION_DECL.

7.6.1 Detailed Description

This module describes the Processor API.

RTI Routing Service **rti::routing::processor::Processor** (p. 56) are **event-oriented pluggable components** that allow you to control the forwarding process that occur within a Route.

7.6.1.1 Development Requirements

	Linux/macOS Systems	Windows Systems
Shared Libraries	librtiroutingservice.so	librtiroutingservice.lib
	librtirsinfrastructure.so	rtirsinfrastructure.lib
	libnddscpp2.so	nddscpp2.lib
	libnddsc.so	nddsc.lib
	libnddscore.so	nddscore.lib
Headers	**rti/routing/processor/ProcessorPlugin.hpp** (p. 212)	

7.6.1.2 Architecture

A **rti::routing::processor::Processor** (p. 56) represents a multiple-input, multiple-output component attached to a **rti::routing::processor::Route** (p. 67), which dispatches events to the **rti::routing::processor::Processor** (p. 56). The figure below shows the role of a **rti::routing::processor::Processor** (p. 56) within RTI Routing Service.

A Processor can access the *N* inputs and *M* outputs of the owner Route. Upon event notification, the Processor can read data from any input, perform any manipulation on the data, and write it on any of the outputs.

Note in the figure above that any data a Processor writes on an output is first transformed with an **rti::routing::transf::Transformation** (p. 111) before passing it to the underlying **rti::routing::adapter::StreamWriter** (p. 109).

An example snippet is shown below. The example code is reading data from two inputs, merging it together in an intermediate data buffer and writing into a single output.

```
on_data_available(Route& route)
{
   auto output = route.output<DynamicData>(0);
   auto output_sample = output.create_data();
   for(const auto& status : route.input<DynamicData>(0).take()) {
        auto periodic  = route.input<DynamicData>(1)
                .select().instance(status.info().instance_handle()).read();
        output_sample.value<int>(
                "id",
                status.data().get<int>("id"));
        output_sample.value<int>(
                "config",
                 status.data().get<string>("config"));
        output_sample.value<int>(
                 "latency",
                 periodic[0].data().get<int>("latency"));
        output.write(output_sample);
   }
}
```

The Processor API component model is shown in figure below.

- **rti::routing::processor::Processor** (p. 56)
- **rti::routing::processor::Route** (p. 67)
- **rti::routing::processor::Input** (p. 39)
- **rti::routing::processor::Output** (p. 55)
- **rti::routing::processor::Selector** (p. 77)

7.6.1.3 Event dispatching

A **rti::routing::processor::Route** (p. 67) will notify its contained **rti::routing::processor::Processor** (p. 56) about the events that affect the Route object. Each event kind maps to a virtual operation in the **rti::routing::processor::↩ Processor** (p. 56) interface:

- **Input Enabled**: Event that represents the transition from disables to enabled on any of the contained **rti↩ ::routing::processor::Input** (p. 39). Enabling an Input results in the creation of the underlying **rti::routing↩ ::adapter::StreamReader** (p. 102). This event is notified through **rti::routing::processor::Processor::on_↩ input_enabled** (p. 58).
- **Input Disabled**: Event that represents the transition from enabled to disabled on any of the contained **rti↩ ::routing::processor::Input** (p. 39). Disabling an Input results in the deletion of the underlying **rti::routing↩ ::adapter::StreamReader** (p. 102). This event is notified through **rti::routing::processor::Processor::on_↩ input_disabled** (p. 58).
- **Output Enabled**: Event that represents the transition from disabled to enabled on any of the contained **rti::routing::processor::Output** (p. 55). Enabling an Output results in the creation of the underlying **rti↩ ::routing::transf::Transformation** (p. 111) and **rti::routing::adapter::StreamWriter** (p. 109). This event is notified through **rti::routing::processor::Processor::on_output_enabled** (p. 59).
- **Output Disabled**: Event that represents the transition from enabled to disabled on any of the contained **rti::routing::processor::Output** (p. 55). Disabling an Output results in the deletion of the underlying **rti↩ ::routing::transf::Transformation** (p. 111) and **rti::routing::adapter::StreamWriter** (p. 109). This event is notified through **rti::routing::processor::Processor::on_output_disabled** (p. 59).
- **Route Started**: This event indicates that the owner **rti::routing::processor::Route** (p. 67) has enabled all inputs and outputs. This event is notified through **rti::routing::processor::Processor::on_start** (p. 60).
- **Route Stopped**: This event indicates that the owner **rti::routing::processor::Route** (p. 67) is about to disable at least one of its inputs or outputs. This event is notified through **rti::routing::processor::Processor::on_stop** (p. 60).
- **Route Running**: This event indicates that the owner **rti::routing::processor::Route** (p. 67) is ready to provide user event notifications. This event is notified through **rti::routing::processor::Processor::on_run** (p. 61).
- **Route Paused**: This event indicates that the owner **rti::routing::processor::Route** (p. 67) will temporarily stop providing user event notifications. This event is notified through **rti::routing::processor::Processor::on_pause** (p. 61).
- **Data Available**: This user event indicates at least one of the inputs within the owner **rti::routing::processor::↩ Route** (p. 67) has received new data. This event is notified through **rti::routing::processor::Processor::on_↩ data_available** (p. 63).
- **Periodic Action**: This user event represents a periodic notification, at a rate configured in the parent Session of the owner **rti::routing::processor::Route** (p. 67). This event is notified through **rti::routing::processor::↩ Processor::on_periodic_action** (p. 62).

7.6.1.4 Threading

A **rti::routing::processor::Route** (p. 67) dispatches all the events to its contained processors **sequentially**, from within one of the parent Session's threads. For any given Route, and hence a Processor, it is guaranteed that only one event is dispatched at at time. That is, the operations on the Processor are never called concurrently for a given Route.

A parent Session with *L* threads can only process one Route at a time but it can dispatch different Routes concurrently. Namely, it can dispatch up to *L* Routes concurrently. You should keep this behavior in mind for implementations that share state between Processor objects.

7.6.1.5 Constraints

There are several constraints that affect the behavior and relationship of the Processor API components:

- A Route can have one and only one Processor. Alternatively, *semantically* a Processor can belong to one and only one Route. That is, each Route will call **rti::routing::processor::ProcessorPlugin::create_processor** (p. 64) and **rti::routing::processor::ProcessorPlugin::delete_processor** (p. 65) on the corresponding plug-in instance, if the implementation is returning the same physical object.
- An Input/Output can belong to one and only one Route. Thus, they cannot be shared across different Routes. Alternatively, operations on an Input, Output and Route can be performed only from within the associated Processor. It is forbidden to pass Input, Output and Route objects between Processors

7.6.2 Macro Definition Documentation

7.6.2.1 RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DECL

```
#define RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DECL(
               PROCESSOR_CLASS )
```

Value:

```
extern "C" RTI_USER_DLL_EXPORT struct RTI_RoutingServiceProcessorPlugin* \
    PROCESSOR_CLASS ## _create_processor_plugin(\
        const struct RTI_RoutingServiceProperties *, \
        RTI_RoutingServiceEnvironment *); \
```

Utility macro that declares a native extern function that can be used to load a ProcessorPlugin through a shared library.

The prototype of the function is given by a native definition of the create operation.

To register a ProcessorPlugin with RTI Routing Service, you must use the tag <processor_plugin> within <plugin_↩ library>. For example:

```
<dds>
    ...
     <plugin_library name="MyPluginLib">
         <processor_plugin name="MyProcessorPlugin">
             <dll>myprocessor</dll>
             <create_function>
                MyProcessorPlugin_create
            </create_function>
        </processor_plugin>
        ...
     </plugin_library>
     ...
     <routing_service>
     ...
     </routing_service>
     ...
</dds>
```

Once a ProcessorPlugin is registered, a Route can use it to create a Processor.

For example:

```
<topic_route name="SquareAndCircleToTriangle">
   ...
   <processor plugin_name="MyPluginLib::MyProcessorPlugin">
```

```
        <property>
            <value>
                <element>
                    <name>my_property_name</name>
                    <value>my_property_value</value>
                </element>
            </value>
        </property>
    </processor>
    <input participant="DomainIn">
        <topic_name>Square</topic_name>
        <registered_type_name>ShapeType</registered_type_name>
    </input>
    <input participant="DomainIn">
        <topic_name>Circle</topic_name>
        <registered_type_name>ShapeType</registered_type_name>
    </input>
    <output participant="DomainOut">
        <topic_name>Triangle</topic_name>
        <registered_type_name>ShapeType</registered_type_name>
    </output>
</topic_route>
```

For additional information on configuring Processors, see the `RTI Routing Service User's Manual`.

Parameters

PROCESSOR_CLASS	Class name of a ProcessorPlugin implementation

7.6.2.2 RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DEF

```
#define RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DEF(
                PROCESSOR_CLASS )
```

Value:

```
struct RTI_RoutingServiceProcessorPlugin * PROCESSOR_CLASS ## _create_processor_plugin( \
        const struct RTI_RoutingServiceProperties * native_properties, \
        RTI_RoutingServiceEnvironment *environment) \
{ \
    PropertySet properties; \
    rti::routing::PropertyAdapter::add_properties_from_native(\
            properties,\
            native_properties); \
    try { \
        return rti::routing::processor::detail::ProcessorPluginForwarder::create_plugin(\
                new PROCESSOR_CLASS(properties)); \
    } catch (const std::exception& ex) {\
        RTI_RoutingServiceEnvironment_set_error(\
                environment,\
                "%s",\
                ex.what());\
    } catch (...) {}\
    \
    return NULL; \
}
```

Utility macro that implements the ProcessorPlugin entry point declared with RTI_PROCESSOR_PLUGIN_CREATE_↩ FUNCTION_DECL.

7.7 RTI Routing Library API

RTI Routing **Service** (p. 83) can be deployed as a native library linked into your application in select architectures.

Data Structures

- class **rti::routing::Logger**

  *The singleton type used to configure RTI Routing **Service** (p. 83) verbosity.*
- class **rti::routing::Service**

  *The RTI Routing **Service** (p. 83).*
- class **rti::routing::ServiceProperty**

  *Configuration for a RTI Routing **Service** (p. 83) object.*

7.7.1 Detailed Description

RTI Routing **Service** (p. 83) can be deployed as a native library linked into your application in select architectures.

This API allows you to create, configure and start RTI Routing **Service** (p. 83) instances from your application.

The following code shows the typical use of the API:

```
int main ()
{
    rti::routing::Service service(
            rti::routing::ServiceProperty()
            .cfg_file("MyRouter.xml")
            .cfg_name("MyRouter"));
    ...
    service.start();
    while(keep_running) {
        sleep();
        ...
    }
    return 0;
}
```

Instead of a file, you can use XML strings to configure RTI Routing **Service** (p. 83). See **ServiceProperty** (p. 88) for more information.

To build your application you need to link with the RTI Routing **Service** (p. 83) native library in <**RTI Connext home**>/**bin**/<**architecture**>/ .

	Linux/macOS Systems	**Windows Systems**
Shared Libraries	libnddscpp2.so	nddscpp2.dll
	librtirsinfrastructure.so	rtirsinfrastructure.dll
	librtiserviceadmincpp.so	rtiserviceadmincpp.dll
	librtidlc.so	rtidlc.dll
	librtiapputilsc.so	rtiapputilsc.dll
	libnddsmetp.so	nddsmetp.dll
	librticonnextmsgc.so	rticonnextmsgc.dll
	libnddsc.so	nddsc.dll
	librtixml2.so	rtixml2.dll
	libnddscore.so	nddscore.dll
Headers	**rti/routing/RoutingService.hpp** (p. 223)	

7.7.1.0.1 Development Requirements

7.8 RTI Routing Service Transformation API

This module describes the Transformation API.

Data Structures

- class **rti::routing::transf::Transformation**

  Provides a way to transform input samples into output samples of a different format and/or content.
- class **rti::routing::transf::TypedTransformation< Data, Info >**

  *A wrapper implementation of a **Transformation** (p. 111) that provides a strongly-typed interface through template parameters for data and info representation.*
- class **rti::routing::transf::TransformationPlugin**

  The top-level plug-in class.

Macros

- #define **RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DECL**(TRANSFORMATION_PLUGIN_↩ CLASS)

  Utility macro that declares a native extern function that can be used to load a TransformationPlugin through a shared library.
- #define **RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DEF**(TRANSFORMATION_PLUGIN_↩ CLASS)

  Utility macro that implements the TransformationPlugin entry point declared with RTI_TRANSFORMATION_PLUGIN_↩ CREATE_FUNCTION_DECL.

Typedefs

- typedef **TypedTransformation**< dds::core::xtypes::DynamicData, dds::sub::SampleInfo > **rti::routing↩ ::transf::DynamicDataTransformation**

  *Convenient definition of **TypedTransformation** (p. 133) that requires dds::core::xtypes::DynamicData for data samples and dds::sub::SampleInfo for info samples.*

7.8.1 Detailed Description

This module describes the Transformation API.

An RTI Routing Service Output transforms the incoming data using a *transformation*, which is an object created by a transformation plugin.

7.8.2 Macro Definition Documentation

7.8.2.1 RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DECL

```
#define RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DECL(
                TRANSFORMATION_PLUGIN_CLASS )
```

Value:

```
extern "C" RTI_USER_DLL_EXPORT struct RTI_RoutingServiceTransformationPlugin * \
    TRANSFORMATION_PLUGIN_CLASS ## _create_transformation_plugin(\
        const struct RTI_RoutingServiceProperties *, \
        RTI_RoutingServiceEnvironment *); \
```

Utility macro that declares a native extern function that can be used to load a TransformationPlugin through a shared library.

The prototype of the function is given by RTI_RoutingServiceTransformationPlugin_CreateFcn.

To register a transformation plugin with RTI Routing Service, you must use the tag <transformation_plugin> within <plugin_library>. For example:

```
<dds>
    ...
     <plugin_library name="MyPluginLib">
         <transformation_plugin name="MyTransfPlugin">
             <dll>mytransformation</dll>
             <create_function>
                MyTransfPlugin_create
            </create_function>
        </transformation_plugin>
        ...
     </plugin_library>
     ...
     <routing_service>
     ...
     </routing_service>
     ...
</dds>
```

Once a transformation plugin is registered, an Output can use it to create a data transformation.

For example:

```
<topic_route name="SquareSwitchCoord">
   <input participant="1">
       <topic_name>Square</topic_name>
       <registered_type_name>ShapeType</registered_type_name>
   </input>
   <output>
       <topic_name>Square</topic_name>
       <registered_type_name>ShapeType</registered_type_name>
       <transformation plugin_name="MyPluginLib::MyTransPlugin">
           <property>
               <value>
                   <element>
                       <name>X</name>
                       <value>Y</value>
                   </element>
                   <element>
                       <name>Y</name>
                       <value>X</value>
                   </element>
               </value>
           </property>
       </transformation>
   </output>
</topic_route>
```

For additional information on configuring transformations, see the `RTI Routing Service User's Manual`.

Parameters

TRANSFORMATION_PLUGIN_CLASS	Class name of a TransformationPlugin implementation

7.8.2.2 RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DEF

```
#define RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DEF(
                TRANSFORMATION_PLUGIN_CLASS )
```

Value:

```
struct RTI_RoutingServiceTransformationPlugin * \
    TRANSFORMATION_PLUGIN_CLASS ## _create_transformation_plugin( \
        const struct RTI_RoutingServiceProperties * native_properties, \
        RTI_RoutingServiceEnvironment *environment) \
{ \
    rti::routing::PropertySet properties; \
    rti::routing::PropertyAdapter::add_properties_from_native(\
            properties,\
            native_properties); \
    try { \
        return rti::routing::transf::detail::TransformationPluginForwarder::create_plugin(\
            new TRANSFORMATION_PLUGIN_CLASS(properties)); \
    } catch (const std::exception& ex) {\
        RTI_RoutingServiceEnvironment_set_error(\
                environment,\
                "%s",\
                ex.what());\
    } catch (...) {}\
    \
    return NULL; \
}
```

Utility macro that implements the TransformationPlugin entry point declared with RTI_TRANSFORMATION_PLUGIN↩ _CREATE_FUNCTION_DECL.

7.8.3 Typedef Documentation

7.8.3.1 DynamicDataTransformation

typedef **TypedTransformation**<dds::core::xtypes::DynamicData, dds::sub::SampleInfo> **rti::routing↩ ::transf::DynamicDataTransformation**

Convenient definition of **TypedTransformation** (p. 133) that requires dds::core::xtypes::DynamicData for data samples and dds::sub::SampleInfo for info samples.

Chapter 8

Data Structure Documentation

8.1 rti::routing::adapter::AdapterPlugin Class Reference

The top-level plug-in class.

`#include <AdapterPlugin.hpp>`

Public Member Functions

- virtual **Connection** * **create_connection** (StreamReaderListener *input_stream_discovery_listener, StreamReaderListener *output_stream_discovery_listener, const **PropertySet** &properties)=0

  *Creates a **Connection** (p. 31).*
- virtual void **delete_connection** (**Connection** *connection)=0

  *Deletes a **Connection** (p. 31).*
- virtual rti::config::LibraryVersion **get_version** () const
- virtual ~**AdapterPlugin** ()

  Virtual destructor.

8.1.1 Detailed Description

The top-level plug-in class.

Represents a factory of **Connection** (p. 31).

8.1.2 Constructor & Destructor Documentation

8.1.2.1 ~AdapterPlugin()

```
virtual rti::routing::adapter::AdapterPlugin::~AdapterPlugin ( )  [inline], [virtual]
```

Virtual destructor.

8.1.3 Member Function Documentation

8.1.3.1 create_connection()

```
virtual  Connection * rti::routing::adapter::AdapterPlugin::create_connection (
            StreamReaderListener * input_stream_discovery_listener,
            StreamReaderListener * output_stream_discovery_listener,
            const  PropertySet & properties )  [pure virtual]
```

Creates a **Connection** (p. 31).

Connection (p. 31) objects are created when the domain routes that contain them are enabled.

Parameters

input_stream_discovery_listener	<<*in*>> The listener of the built-in **DiscoveryStreamReader** (p. 37) that notifies the discovery of new input streams.
output_stream_discovery_listener	<<*in*>> The listener of the built-in **DiscoveryStreamReader** (p. 37) that notifies the discovery of new output streams.
properties	<<*in*>> Configuration properties for the **Connection** (p. 31). These properties corresponds to the properties specified within the tag <connection>.

Returns

New **Connection** (p. 31) if successful. Cannot return nullptr.

Exceptions

std::exception	

Multi-threading safety Safe

8.1.3.2 delete_connection()

```
virtual void rti::routing::adapter::AdapterPlugin::delete_connection (
              Connection * connection )  [pure virtual]
```

Deletes a **Connection** (p. 31).

Connection (p. 31) objects are deleted when the domain routes that contain them are disabled or RTI Routing **Service** (p. 83) is stopped.

Parameters

connection	<<*in*>> **Connection** (p. 31) to be deleted

Exceptions

std::exception	

Multi-threading safety Safe

8.1.3.3 get_version()

```
virtual rti::config::LibraryVersion rti::routing::adapter::AdapterPlugin::get_version ( ) const
[inline], [virtual]
```

Returns

The version of this **AdapterPlugin** (p. 29).

Version is used for logging purposes and allows you to track which version of the **AdapterPlugin** (p. 29) RTI Routing **Service** (p. 83) is using.

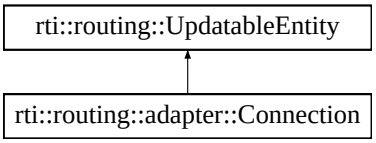
Default implementation of this operation returns the version of the required Connext libraries.

8.2 rti::routing::adapter::Connection Class Reference

A **Connection** (p. 31) provides access to a data domain (such as a DDS domain or a JMS network provider).

```
#include <Connection.hpp>
```

Inheritance diagram for rti::routing::adapter::Connection:



Public Member Functions

- virtual **Session** * **create_session** (const **PropertySet** &properties)
  *Creates a **Session** (p. 99).*
- virtual void **delete_session** (**Session** *session)
  *Deletes a **Session** (p. 99).*
- virtual **StreamWriter** * **create_stream_writer** (**Session** *session, const **StreamInfo** &stream_info, const **PropertySet** &properties)
  *Creates a **StreamWriter** (p. 109).*
- virtual void **delete_stream_writer** (**StreamWriter** *stream_writer)
  *Deletes a **StreamWriter** (p. 109).*
- virtual **StreamReader** * **create_stream_reader** (**Session** *session, const **StreamInfo** &stream_info, const **PropertySet** &properties, StreamReaderListener *listener)
  *Creates a **StreamReader** (p. 102).*
- virtual void **delete_stream_reader** (**StreamReader** *stream_reader)
  *Deletes a **StreamReader** (p. 102).*
- virtual **DiscoveryStreamReader** * **input_stream_discovery_reader** ()
  *Gets the input stream discovery **StreamReader** (p. 102).*
- virtual **DiscoveryStreamReader** * **output_stream_discovery_reader** ()
  *Gets the output stream discovery **StreamReader** (p. 102).*
- virtual ~**Connection** ()
  Virtual destructor.

8.2.1 Detailed Description

A **Connection** (p. 31) provides access to a data domain (such as a DDS domain or a JMS network provider).

In the XML configuration file, Connections are associated with the tag <connection> within a domain route.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 ~Connection()

```
virtual rti::routing::adapter::Connection::~Connection ( )  [inline], [virtual]
```

Virtual destructor.

8.2.3 Member Function Documentation

8.2.3.1 create_session()

```
virtual  Session * rti::routing::adapter::Connection::create_session (
            const  PropertySet & properties )  [inline], [virtual]
```

Creates a **Session** (p. 99).

Session (p. 99) objects are created when the associated RTI Routing **Service** (p. 83) Sessions are enabled. This operation is called once for each service **Session** (p. 99) and for each **Connection** (p. 31) within the DomainRoute.

Parameters

properties	<<*in*>> Configuration properties for the **Session** (p. 99).

Returns

New **Session** (p. 99) if successful. Can return nullptr if sessions are not required by this **AdapterPlugin** (p. 29).

Exceptions

std::exception	

Multi-threading safety Safe

8.2.3.2 delete_session()

```
virtual void rti::routing::adapter::Connection::delete_session (
              Session * session )  [inline], [virtual]
```

Deletes a **Session** (p. 99).

Session (p. 99) objects are deleted when the RTI Routing **Service** (p. 83) Sessions that contain them are disabled.

Parameters

session	<<*in*>> **Session** (p. 99) to be deleted.

Exceptions

std::exception	

Multi-threading safety Safe

8.2.3.3 create_stream_writer()

```
virtual  StreamWriter * rti::routing::adapter::Connection::create_stream_writer (
              Session * session,
```

```
        const  StreamInfo & stream_info,
        const  PropertySet & properties )  [inline], [virtual]
```

Creates a **StreamWriter** (p. 109).

This method is called for each Route's output when the associated 'creation mode' condition is met. The operation is called on the output **Connection** (p. 31) as a specified through the 'connection' attribute in the <output> tag.

Parameters

session	<<*in*>> **Session** (p. 99) associated with the **StreamWriter** (p. 109). This parameter is nullptr if Sessions are not used by the adapter.
stream_info	<<*in*>> Information related to the stream where the **StreamWriter** (p. 109) sends data.
properties	<<*in*>> Configuration properties for the **StreamWriter** (p. 109).

Returns

New **StreamWriter** (p. 109) if successful. Cannot return nullptr

Exceptions

std::exception	

Multi-threading safety Partially Safe. This operation is never called concurrently for the same **Connection** (p. 31), even if multiple outputs are configured in different Sessions. It can only be called concurrently for different Connections.

8.2.3.4 delete_stream_writer()

```
virtual void rti::routing::adapter::Connection::delete_stream_writer (
            StreamWriter * stream_writer )  [inline], [virtual]
```

Deletes a **StreamWriter** (p. 109).

A **StreamWriter** (p. 109) object is deleted when:

- The route that contains it is disabled
- The 'creation mode' condition associated with the route's output becomes false
- RTI Routing **Service** (p. 83) is shutdown.

Parameters

stream_writer	<<*in*>> **StreamWriter** (p. 109) to be deleted.

Exceptions

std::exception	

8.2.3.5 create_stream_reader()

```
virtual  StreamReader * rti::routing::adapter::Connection::create_stream_reader (
              Session * session,
             const  StreamInfo & stream_info,
             const  PropertySet & properties,
             StreamReaderListener * listener )  [inline], [virtual]
```

Creates a **StreamReader** (p. 102).

This method is called for each Route's input when the associated 'creation mode' condition is met. The operation is called on the input **Connection** (p. 31) as a specified through the 'connection' attribute in the <output> tag.

Parameters

session	<<*in*>> **Session** (p. 99) associated with the **StreamReader** (p. 102). This parameter is nullptr if Sessions are not used by the adapter.
stream_info	<<*in*>> Information related to the stream where the **StreamReader** (p. 102) read data.
properties	<<*in*>> Configuration properties for the **StreamReader** (p. 102).
listener	<<*in*>> The listener for the **StreamReader** (p. 102) implementation used to notify RTI Routing **Service** (p. 83) when new data is available.

Returns

New **StreamReader** (p. 102) if successful. Cannot return nullptr.

Exceptions

std::exception	

Multi-threading safety Partially Safe. This operation is never called concurrently for the same **Connection** (p. 31), even if multiple outputs are configured in different Sessions. It can only be called concurrently for different Connections.

8.2.3.6 delete_stream_reader()

```
virtual void rti::routing::adapter::Connection::delete_stream_reader (
              StreamReader * stream_reader )  [inline], [virtual]
```

Deletes a **StreamReader** (p. 102).

A **StreamReader** (p. 102) object is deleted when:

- The route that contains it is disabled
- The 'creation mode' condition associated with the route's input becomes false
- RTI Routing **Service** (p. 83) is shutdown.

Parameters

stream_reader	<<*in*>> **StreamReader** (p. 102) to be deleted.

Exceptions

std::exception	

8.2.3.7 input_stream_discovery_reader()

```
virtual  DiscoveryStreamReader * rti::routing::adapter::Connection::input_stream_discovery_reader
( )  [inline], [virtual]
```

Gets the input stream discovery **StreamReader** (p. 102).

Returns a **DiscoveryStreamReader** (p. 37) that is used by RTI Routing **Service** (p. 83) to discover input streams.

An input stream is a stream from which a **StreamReader** (p. 102) can read data. Disposed scenarios, where an input stream disappears, are also notified using this **DiscoveryStreamReader** (p. 37).

The **StreamReaderListener** (p. 108) associated with this **DiscoveryStreamReader** (p. 37) is provided as a parameter to the method **AdapterPlugin::create_connection** (p. 30)

Implementations may return nullptr if they do not support stream discovery.

Exceptions

std::exception	

8.2.3.8 output_stream_discovery_reader()

```
virtual DiscoveryStreamReader * rti::routing::adapter::Connection::output_stream_discovery_reader
( ) [inline], [virtual]
```

Gets the output stream discovery **StreamReader** (p. 102).

Returns a **DiscoveryStreamReader** (p. 37) that is used by RTI Routing **Service** (p. 83) to discover output streams.

An output stream is a stream from which a StreamWrite can write data. Disposed scenarios, where an output stream disappears, are also notified using this **DiscoveryStreamReader** (p. 37).

The **StreamReaderListener** (p. 108) associated with this **DiscoveryStreamReader** (p. 37) is provided as a parameter to the method **AdapterPlugin::create_connection** (p. 30).

Implementations may return nullptr if they do not support stream discovery.

Exceptions

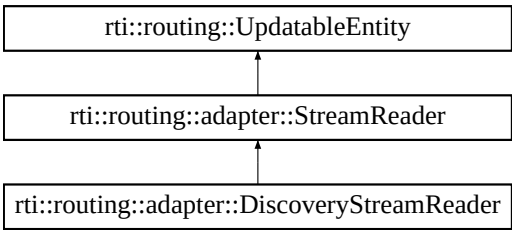
std::exception	

8.3 rti::routing::adapter::DiscoveryStreamReader Class Reference

Definition of a special **StreamReader** (p. 102) to read information about data streams.

```
#include <DiscoveryStreamReader.hpp>
```

Inheritance diagram for rti::routing::adapter::DiscoveryStreamReader:



Public Member Functions

- virtual void **take** (std::vector< **rti::routing::StreamInfo** * > &sample_seq)=0

  *Takes a collection of all available **StreamInfo** (p. 99) samples.*
- virtual void **return_loan** (std::vector< **rti::routing::StreamInfo** * > &sample_seq)=0

  *Returns a loan on the read or taken **StreamInfo** (p. 99) samples.*

8.3.1 Detailed Description

Definition of a special **StreamReader** (p. 102) to read information about data streams.

Multi-threading safety Safe All the operations on a **DiscoveryStreamReader** (p. 37) always serialized, no matter from which **Connection** (p. 31) it belongs.

8.3.2 Member Function Documentation

8.3.2.1 take()

```
virtual void rti::routing::adapter::DiscoveryStreamReader::take (
            std::vector< rti::routing::StreamInfo * > & sample_seq ) [pure virtual]
```

Takes a collection of all available **StreamInfo** (p. 99) samples.

When RTI Routing **Service** (p. 83) is done using the samples, it will 'return the loan' to the **DiscoveryStreamReader** (p. 37) by calling DiscoveryStreamReader::return_loan. It is guaranteed RTI Routing **Service** (p. 83) takes one outstanding loan at time.

The samples provided with this operation are considered 'removed' from the **DiscoveryStreamReader** (p. 37)'s cache, and they shall no longer be returned by subsequent DiscoveryStreamReader::take.

Parameters

sample_seq	<<*inout*>> Vector that will hold the **StreamInfo** (p. 99) samples. RTI Routing **Service** (p. 83) provides this vector to the **StreamReader** (p. 102) to fill. For each call, the size of the vector is zero.

Exceptions

std::exception	

8.3.2.2 return_loan()

```
virtual void rti::routing::adapter::DiscoveryStreamReader::return_loan (
            std::vector< rti::routing::StreamInfo * > & sample_seq ) [pure virtual]
```

Returns a loan on the read or taken **StreamInfo** (p. 99) samples.

RTI Routing **Service** (p. 83) calls this method to indicate that it is done accessing the collection of **StreamInfo** (p. 99) samples obtained by an earlier invocation of DiscoveryStreamReader::take.

Parameters

sample_seq	<<*in*>> Vector of loaned data samples.

Exceptions

std::exception	

8.4 rti::routing::processor::Input Class Reference

Generic Representation of a **Route** (p. 67)'s input.

`#include <Input.hpp>`

Public Member Functions

- template<typename Data , typename Info >
  TypedInput< Data, Info > **get** ()
  *Returns a typed version of this **Input** (p. 39).*
- template<typename Data >
  TypedInput< Data, dds::sub::SampleInfo > **get** ()
  *Returns a typed version of this **Input** (p. 39).*

8.4.1 Detailed Description

Generic Representation of a **Route** (p. 67)'s input.

This class provides a base representation of the inputs contained in a **Route** (p. 67). Since it is generic, this object does not allow you to perform any operation that requires dealing with the data provided by the underlying **rti::routing↩ ::adapter::StreamReader** (p. 102).

8.4.2 Member Function Documentation

8.4.2.1 get() [1/2]

```
template<typename Data , typename Info >
TypedInput< Data, Info > rti::routing::processor::Input::get ( )  [inline]
```

Returns a typed version of this **Input** (p. 39).

Note: This operation is not type-safe. It is the caller's responsibility to use the appropriate type.

See also

TypedInput (p. 126)

TypedInput<DataRep, Info> **Route<DataRep, InfoRep>::input(int32_t index)** (p. 70)

8.4.2.2 get() [2/2]

```
template<typename Data >
TypedInput< Data, dds::sub::SampleInfo > rti::routing::processor::Input::get ( )  [inline]
```

Returns a typed version of this **Input** (p. 39).

Note: This operation is not type-safe. It is the caller's responsibility to use the appropriate type.

See also

TypedInput (p. 126)

TypedInput (p. 126)<DataRep, dds::sub::SampleInfo> **Route<DataRep>::input(int32_t index)** (p. 70)

8.5 rti::routing::processor::LoanedSamples< T, U > Class Template Reference

Provides temporary access to a collection of samples (data and info) from a **TypedInput** (p. 126).

```
#include <LoanedSamples.hpp>
```

Public Types

- typedef **SampleIterator**< T, U > **iterator**

  The iterator type.

Public Member Functions

- **LoanedSamples** ()

  *Creates an empty **LoanedSamples** (p. 40) object.*
- ~**LoanedSamples** () throw ()

  *Automatically returns the loan to the **TypedInput** (p. 126) used to obtained these samples.*
- LoanedSample< T, U > **operator[ ]** (size_t index)

  Provides access to the underlying LoanedSample object in array-like syntax.
- int32_t **length** () const

  Gets the number of samples in this collection.
- void **return_loan** ()

  *Returns the samples to the **TypedInput** (p. 126) used to get these samples.*
- bool **has_infos** ()

  Returns whether the Info part is available for each Data item of this set of loaned samples.
- **iterator begin** ()

  Gets an iterator to the first sample.
- **iterator end** ()

  Gets an iterator to one past the last sample.
- **const_iterator begin** () const

  Gets an iterator to the first sample.
- **const_iterator end** () const

  Gets an iterator to one past the last sample.
- void **swap** (**LoanedSamples** &other) throw ()

  *Swaps two **LoanedSamples** (p. 40) containers.*

8.5.1 Detailed Description

template<typename T, typename U = dds::sub::SampleInfo>
class rti::routing::processor::LoanedSamples< T, U >

Provides temporary access to a collection of samples (data and info) from a **TypedInput** (p. 126).

Template Parameters

T	The sample data type representation. It has to match the type of the underlying **rti::routing::adapter::StreamReader** (p. 102) of the **TypedInput** (p. 126).
U	The sample info type representation. It has to match the type of the underlying **rti::routing::adapter::StreamReader** (p. 102) of the **TypedInput** (p. 126).

This STL-like container encapsulates a collection of loaned, read-only data samples rti::sub::LoanedSample::data() and rti::sub::LoanedSample::info() from a **TypedInput** (p. 126).

To obtain a **LoanedSamples** (p. 40), you need to call one of the read/take operations from a **TypedInput** (p. 126). The samples have to be eventually returned to the **TypedInput** (p. 126) by calling **rti::routing::adapter::StreamReader**↩ **::return_loan** (p. 107) on the underlying **rti::routing::adapter::StreamReader** (p. 102) of the **TypedInput** (p. 126). The destructor takes care of that, and the **return_loan()** (p. 43) function lets you do it explicitly if needed.

As a move-only type, copying a **LoanedSamples** (p. 40) is not allowed. If you need to return a **LoanedSamples** (p. 40) from a function or assign it to another variable, use `dds::core::move()` (or `std::move()` for C++11).

Iterators and overloaded subscript operators let you access the samples in this container, which are of the type rti↩ ::routing::processor::LoanedSample.

This code demonstrates how to access the info and data of each sample in a DataReader:

```
auto samples = reader.take();
for (const auto& sample : samples) {
    if (sample.info().valid()) {
        std::cout « sample.data() « std::endl;
    }
}
```

See also

dds::sub::LoanedSamples

8.5.2 Member Typedef Documentation

8.5.2.1 iterator

```
template<typename T , typename U = dds::sub::SampleInfo>
typedef  SampleIterator<T, U>  rti::routing::processor::LoanedSamples< T, U >::iterator
```

The iterator type.

8.5.3 Constructor & Destructor Documentation

8.5.3.1 LoanedSamples()

```
template<typename T , typename U = dds::sub::SampleInfo>
rti::routing::processor::LoanedSamples< T, U >::LoanedSamples ( )  [inline]
```

Creates an empty **LoanedSamples** (p. 40) object.

8.5.3.2 ~LoanedSamples()

```
template<typename T , typename U = dds::sub::SampleInfo>
rti::routing::processor::LoanedSamples< T, U >::~ LoanedSamples ( ) throw ( )   [inline]
```

Automatically returns the loan to the **TypedInput** (p. 126) used to obtained these samples.

See also

return_loan (p. 43)

References **rti::routing::processor::LoanedSamples< T, U >::return_loan()**.

8.5.4 Member Function Documentation

8.5.4.1 operator[]()

```
template<typename T , typename U = dds::sub::SampleInfo>
LoanedSample< T, U >  rti::routing::processor::LoanedSamples< T, U >::operator[] (
             size_t index )  [inline]
```

Provides access to the underlying LoanedSample object in array-like syntax.

Parameters

in	*index*	The index of the Sample. Allowed values are from 0 to **length()** (p. 43)-1.

Returns

A LoanedSample object that refers to data and info at the specified `index`.

8.5.4.2 length()

```
template<typename T , typename U = dds::sub::SampleInfo>
int32_t  rti::routing::processor::LoanedSamples< T, U >::length ( ) const  [inline]
```

Gets the number of samples in this collection.

8.5.4.3 return_loan()

```
template<typename T , typename U = dds::sub::SampleInfo>
void  rti::routing::processor::LoanedSamples< T, U >::return_loan ( )  [inline]
```

Returns the samples to the **TypedInput** (p. 126) used to get these samples.

Thes operation returns the samples by calling the **return_loan** (p. 107) operation on the underlying rti::routing::stream↩ ::StreamReader of the **TypedInput** (p. 126).

Note

Explicitly calling return_loan is optional, since the destructor does it implicitly.

This operation tells the **TypedInput** (p. 126) that the application is done accessing the collection of samples.

It is not necessary for an application to return the loans immediately after the call to read or take. However, as these buffers correspond to internal resources, the application should not retain them indefinitely.

Exceptions

std::exception	If there was an error returning the loan to the **TypedInput** (p. 126).

Referenced by **rti::routing::processor::LoanedSamples< T, U >::~LoanedSamples()**.

8.5.4.4 has_infos()

```
template<typename T , typename U = dds::sub::SampleInfo>
bool  rti::routing::processor::LoanedSamples< T, U >::has_infos ( )  [inline]
```

Returns whether the Info part is available for each Data item of this set of loaned samples.

8.5.4.5 begin() [1/2]

```
template<typename T , typename U = dds::sub::SampleInfo>
iterator  rti::routing::processor::LoanedSamples< T, U >::begin ( )  [inline]
```

Gets an iterator to the first sample.

8.5.4.6 end() [1/2]

```
template<typename T , typename U = dds::sub::SampleInfo>
iterator  rti::routing::processor::LoanedSamples< T, U >::end ( )  [inline]
```

Gets an iterator to one past the last sample.

8.5.4.7 begin() [2/2]

```
template<typename T , typename U = dds::sub::SampleInfo>
const_iterator  rti::routing::processor::LoanedSamples< T, U >::begin ( ) const  [inline]
```

Gets an iterator to the first sample.

8.5.4.8 end() [2/2]

```
template<typename T , typename U = dds::sub::SampleInfo>
const_iterator  rti::routing::processor::LoanedSamples< T, U >::end ( ) const  [inline]
```

Gets an iterator to one past the last sample.

8.5.4.9 swap()

```
template<typename T , typename U = dds::sub::SampleInfo>
void  rti::routing::processor::LoanedSamples< T, U >::swap (
            LoanedSamples< T, U > & other ) throw ( )   [inline]
```

Swaps two **LoanedSamples** (p. 40) containers.

8.6 rti::routing::Logger Class Reference

The singleton type used to configure RTI Routing **Service** (p. 83) verbosity.

```
#include <Logger.hpp>
```

Public Member Functions

- void **service_verbosity** (rti::config::Verbosity verbosity)

  *Sets the verbosity for the log messages generated at the RTI Routing **Service** (p. 83) level.*
- rti::config::Verbosity **service_verbosity** ()

  Getter for the same attribute.
- void **error** (const std::string &msg)

  Logs as message with EXCEPTION level.
- void **error** (const char ∗msg)

  *overload of **error(const std::string& msg)** (p. 47)*
- void **warn** (const std::string &msg)

  Logs as message with WARNING level.
- void **warn** (const char ∗msg)

  *overload of **warn(const std::string& msg)** (p. 47)*
- void **local** (const std::string &msg)

  Logs as message with WARNING level.
- void **local** (const char ∗msg)

  *overload of **local(const std::string& msg)** (p. 48)*
- void **remote** (const std::string &msg)

  Logs as message with WARNING level.
- void **remote** (const char ∗msg)

  *overload of **remote(const std::string& msg)** (p. 49)*
- void **debug** (const std::string &msg)

  Logs as message with WARNING level.
- void **debug** (const char ∗msg)

  *overload of **debug(const std::string& msg)** (p. 49)*

8.6.1 Detailed Description

The singleton type used to configure RTI Routing **Service** (p. 83) verbosity.

For configuring other aspects such as output file, print format or RTI Connext specific logging, use rti::config::Logger.

8.6.2 Member Function Documentation

8.6.2.1 service_verbosity() [1/2]

```
void rti::routing::Logger::service_verbosity (
            rti::config::Verbosity verbosity )  [inline]
```

Sets the verbosity for the log messages generated at the RTI Routing **Service** (p. 83) level.

Parameters

verbosity	<<*in*>> The verbosity of the service logs

8.6.2.2 service_verbosity() [2/2]

```
rti::config::Verbosity rti::routing::Logger::service_verbosity ( )  [inline]
```

Getter for the same attribute.

See also

service_verbosity(rti::config::Verbosity) (p. 46)

8.6.2.3 error() [1/2]

```
void rti::routing::Logger::error (
            const std::string & msg )  [inline]
```

Logs as message with EXCEPTION level.

The generated log will be part of logging stream of the running service if the service verbosity contains EXCEPTION messages. The result log message may include additional information according to the Connext logging configuration, such as Advlog Context, thread ID, line number, etc. Additionally, the result log message will contain a newline character at the end, so the format does not need to contain it.

Parameters

in	*msg*	Message to be logged

References **error()**.

Referenced by **error()**.

8.6.2.4 error() [2/2]

```
void rti::routing::Logger::error (
            const char * msg )  [inline]
```

overload of **error(const std::string& msg)** (p. 47)

8.6.2.5 warn() [1/2]

```
void rti::routing::Logger::warn (
            const std::string & msg )  [inline]
```

Logs as message with WARNING level.

This operation behaves similar to **error()** (p. 47) except the log level is WARNING.

Parameters

in	*msg*	Message to be logged

See also

rti::routing::Logger::error (p. 47)

References **warn()**.

Referenced by **rti::routing::Service::finalize_globals()**, and **warn()**.

8.6.2.6 warn() [2/2]

```
void rti::routing::Logger::warn (
            const char * msg )  [inline]
```

overload of **warn(const std::string& msg)** (p. 47)

8.6.2.7 local() [1/2]

```
void rti::routing::Logger::local (
            const std::string & msg )  [inline]
```

Logs as message with WARNING level.

This operation behaves similar to **error()** (p. 47) except the log level is STATUS_LOCAL.

Parameters

in	*msg*	Message to be logged

See also

rti::routing::Logger::error (p. 47)

References **local()**.

Referenced by **local()**.

8.6.2.8 local() [2/2]

```
void rti::routing::Logger::local (
            const char * msg )  [inline]
```

overload of **local(const std::string& msg)** (p. 48)

8.6.2.9 remote() [1/2]

```
void rti::routing::Logger::remote (
            const std::string & msg )  [inline]
```

Logs as message with WARNING level.

This operation behaves similar to **error()** (p. 47) except the log level is STATUS_REMOTE.

Parameters

in	*msg*	Message to be logged

See also

rti::routing::Logger::error (p. 47)

References **remote()**.

Referenced by **remote()**.

8.6.2.10 remote() [2/2]

```
void rti::routing::Logger::remote (
            const char * msg )  [inline]
```

overload of **remote(const std::string& msg)** (p. 49)

8.6.2.11 debug() [1/2]

```
void rti::routing::Logger::debug (
            const std::string & msg )  [inline]
```

Logs as message with WARNING level.

This operation behaves similar to **error()** (p. 47) except the log level is STATUS_DEBUG.

Parameters

in	*msg*	Message to be logged

See also

rti::routing::Logger::error (p. 47)

References **debug()**.

Referenced by **debug()**.

8.6.2.12 debug() [2/2]

```
void rti::routing::Logger::debug (
            const char * msg )  [inline]
```

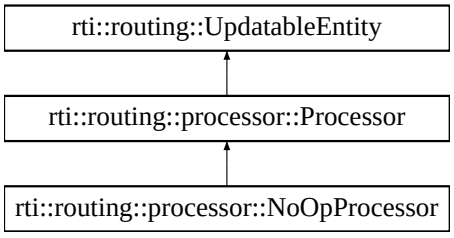
overload of **debug(const std::string& msg)** (p. 49)

8.7 rti::routing::processor::NoOpProcessor Class Reference

An empty implementation of the pure virtual interface **rti::routing::processor::Processor** (p. 56).

`#include <Processor.hpp>`

Inheritance diagram for rti::routing::processor::NoOpProcessor:



Additional Inherited Members

8.7.1 Detailed Description

An empty implementation of the pure virtual interface **rti::routing::processor::Processor** (p. 56).

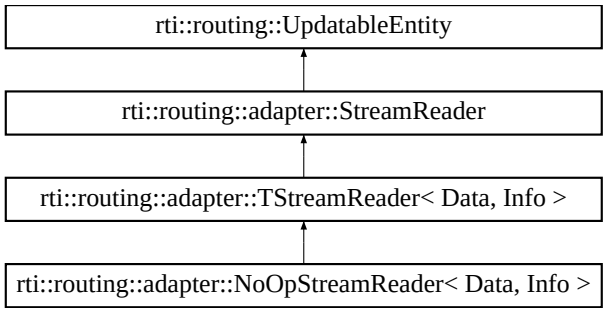
Your **Processor** (p. 56) implementation can inherit from this class and override just the methods that are of interest.

8.8 rti::routing::adapter::NoOpStreamReader< Data, Info > Class Template Reference

An empty implementation of the **TStreamReader** (p. 115) interface.

`#include <StreamReader.hpp>`

Inheritance diagram for rti::routing::adapter::NoOpStreamReader< Data, Info >:



Public Member Functions

- virtual void **take** (std::vector< Data * > &, std::vector< Info * > &) RTI_OVERRIDE
  Interface redefinition.
- virtual void **read** (std::vector< Data * > &, std::vector< Info * > &) RTI_OVERRIDE
  Interface redefinition.
- virtual void **take** (std::vector< Data * > &, std::vector< Info * > &, const **SelectorState** &) RTI_OVERRIDE
  Interface redefinition.
- virtual void **read** (std::vector< Data * > &, std::vector< Info * > &, const **SelectorState** &) RTI_OVERRIDE
  Interface redefinition.
- virtual void **return_loan** (std::vector< Data * > &, std::vector< Info * > &) RTI_OVERRIDE
  Interface redefinition.
- virtual void * **create_content_query** (void *, const dds::topic::Filter &) RTI_OVERRIDE
  Creates a query object that selects the data with the specified filter.
- virtual void **delete_content_query** (void *) RTI_OVERRIDE
  *Deletes a content query created from this **StreamReader** (p. 102).*

Additional Inherited Members

8.8.1 Detailed Description

template<typename Data, typename Info>
class rti::routing::adapter::NoOpStreamReader< Data, Info >

An empty implementation of the **TStreamReader** (p. 115) interface.

Each implemented operation performs as a no-op.

8.8.2 Member Function Documentation

8.8.2.1 take() [1/2]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::NoOpStreamReader< Data, Info >::take (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq )  [inline], [virtual]
```

Interface redefinition.

See also

StreamReader::take (p. 104)

Implements **rti::routing::adapter::TStreamReader< Data, Info >** (p. 119).

8.8.2.2 read() [1/2]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::NoOpStreamReader< Data, Info >::read (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq )  [inline], [virtual]
```

Interface redefinition.

See also

StreamReader::read (p. 104)

Implements **rti::routing::adapter::TStreamReader< Data, Info >** (p. 119).

8.8.2.3 take() [2/2]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::NoOpStreamReader< Data, Info >::take (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq,
            const  SelectorState & selector_state )  [inline], [virtual]
```

Interface redefinition.

See also

StreamReader::take(std::vector&, std::vector&,const SelectorState&);

Implements **rti::routing::adapter::TStreamReader< Data, Info >** (p. 120).

8.8.2.4 read() [2/2]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::NoOpStreamReader< Data, Info >::read (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq,
            const  SelectorState & selector_state )  [inline], [virtual]
```

Interface redefinition.

See also

StreamReader::read(std::vector&, std::vector&,const SelectorState&);

Implements **rti::routing::adapter::TStreamReader< Data, Info >** (p. 120).

8.8.2.5 return_loan()

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::NoOpStreamReader< Data, Info >::return_loan (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq )  [inline], [virtual]
```

Interface redefinition.

See also

StreamReader::return_loan (p. 107)

Implements **rti::routing::adapter::TStreamReader< Data, Info >** (p. 120).

8.8.2.6 create_content_query()

```
template<typename Data , typename Info >
virtual void *  rti::routing::adapter::NoOpStreamReader< Data, Info >::create_content_query (
            void * old_query_data,
            const dds::topic::Filter & filter )  [inline], [virtual]
```

Creates a query object that selects the data with the specified filter.

This operation allows to read data with a **SelectorState** (p. 83) that contains a query object returned by this operation.

A query object type is implementation dependent and it's guaranteed to to be used only within the same **StreamReader** (p. 102) that created it. Because a query object may be a expensive resource, this operation allows to receive a previously created query for a potential reuse and update of its filter.

Parameters

in	*old_query_data*	A previously created query object that that is provided for potential reuse and update of its filter.
in	*filter*	The query content filter

Returns

The new or updated query object.

Exceptions

std::exception	

Implements **rti::routing::adapter::StreamReader** (p. 107).

8.8.2.7 delete_content_query()

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::NoOpStreamReader< Data, Info >::delete_content_query (
            void * query_data )  [inline], [virtual]
```

Deletes a content query created from this **StreamReader** (p. 102).

Parameters

in	*query_data*	the query object to be deleted.

Implements **rti::routing::adapter::StreamReader** (p. 108).

8.9 rti::routing::processor::Output Class Reference

Generic Representation of a **Route** (p. 67)'s output.

`#include <Output.hpp>`

Public Member Functions

- template<typename Data , typename Info >
  TypedOutput< Data, Info > **get** ()
  *Returns a typed version of this **Output** (p. 55).*
- template<typename Data >
  TypedOutput< Data, dds::sub::SampleInfo > **get** ()
  *Returns a typed version of this **Output** (p. 55).*

8.9.1 Detailed Description

Generic Representation of a **Route** (p. 67)'s output.

This class provides a base representation of the outputs contained in a **Route** (p. 67). Since it is generic, this object does not allow performing any operation that requires dealing with the data passed to the underlying **rti::routing::adapter::StreamWriter** (p. 109).

8.9.2 Member Function Documentation

8.9.2.1 get() [1/2]

```
template<typename Data , typename Info >
TypedOutput< Data, Info > rti::routing::processor::Output::get ( )  [inline]
```

Returns a typed version of this **Output** (p. 55).

Note: This operation is not type-safe and it is the caller's responsibility to use the appropriate type.

See also

TypedOutput (p. 130)

`TypedOutput<DataRep, Info> Route<DataRep, InfoRep>::output(int32_t index)` (p. 72)

8.9.2.2 get() [2/2]

```
template<typename Data >
TypedOutput< Data, dds::sub::SampleInfo > rti::routing::processor::Output::get ( )  [inline]
```

Returns a typed version of this **Output** (p. 55).

Note: This operation is not type-safe and it is the caller's responsibility to use the appropriate type.

See also

TypedOutput (p. 130)

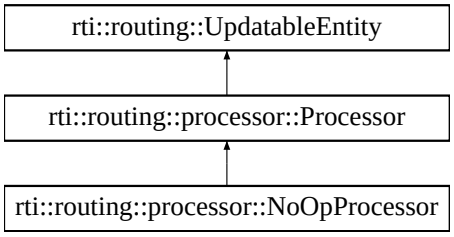
TypedOutput (p. 130)<DataRep, dds::sub::SampleInfo> **Route**<**DataRep**>**::output(int32_t index)** (p. 72)

8.10 rti::routing::processor::Processor Class Reference

Processor (p. 56) interface definition. Provides a way to process **Route** (p. 67) events and control the data forwarding process.

```
#include <Processor.hpp>
```

Inheritance diagram for rti::routing::processor::Processor:



Public Member Functions

- virtual void **on_input_enabled** (**Route** &route, **rti::routing::processor::Input** &input)=0

  Notification of the INPUT_ENABLED event.
- virtual void **on_input_disabled** (**Route** &route, **rti::routing::processor::Input** &input)=0

  *Notification of the **Input** (p. 39) disabled event.*
- virtual void **on_output_enabled** (**Route** &route, **rti::routing::processor::Output** &output)=0

  *Notification of the **Output** (p. 55) enabled event.*
- virtual void **on_output_disabled** (**Route** &route, **rti::routing::processor::Output** &output)=0

  *Notification of the **Output** (p. 55) disabled event.*
- virtual void **on_start** (**Route** &route)=0

  *Notification of the **Route** (p. 67) started event.*
- virtual void **on_stop** (**Route** &route)=0

  *Notification of the **Route** (p. 67) stopped event.*

- virtual void **on_run** (**Route** &route)=0

  *Notification of the **Route** (p. 67) `RUN` event.*
- virtual void **on_pause** (**Route** &route)=0

  *Notification of the **Route** (p. 67) stopped event.*
- virtual void **on_periodic_action** (**Route** &route)=0

  *Notification of the **Route** (p. 67) periodic action event.*
- virtual void **on_data_available** (**Route** &route)=0

  *Notification of the **Route** (p. 67) `DATA_AVAILABLE` event.*
- virtual ~**Processor** ()

  Virtual destructor.

8.10.1 Detailed Description

Processor (p. 56) interface definition. Provides a way to process **Route** (p. 67) events and control the data forwarding process.

A **Processor** (p. 56) receives event notifications from its **Route** (p. 67) owner in the form of operation callbacks. Each event occurrence will be dispatched to the **Processor** (p. 56) by calling the corresponding method.

Each dispatching method has a signature according to the event kind and data it is associated with. Each event is associated with a **Route** (p. 67) state; hence limitations and constraints may apply individually to each method.

Note that throwing an exception while processing any of the event notifications is allowed. In this situation, the **Route** (p. 67) owner will reject the events and none of the associated post conditions will be applied, including state transitions.

Multi-threading safety Partially Safe All operations on a concrete **Processor** (p. 56) object are safe and always serialized on a given Session. Operations on different **Processor** (p. 56) objects may be called concurrently if they belong to different Routes.

See also

ProcessorPlugin (p. 63)

Route (p. 67)

8.10.2 Constructor & Destructor Documentation

8.10.2.1 ~Processor()

```
virtual rti::routing::processor::Processor::~Processor ( )  [inline], [virtual]
```

Virtual destructor.

8.10.3 Member Function Documentation

8.10.3.1 on_input_enabled()

```
virtual void rti::routing::processor::Processor::on_input_enabled (
              Route & route,
              rti::routing::processor::Input & input )  [pure virtual]
```

Notification of the INPUT_ENABLED event.

This operation is called when an **Input** (p. 39) enabled event occurs that affects any of the inputs contained in the owner **Route** (p. 67).

This operation is called right after the affected **Input** (p. 39) has been enabled.

A running Routing **Service** (p. 83) instance will produce a log indicating the occurrence of the event. For example:
`[.../sessions/Session/routes/Route/inputs/Input1|ENABLE] stream=Example`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).
in	*input*	The just enabled **Input** (p. 39).

Exceptions

std::exception	Throwing an error in this operation will cause the input to be automatically disabled right after.

8.10.3.2 on_input_disabled()

```
virtual void rti::routing::processor::Processor::on_input_disabled (
              Route & route,
              rti::routing::processor::Input & input )  [pure virtual]
```

Notification of the **Input** (p. 39) disabled event.

This operation is called when an INPUT_DISABLED event occurs that affects any of the inputs contained in the owner **Route** (p. 67).

This operation is called right before the affected **Input** (p. 39) is disabled.

A running Routing **Service** (p. 83) instance will produce a log indicating the occurrence of the event. For example:
`[.../sessions/Session/routes/Route/inputs/Input1|DISABLE] stream=Example`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).
in	*input*	The **Input** (p. 39) about to be disabled.

Exceptions

std::exception	Throwing an error in this operation will cause the `input` to remain enabled.

8.10.3.3 on_output_enabled()

```
virtual void rti::routing::processor::Processor::on_output_enabled (
              Route & route,
              rti::routing::processor::Output & output )  [pure virtual]
```

Notification of the **Output** (p. 55) enabled event.

This operation is called when an `OUTPUT_DISABLED` event occurs that affects any of the outputs contained in the owner **Route** (p. 67).

This operation is called right after the affected **Output** (p. 55) has been enabled.

A running Routing **Service** (p. 83) instance will produce a log indicating the occurrence of the event. For example:
`[.../sessions/Session/routes/Route/outputs/Output1|ENABLE] stream=Example`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).
in	*output*	The just enabled output.

Exceptions

std::exception	Throwing an error in this operation will cause the `output` to be automatically disabled right after.

8.10.3.4 on_output_disabled()

```
virtual void rti::routing::processor::Processor::on_output_disabled (
              Route & route,
              rti::routing::processor::Output & output )  [pure virtual]
```

Notification of the **Output** (p. 55) disabled event.

This operation is called when an OUTPUT_DISABLED event occurs that affects any of the outputs contained in the owner **Route** (p. 67).

This operation is called right before the affected **Output** (p. 55) is disabled.

A running Routing **Service** (p. 83) instance will produce a log indicating the occurrence of the event. For example:
`[.../sessions/Session/routes/Route/outputs/Output1|DISABLE] stream=Example`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).
in	*output*	The **Output** (p. 55) about to be disabled.

Exceptions

std::exception	Throwing an error in this operation will cause the output to remain enabled.

8.10.3.5 on_start()

```
virtual void rti::routing::processor::Processor::on_start (
            Route & route )  [pure virtual]
```

Notification of the **Route** (p. 67) started event.

This operation is called right before the **Route** (p. 67) enters the STARTED state. At the time this operation is called, all the inputs and outputs within the **Route** (p. 67) are enabled.

A running Routing **Service** (p. 83) instance will produce a log indicating the occurrence of the event. For example:
`[.../domain_routes/DomainRoute/sessions/Session/routes/Route|START]`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).

Exceptions

std::exception	Throwing an error in this operation will cause the route to remain STOPPED.

8.10.3.6 on_stop()

```
virtual void rti::routing::processor::Processor::on_stop (
            Route & route )  [pure virtual]
```

Notification of the **Route** (p. 67) stopped event.

This operation is called right before the **Route** (p. 67) enters the `STOPPED` state. At the time this operation is called, all the inputs and outputs within the **Route** (p. 67) are still enabled.

A running Routing **Service** (p. 83) instance will produce a log indicating the occurrence of the event. For example:
`[.../domain_routes/DomainRoute/sessions/Session/routes/Route|STOP]`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).

Exceptions

std::exception	Throwing an error in this operation will cause the `route` to remain `STARTED`.

8.10.3.7 on_run()

```
virtual void rti::routing::processor::Processor::on_run (
              Route & route )  [pure virtual]
```

Notification of the **Route** (p. 67) `RUN` event.

This operation is called right before the **Route** (p. 67) enters the `RUNNING` state. This operation is called after the **Route** (p. 67) went to `STARTED` after a successful notification to this **Processor** (p. 56),

Note

> If the **Route** (p. 67) was manually paused before via an Administration call, this operation will not be called until a manual run operation is performed.

A Routing **Service** (p. 83) running instance will produce a log indicating the occurrence of the event. For example:
`[.../domain_routes/DomainRoute/sessions/Session/routes/Route|RUN]`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).

Exceptions

std::exception	Throwing an error in this operation will cause the `route` to remain `PAUSED`.

8.10.3.8 on_pause()

```
virtual void rti::routing::processor::Processor::on_pause (
            Route & route )  [pure virtual]
```

Notification of the **Route** (p. 67) stopped event.

This operation is called right before the **Route** (p. 67) enters the PAUSED state. At the time this operation is called, all the inputs and outputs within the **Route** (p. 67) are still enabled.

A Routing **Service** (p. 83) running instance will produce a log indicating the occurrence of the event. For example:
`[.../domain_routes/DomainRoute/sessions/Session/routes/Route|PAUSE]`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).

Exceptions

std::exception	Throwing an error in this operation will cause the route to remain RUNNING.

8.10.3.9 on_periodic_action()

```
virtual void rti::routing::processor::Processor::on_periodic_action (
            Route & route )  [pure virtual]
```

Notification of the **Route** (p. 67) periodic action event.

This operation is called periodically at the rate specified in the parent Session of the **Route** (p. 67) owner.

Periodic notifications can occur only while the **Route** (p. 67) is in the RUNNING state.

Implementations are allowed to access any of the **Input** (p. 39) and **Output** (p. 55) of the owner **Route** (p. 67) to read and write data, respectively.

A Routing **Service** (p. 83) running instance will produce a log indicating the occurrence of the event. For example:
`[.../domain_routes/DomainRoute/sessions/Session/routes/Route]  process event=PERODIC_ACTION`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).

Exceptions

std::exception	Throwing an error in this operation will produce a log message without affecting the **Route** (p. 67) state and notifications of this event to this **Processor** (p. 56) will keep occurring.

8.10.3.10 on_data_available()

```
virtual void rti::routing::processor::Processor::on_data_available (
              Route & route )  [pure virtual]
```

Notification of the **Route** (p. 67) DATA_AVAILABLE event.

This operation is called each time any of the inputs contained in the owner **Route** (p. 67) is notified about new data. Notifications of this event can occur only while the **Route** (p. 67) is in the RUNNING state.

Implementations are allowed to access any of the **Input** (p. 39) and **Output** (p. 55) of the owner **Route** (p. 67) to read and write data, respectively.

A running Routing **Service** (p. 83) instance will produce a log indicating the occurrence of the event. For example:
`[.../domain_routes/DomainRoute/sessions/Session/routes/Route] process event=DATA_ON_INPUTS`

Parameters

in	*route*	**Route** (p. 67) owner that contains this **Processor** (p. 56).

Exceptions

std::exception	Throwing an error in this operation will produce a log message without affecting the **Route** (p. 67) state and notifications of this event to this **Processor** (p. 56) will keep occurring.

8.11 rti::routing::processor::ProcessorPlugin Class Reference

The top-level plug-in class.

```
#include <ProcessorPlugin.hpp>
```

Public Member Functions

- virtual **Processor** * **create_processor** (**Route** &route, const **rti::routing::PropertySet** &properties)=0

  *Creates a **Route** (p. 67) **Processor** (p. 56).*
- virtual void **delete_processor** (**Route** &route, **Processor** *processor)=0

  *Deletes a **Route** (p. 67) **Processor** (p. 56).*
- virtual rti::config::LibraryVersion **get_version** () const
- virtual ~**ProcessorPlugin** ()

  Virtual destructor.

8.11.1 Detailed Description

The top-level plug-in class.

Represents a factory of **Processor** (p. 56).

8.11.2 Constructor & Destructor Documentation

8.11.2.1 ~ProcessorPlugin()

```
virtual rti::routing::processor::ProcessorPlugin::~ProcessorPlugin ( )  [inline], [virtual]
```

Virtual destructor.

8.11.3 Member Function Documentation

8.11.3.1 create_processor()

```
virtual  Processor * rti::routing::processor::ProcessorPlugin::create_processor (
             Route & route,
            const  rti::routing::PropertySet & properties )  [pure virtual]
```

Creates a **Route** (p. 67) **Processor** (p. 56).

This function is called when the **Route** (p. 67) containing the **Processor** (p. 56) is enabled.

A Routing **Service** (p. 83) running instance will product a log indicating the occurrence of this event. For example:
`[.../domain_routes/DomainRoute|START|/sessions/Session|START|/routes/Route|ENABLE]`

Required: yes

Parameters

route	<<*in*>> An object representation of the **Route** (p. 67) that owns the **Processor** (p. 56).
properties	<<*in*>> Configuration properties for the **Processor** (p. 56). These properties corresponds to the properties specified within the tag <processor>.

Returns

New **Processor** (p. 56) if successful. Cannot return nullptr.

Exceptions

std::exception	

Multi-threading safety Safe

8.11.3.2 delete_processor()

```
virtual void rti::routing::processor::ProcessorPlugin::delete_processor (
            Route & route,
            Processor * processor )  [pure virtual]
```

Deletes a **Route** (p. 67) **Processor** (p. 56).

This function is called when the **Route** (p. 67) containing the **Processor** (p. 56) is disabled.

Required: yes

Parameters

route	<<*in*>> An object representation of the **Route** (p. 67) that owns the **Processor** (p. 56).
processor	<<*in*>> **Processor** (p. 56) object to be deleted.

Exceptions

std::exception	

Multi-threading safety Safe

8.11.3.3 get_version()

```
virtual rti::config::LibraryVersion rti::routing::processor::ProcessorPlugin::get_version ( )
const  [inline], [virtual]
```

Returns

The version of this TransformationPlugin.

The version is used for logging purposes. It allows you to track which version of the TransformationPlugin the RTI Routing **Service** (p. 83) is using.

Default implementation of this operation returns the version of the required Connext libraries.

8.12 rti::routing::processor::Query Class Reference

Encapsulates a content query to select data from a **rti::routing::adapter::StreamReader** (p. 102).

`#include <Query.hpp>`

Inherits dds::core::Reference< rti::routing::processor::QueryHolder >.

Public Member Functions

- **Query** (RTI_RoutingServiceInput *input, const dds::topic::Filter & **filter**)
  *Creates a **Query** (p. 66) described by a filter on an input.*
- void **filter** (const dds::topic::Filter &filter)
  *Updates the current **Query** (p. 66) content filter.*

8.12.1 Detailed Description

Encapsulates a content query to select data from a **rti::routing::adapter::StreamReader** (p. 102).

Note that syntax for the query filter is StreamReader implementation dependent. For the DDS Adapter case, the syntax corresponds to that of the SQL filter.

A **Query** (p. 66) object holds a custom query obtained by calling **rti::routing::adapter::StreamReader::create_content_query** (p. 107). Likewise, the deletion of a **Query** (p. 66) object will call **rti::routing::adapter::StreamReader::delete_content_query** (p. 108).

8.12.2 Constructor & Destructor Documentation

8.12.2.1 Query()

```
rti::routing::processor::Query::Query (
            RTI_RoutingServiceInput * input,
            const dds::topic::Filter & filter )  [inline]
```

Creates a **Query** (p. 66) described by a filter on an input.

Parameters

in	*input*	The native **Input** (p. 39) to query the data from.
in	*filter*	Description of the content query based on an expression and parameters.

See also

dds::topic::Filter

8.12.3 Member Function Documentation

8.12.3.1 filter()

```
void rti::routing::processor::Query::filter (
            const dds::topic::Filter & filter )  [inline]
```

Updates the current **Query** (p. 66) content filter.

This operation will call **rti::routing::adapter::StreamReader::create_content_query** (p. 107) providing the current query content data this **Query** (p. 66) holds.

References **filter()**.

Referenced by **filter()**.

8.13 rti::routing::processor::Route Class Reference

Representation of the **Route** (p. 67) object that owns a **Processor** (p. 56).

```
#include <Route.hpp>
```

Public Member Functions

- int32_t **input_count** () const

  *Returns the total number of inputs in this **Route** (p. 67).*
- int32_t **output_count** () const

  *Returns the total number of outputs in this **Route** (p. 67).*
- bool **input_enabled** (int32_t index) const

  *Checks whether the **Input** (p. 39) at the specified index is enabled.*
- bool **output_enabled** (int32_t index) const

  *Checks whether the **Input** (p. 39) at the specified index is enabled.*
- **Input** & **input** (int32_t index)

  *Returns the **Input** (p. 39) object at the specified index.*
- **Input** & **input** (const std::string &name)

  *Returns the **Input** (p. 39) object with the specified name or throws an exception if the input with the given name is not found or if it is found but is not enabled.*
- template<typename DataRep , typename InfoRep >
  TypedInput< DataRep, InfoRep > **input** (int32_t index)

*Returns the **TypedInput** (p. 126) object at the specified index.*

- template<typename DataRep >
  TypedInput< DataRep > **input** (int32_t index)

  *Same as **input<DataRep, InfoRep>(int32_t)** (p. 70) with InfoRep = dds::sub::SampleInfo.*
- template<typename DataRep , typename InfoRep >
  TypedInput< DataRep, InfoRep > **input** (const std::string &name)

  *Same as **input<DataRep, InfoRep>(int32_t)** (p. 70) but using the configuration name of the input.*
- template<typename DataRep >
  TypedInput< DataRep > **input** (const std::string &name)

  Same as `input<DataRep, InfoRep>(std::string&)` with InfoRep = dds::sub::SampleInfo.
- **Output** & **output** (int32_t index)

  *Returns the **Output** (p. 55) object at the specified index.*
- **Output** & **output** (const std::string &name)

  *Returns the **Output** (p. 55) object with the specified name or throws an exception if the output with the given name is not found or if it is found but is not enabled.*
- template<typename DataRep , typename InfoRep >
  TypedOutput< DataRep, InfoRep > **output** (int32_t index)

  *Returns the **TypedOutput** (p. 130) object at the specified index.*
- template<typename DataRep >
  TypedOutput< DataRep > **output** (int32_t index)

  *Same as **output<DataRep, InfoRep>(int32_t)** (p. 72) with InfoRep = dds::sub::SampleInfo.*
- template<typename DataRep , typename InfoRep >
  TypedOutput< DataRep, InfoRep > **output** (const std::string &name)

  *Same as **output<DataRep, InfoRep>(int32_t)** (p. 72) but using the configuration name of the output.*
- template<typename DataRep >
  TypedOutput< DataRep > **output** (const std::string &name)

  Same as `output<DataRep, InfoRep>(std::string&)` with InfoRep = dds::sub::SampleInfo.
- template<typename DataRep , typename InfoRep >
  Inputs< DataRep, InfoRep > **inputs** ()

  Returns an accessor for the contained inputs.
- template<typename DataRep >
  Inputs< DataRep > **inputs** ()

  *Same as `Inputs<DataRep, InfoRep>` **inputs()** (p. 74) with InfoRep = dds::sub::SampleInfo.*
- template<typename DataRep , typename InfoRep >
  Outputs< DataRep, InfoRep > **outputs** ()

  Returns an accessor for the contained outputs.
- template<typename DataRep >
  Outputs< DataRep > **outputs** ()

  *Same as `Inputs<DataRep, InfoRep>` **outputs()** (p. 75) with InfoRep = dds::sub::SampleInfo.*
- void **period** (const dds::core::Duration &period)

  *Changes the event period of this **Route** (p. 67).*
- const std::string & **full_name** () const

  *Returns the fully qualified of this **Route** (p. 67), derived from the configuration.*

8.13.1 Detailed Description

Representation of the **Route** (p. 67) object that owns a **Processor** (p. 56).

This class provides an interface to access the capabilities of a **Route** (p. 67), such as retrieving any of its inputs and outputs. Usage of this class is restricted within the **Processor** (p. 56) context where the object is passed.

An example of how to use this object is shown below. This example illustrates a simple use case of reading from all inputs and writing to all outputs, assuming that the sample data and info have the same representation and type.

```
 {.c++}
using dds::core::xtypes::DynamicData;
using dds::sub::SampleInfo;
using namespace rti::routing::processor;
...
for (auto input : route.inputs<DynamicData>()) {
    LoanedSamples<DynamicData> samples = input->read();
    for (auto output : route.outputs<DynamicData>()) {
        for(auto sample = samples) {
             output.write(sample.data(), sample.info());
        }
    }
}
```

8.13.2 Member Function Documentation

8.13.2.1 input_count()

```
int32_t rti::routing::processor::Route::input_count ( ) const  [inline]
```

Returns the total number of inputs in this **Route** (p. 67).

8.13.2.2 output_count()

```
int32_t rti::routing::processor::Route::output_count ( ) const  [inline]
```

Returns the total number of outputs in this **Route** (p. 67).

8.13.2.3 input_enabled()

```
bool rti::routing::processor::Route::input_enabled (
             int32_t index ) const  [inline]
```

Checks whether the **Input** (p. 39) at the specified index is enabled.

8.13.2.4 output_enabled()

```
bool rti::routing::processor::Route::output_enabled (
            int32_t index ) const  [inline]
```

Checks whether the **Input** (p. 39) at the specified index is enabled.

8.13.2.5 input() [1/6]

```
Input & rti::routing::processor::Route::input (
            int32_t index )  [inline]
```

Returns the **Input** (p. 39) object at the specified index.

Input (p. 39) indexes are assigned in increasing creation order, starting at 0. The creation order is the same order than the inputs are defined within the **Route** (p. 67) configuration.

The returned object can be used to construct the appropriate **TypedInput** (p. 126), which can be used for reading data.

Parameters

in	*index*	The index of the **Input** (p. 39) to be retrieved.

Returns

The **Input** (p. 39) at the specified index.

See also

TypedInput (p. 126)

Referenced by **input()**.

8.13.2.6 input() [2/6]

```
Input & rti::routing::processor::Route::input (
            const std::string & name )  [inline]
```

Returns the **Input** (p. 39) object with the specified name or throws an exception if the input with the given name is not found or if it is found but is not enabled.

NOTE: The condition that the input must be enabled will be removed in a future version. This is being tracked with issue ID ROUTING-1131.

Parameters

in	*name*	The input configuration name

See also

TypedInput::name (p. 127)

Exceptions

std::logic_error	

8.13.2.7 input() [3/6]

```
template<typename DataRep , typename InfoRep >
TypedInput< DataRep, InfoRep > rti::routing::processor::Route::input (
            int32_t index )  [inline]
```

Returns the **TypedInput** (p. 126) object at the specified index.

The returned object is typed so it can read samples whose type is `DataRep` for data, and `InfoRep` for info. These types must match the expected type representations of the underlying **rti::routing::adapter::StreamReader** (p. 102).

Typically, these template arguments are typed dds::core::xtypes::DynamicData and dds::sub::SampleInfo, respectively.

Warning

This operation performs no type integrity check. It is responsibility of the caller to ensure the provided template arguments match with the expected types of the input. Otherwise behavior accessing the input is undefined.

Returns

The **TypedInput** (p. 126) object at the specified index.

See also

input(int32_t) (p. 70)

TypedInput (p. 126)

References **input()**.

8.13.2.8 input() [4/6]

```
template<typename DataRep >
TypedInput< DataRep > rti::routing::processor::Route::input (
            int32_t index ) [inline]
```

Same as **input<DataRep, InfoRep>(int32_t)** (p. 70) with InfoRep = dds::sub::SampleInfo.

8.13.2.9 input() [5/6]

```
template<typename DataRep , typename InfoRep >
TypedInput< DataRep, InfoRep > rti::routing::processor::Route::input (
            const std::string & name ) [inline]
```

Same as **input<DataRep, InfoRep>(int32_t)** (p. 70) but using the configuration name of the input.

References **input()**.

8.13.2.10 input() [6/6]

```
template<typename DataRep >
TypedInput< DataRep > rti::routing::processor::Route::input (
            const std::string & name ) [inline]
```

Same as input<DataRep, InfoRep>(std::string&) with InfoRep = dds::sub::SampleInfo.

8.13.2.11 output() [1/6]

```
Output & rti::routing::processor::Route::output (
            int32_t index ) [inline]
```

Returns the **Output** (p. 55) object at the specified index.

Output (p. 55) indexes are assigned in increasing creation order, starting at 0. The creation order is the same order than the outputs are defined within the **Route** (p. 67) configuration.

The returned object can be used to construct the appropriate **TypedOutput** (p. 130), which can be used for writing data.

Parameters

in	*index*	The index of the **Output** (p. 55) to be retrieved.

Returns

The **Output** (p. 55) at the specified index.

See also

TypedOutput (p. 130)

Referenced by **output()**.

8.13.2.12 output() [2/6]

```
Output & rti::routing::processor::Route::output (
            const std::string & name )  [inline]
```

Returns the **Output** (p. 55) object with the specified name or throws an exception if the output with the given name is not found or if it is found but is not enabled.

NOTE: The condition that the input must be enabled will be removed in a future version. This is being tracked with issue ID ROUTING-1131.

Parameters

in	*name*	The output configuration name

See also

TypedOutput::name (p. 131)

Exceptions

std::logic_error	

8.13.2.13 output() [3/6]

```
template<typename DataRep , typename InfoRep >
TypedOutput< DataRep, InfoRep > rti::routing::processor::Route::output (
            int32_t index )  [inline]
```

Returns the **TypedOutput** (p. 130) object at the specified index.

The returned object is typed so it can write samples whose type is `DataRep` for data, and `InfoRep` for info. These types must match the expected type representations of the underlying **rti::routing::adapter::StreamWriter** (p. 109).

Typically, these template arguments are typed dds::core::xtypes::DynamicData and dds::sub::SampleInfo, respectively.

Warning

This operation performs no type integrity check. It is responsibility of the caller to ensure the provided template arguments match with the expected types of the output. Otherwise behavior accessing the output is undefined.

Returns

The **TypedOutput** (p. 130) object at the specified index.

See also

input(int32_t) (p. 70)

References **output()**.

8.13.2.14 output() [4/6]

```
template<typename DataRep >
TypedOutput< DataRep > rti::routing::processor::Route::output (
            int32_t index ) [inline]
```

Same as **output<DataRep, InfoRep>(int32_t)** (p. 72) with InfoRep = dds::sub::SampleInfo.

8.13.2.15 output() [5/6]

```
template<typename DataRep , typename InfoRep >
TypedOutput< DataRep, InfoRep > rti::routing::processor::Route::output (
            const std::string & name ) [inline]
```

Same as **output<DataRep, InfoRep>(int32_t)** (p. 72) but using the configuration name of the output.

References **output()**.

8.13.2.16 output() [6/6]

```
template<typename DataRep >
TypedOutput< DataRep > rti::routing::processor::Route::output (
            const std::string & name ) [inline]
```

Same as output<DataRep, InfoRep>(std::string&) with InfoRep = dds::sub::SampleInfo.

8.13.2.17 inputs() [1/2]

```
template<typename DataRep , typename InfoRep >
Inputs< DataRep, InfoRep > rti::routing::processor::Route::inputs ( )  [inline]
```

Returns an accessor for the contained inputs.

The returned object that allows iterating over the contained inputs, treating them as `TypedInput<DataRep, InfoRep>`.

See also

`Inputs<DataRep, InfoRep>`

`TypedInput<DataRep, InfoRep>` **input(int32_t index)** (p. 70)

8.13.2.18 inputs() [2/2]

```
template<typename DataRep >
Inputs< DataRep > rti::routing::processor::Route::inputs ( )  [inline]
```

Same as `Inputs<DataRep, InfoRep>` **inputs()** (p. 74) with InfoRep = dds::sub::SampleInfo.

8.13.2.19 outputs() [1/2]

```
template<typename DataRep , typename InfoRep >
Outputs< DataRep, InfoRep > rti::routing::processor::Route::outputs ( )  [inline]
```

Returns an accessor for the contained outputs.

The returned object that allows iterating over the contained outputs, treating them as `TypedOutput<DataRep, InfoRep>`.

See also

`Outputs<DataRep, InfoRep>`

`TypedOutput<DataRep, InfoRep>` **input(int32_t index)** (p. 70)

8.13.2.20 outputs() [2/2]

```
template<typename DataRep >
Outputs< DataRep > rti::routing::processor::Route::outputs ( )  [inline]
```

Same as `Inputs<DataRep, InfoRep>` **outputs()** (p. 75) with InfoRep = dds::sub::SampleInfo.

8.13.2.21 period()

```
void rti::routing::processor::Route::period (
            const dds::core::Duration & period )  [inline]
```

Changes the event period of this **Route** (p. 67).

This operation can be called many times within the same periodic event, in which only the last call will have effect.

The operation will enable the periodic event in this route if this operation is called for the first time in a route with no period set in the configuration.

Parameters

in	*period*	New session period

8.13.2.22 full_name()

```
const std::string & rti::routing::processor::Route::full_name ( ) const  [inline]
```

Returns the fully qualified of this **Route** (p. 67), derived from the configuration.

8.14 rti::routing::processor::SampleIterator< T, U > Class Template Reference

A random-access iterator of LoanedSample.

`#include <SampleIterator.hpp>`

8.14.1 Detailed Description

template<typename T, typename U>
class rti::routing::processor::SampleIterator< T, U >

A random-access iterator of LoanedSample.

See also

rti::routing::processor::LoanedSamples (p. 40)

8.15 rti::routing::processor::Selector< Data, Info > Class Template Reference

An element that allows reading data that meet a set of specified attributes.

`#include <Input.hpp>`

Public Member Functions

- **Selector** (const **rti::routing::processor::TypedInput**< Data, Info > input)
  *Create a **Selector** (p. 77) for a **TypedInput** (p. 126).*
- **Selector** (const **Selector** &other)
  Copy constructor.
- **Selector** & **state** (const dds::sub::status::DataState &the_state)
  Select a specific dds::sub::status::DataState.
- **Selector** & **max_samples** (const int32_t count)
  Choose to only read/take up to a maximum number of samples.
- **Selector** & **instance** (const dds::core::InstanceHandle &the_handle)
  Select a specific instance to read/take.
- **Selector** & **next_instance** (const dds::core::InstanceHandle &the_handle)
  Select the instance after a specific instance.
- **Selector** & **filter** (const dds::topic::Filter &the_filter)
  Select samples based on a content filter parameters.
- **Selector** & **query** (const **rti::routing::processor::Query** &the_query)
  *Select samples based on a **rti::routing::processor::Query** (p. 66).*
- **LoanedSamples**< Data, Info > **take** ()
  *Take samples based on the state associated with this **Selector** (p. 77).*
- **LoanedSamples**< Data, Info > **read** ()
  *Read samples based on the state associated with this **Selector** (p. 77).*

8.15.1 Detailed Description

template<typename Data, typename Info>
class rti::routing::processor::Selector< Data, Info >

An element that allows reading data that meet a set of specified attributes.

The **Selector** (p. 77) class is used by the **TypedInput** (p. 126) to compose read and take operations.

A **Selector** (p. 77) has an associated **TypedInput** (p. 126) and configures the behavior of the read or take operation performed by that **TypedInput** (p. 126).

For example, to perform a read of at most 5 unread samples:

```
LoanedSamples<Foo> samples = reader.select()
    .read()
    .max_samples(5)
    .state(dds::sub::status::DataState::new_data());
```

Note

It's very important to consider that the behavior of this class is tightly coupled and dependent on the underlying **rti::routing::adapter::StreamReader** (p. 102) of the attached **TypedInput** (p. 126). In particular, the samples returned by a **Selector** (p. 77) depend on the behavior of the rti::routing::adapter::StreamReader::take(SelectorState) and rti::routing::adapter::StreamReader::read(SelectorState). If the implementation of these operations behave according to the requirements, then the behavior offered by this class will be consistent.

See also

rti::routing::adapter::SelectorState (p. 83)

dds::sub::DataReader::Selector for equivalent semantics and behavior with DataReaders

8.15.2 Constructor & Destructor Documentation

8.15.2.1 Selector() [1/2]

```
template<typename Data , typename Info >
rti::routing::processor::Selector< Data, Info >::Selector (
            const  rti::routing::processor::TypedInput< Data, Info > input )  [inline]
```

Create a **Selector** (p. 77) for a **TypedInput** (p. 126).

Selectors are created with the default filter state of the **TypedInput** (p. 126).

Parameters

in	*input*	The **TypedInput** (p. 126) that this **Selector** (p. 77) is attached to.

8.15.2.2 Selector() [2/2]

```
template<typename Data , typename Info >
rti::routing::processor::Selector< Data, Info >::Selector (
            const  Selector< Data, Info > & other )  [inline]
```

Copy constructor.

8.15.3 Member Function Documentation

8.15.3.1 state()

```
template<typename Data , typename Info >
Selector &  rti::routing::processor::Selector< Data, Info >::state (
            const dds::sub::status::DataState & the_state )  [inline]
```

Select a specific dds::sub::status::DataState.

By setting the DataState, you can specify the state of the samples that should be read or taken. The DataState of a sample encapsulates the SampleState, ViewState, and InstanceState of a sample.

Parameters

in	*the_state*	The selected DataState

8.15.3.2 max_samples()

```
template<typename Data , typename Info >
Selector &  rti::routing::processor::Selector< Data, Info >::max_samples (
            const int32_t count )  [inline]
```

Choose to only read/take up to a maximum number of samples.

Parameters

in	*count*	The maximum number of samples to read/take.

8.15.3.3 instance()

```
template<typename Data , typename Info >
Selector &  rti::routing::processor::Selector< Data, Info >::instance (
             const dds::core::InstanceHandle & the_handle )  [inline]
```

Select a specific instance to read/take.

This operation causes the subsequent read or take operation to access only samples belonging the single specified instance whose handle is `the_handle`.

Upon successful completion, the data collection will contain samples all belonging to the same instance.

The subsequent read/take may operation may fail if the InstanceHandle does not correspond to an existing data-object known to the **TypedInput** (p. 126).

Parameters

in	*the_handle*	The handle of the instance to select

See also

rti::routing::adapter::SelectorState (p. 83)

dds::sub::DataReader::Selector::instance

8.15.3.4 next_instance()

```
template<typename Data , typename Info >
Selector &  rti::routing::processor::Selector< Data, Info >::next_instance (
             const dds::core::InstanceHandle & the_handle )  [inline]
```

Select the instance after a specific instance.

This operation causes the subsequent read or take operation to access only samples belonging a single instance whose handle is considered 'next' after the provided InstanceHandle, `the_handle`.

The accessed samples will all belong to the 'next' instance with InstanceHandle 'greater' than the specified previous handle that has available samples.

The special value dds::core::InstanceHandle::nil() is guaranteed to be 'less than' any valid instance_handle. So the use of the parameter value previous_handle == dds::core::InstanceHandle::nil() will return the samples for the instance that has the smallest instance_handle among all the instances that contain available samples.

Note that it is possible to call the dds::sub::next_instance(const dds::core::InstanceHandle& h) operation with a previous_handle that does not correspond to an instance currently managed by the underlying **rti::routing::adapter**↩ **::StreamReader** (p. 102).

Parameters

in	*the_handle*	The reference instance. The instance after this one will be selected.

See also

rti::routing::adapter::SelectorState (p. 83)

dds::sub::DataReader::Selector::next_instance

8.15.3.5 filter()

```
template<typename Data , typename Info >
Selector &  rti::routing::processor::Selector< Data, Info >::filter (
             const dds::topic::Filter & the_filter )  [inline]
```

Select samples based on a content filter parameters.

The effect of using this manipulator is that the subsequent read/take will filter the samples based on the dds::topic::Filter. If the **Input** (p. 39) has no samples that meet the constraints, the read/take will not return any data.

This selection is applied in combination with the other settings of this **Selector** (p. 77).

Parameters

in	*the_filter*	

8.15.3.6 query()

```
template<typename Data , typename Info >
Selector &  rti::routing::processor::Selector< Data, Info >::query (
             const  rti::routing::processor::Query & the_query )  [inline]
```

Select samples based on a **rti::routing::processor::Query** (p. 66).

When passing a **Query** (p. 66), the effect of calling this method is that the underlying **rti::routing::adapter::Stream↩ Reader** (p. 102) shall filter the samples based only on the **Query** (p. 66)'s settings.

Using this manipulator will always override the selection specified with **Selector::filter** (p. 81), no matter the order in which operations are called. To reset this selection, you can call this operation passing dds::core::null.

Parameters

in	*the_query*	The **Query** (p. 66) to read/take with.

8.15.3.7 take()

```
template<typename Data , typename Info >
LoanedSamples< Data, Info >  rti::routing::processor::Selector< Data, Info >::take ( )  [inline]
```

Take samples based on the state associated with this **Selector** (p. 77).

Note

This operation will call rti::routing::adapter::StreamReader::take(SelectorState) on the underlying **rti::routing::adapter::StreamReader** (p. 102).

Exceptions

std::exception	

Returns

rti::routing::processor::LoanedSamples (p. 40)

8.15.3.8 read()

```
template<typename Data , typename Info >
LoanedSamples< Data, Info >  rti::routing::processor::Selector< Data, Info >::read ( )  [inline]
```

Read samples based on the state associated with this **Selector** (p. 77).

Note

This operation will call rti::routing::adapter::StreamReader::read(SelectorState) on the underlying **rti::routing::adapter::StreamReader** (p. 102).

Exceptions

std::exception	

Returns

rti::routing::processor::LoanedSamples (p. 40)

8.16 rti::routing::adapter::SelectorState Class Reference

Defines a set of attributes that can be used to read a subset of data from **StreamReader** (p. 102).

`#include <StreamReader.hpp>`

Inherits rti::core::NativeValueType< SelectorState >.

8.16.1 Detailed Description

Defines a set of attributes that can be used to read a subset of data from **StreamReader** (p. 102).

8.17 rti::routing::Service Class Reference

The RTI Routing **Service** (p. 83).

`#include <Service.hpp>`

Inherits dds::core::Reference< RoutingServiceImpl >.

Public Member Functions

- **Service** (const **ServiceProperty** &property)

  *Creates a RTI Routing **Service** (p. 83) instance.*
- template<typename HookFunc >
  Service (const **ServiceProperty** &property, HookFunc &shutdown_hook)

  *Creates a RTI Routing **Service** (p. 83) instance.*
- void **start** ()

  *Starts RTI Routing **Service** (p. 83).*
- void **stop** ()

  *Stops RTI Routing **Service** (p. 83).*
- void **attach_adapter_plugin** (**rti::routing::adapter::AdapterPlugin** *adapter_plugin, const std::string &plugin_name)

  *Attaches an Adapter plugin to be used by RTI Routing **Service** (p. 83) when it is started.*
- void **attach_processor_plugin** (**rti::routing::processor::ProcessorPlugin** *processor_plugin, const std::string &plugin_name)

  *Attaches a Processor plugin to be used by RTI Routing **Service** (p. 83) when it is started.*
- void **attach_transformation_plugin** (**rti::routing::transf::TransformationPlugin** *transformation_plugin, const std::string &plugin_name)

  *Attaches a Transformation plugin to be used by RTI Routing **Service** (p. 83) when it is started.*
- CommandReply **execute_command** (const CommandRequest &request)

  Executes an Administration command on this service.
- CommandReply & **execute_command** (CommandReply &reply, const CommandRequest &request)

  Executes an Administration command on this service.

Static Public Member Functions

- static void **finalize_globals** ()

8.17.1 Detailed Description

The RTI Routing **Service** (p. 83).

8.17.2 Constructor & Destructor Documentation

8.17.2.1 Service() [1/2]

```
rti::routing::Service::Service (
            const  ServiceProperty & property )  [inline], [explicit]
```

Creates a RTI Routing **Service** (p. 83) instance.

Parameters

property	<<*in*>> The property to configure RTI Routing **Service** (p. 83) instance.

Multi-threading safety On non-Linux, non-Windows systems (i.e. VxWorks): UNSAFE for multiple threads to simultaneously make the FIRST call to any of the **Service** (p. 83) constructors. Subsequent calls are thread safe. On Windows and Linux systems, these calls are thread-safe.

8.17.2.2 Service() [2/2]

```
template<typename HookFunc >
rti::routing::Service::Service (
            const  ServiceProperty & property,
            HookFunc & shutdown_hook )  [inline]
```

Creates a RTI Routing **Service** (p. 83) instance.

A callable shutdown hook can optionally be provided to handle the shutdown command received through remote administration. Upon reception of this command, RTI Routing **Service** (p. 83) will notify the installed hook.

The following example shows simple implementation of a shutdown hook that sets a boolean flag to true when invoked:

```
// Hook implementation
struct FlagShutdownHook {
```

```
public:
    FlagShutdownHook(bool& the_flag) : flag(the_flag) {
        flag = false;
    }
    void operator()() {
        flag = true;
    }
    bool& flag;
};
// Instantiate service with hook
bool my_flag;
FlagShutdownHook my_hook(my_flag)
rti::routing::Service routing_service(
      property,
      my_hook);
...
```

Parameters

in	*property*	The property to configure RTI Routing **Service** (p. 83) instance.
in	*shutdown_hook*	Callable object to handle the shutdown command. The expected type is a void operator()() (C++11 equivalent is std::function<void()>)

Multi-threading safety On non-Linux, non-Windows systems (i.e. VxWorks): UNSAFE for multiple threads to simultaneously make the FIRST call to any of the **Service** (p. 83) constructors. Subsequent calls are thread safe. On Windows and Linux systems, these calls are thread-safe.

8.17.3 Member Function Documentation

8.17.3.1 start()

```
void rti::routing::Service::start ( )  [inline]
```

Starts RTI Routing **Service** (p. 83).

This is a non-blocking operation. RTI Routing **Service** (p. 83) will create its own set of threads to perform its tasks.

8.17.3.2 stop()

```
void rti::routing::Service::stop ( )  [inline]
```

Stops RTI Routing **Service** (p. 83).

This operation will bloc the instance is fully stopped.

8.17.3.3 attach_adapter_plugin()

```
void rti::routing::Service::attach_adapter_plugin (
              rti::routing::adapter::AdapterPlugin * adapter_plugin,
             const std::string & plugin_name )  [inline]
```

Attaches an Adapter plugin to be used by RTI Routing **Service** (p. 83) when it is started.

By using this function an adapter can be statically compiled, created in your application and have the service load it, instead of registering a shared library and a create function in the configuration.

The name passed in this function is the name that has to be used in the configuration to instantiate connections from the plugin.

Example:

```
rti::routing::RoutingService service(property);
service.attach_adapter_plugin(myAdapter, "MyAdapter");
service.start();
...
```

And our configuration would look like this:

```
<dds>
 <!-- No need to register the plugin in
      <plugin_library><adapter_plugin>
 -->
 <routing_service name="example">
  <domain_route name="myadapter_to_dds">
    <connection name="MyConnection" plugin_name="MyAdapter">
      ...
    </connection>
    ...
  </domain_route>
 </routing_service>
</dds>
```

This function can be called as many times as desired to attach several plugins.

Precondition

Routing **Service** (p. 83) must not be started

Parameters

adapter_plugin	<<*in*>> The adapter plugin object to be attached. The object shall remain alive during the execution of the service. Once the plugin is attached, the memory is owned by RTI Routing **Service** (p. 83) and will delete it upon service stop.
plugin_name	<<*in*>> The name used for this plugin in the <connection> tag in the configuration.

8.17.3.4 attach_processor_plugin()

```
void rti::routing::Service::attach_processor_plugin (
              rti::routing::processor::ProcessorPlugin * processor_plugin,
             const std::string & plugin_name )  [inline]
```

Attaches a Processor plugin to be used by RTI Routing **Service** (p. 83) when it is started.

See also

attach_adapter_plugin (p. 85)

8.17.3.5 attach_transformation_plugin()

```
void rti::routing::Service::attach_transformation_plugin (
              rti::routing::transf::TransformationPlugin * transformation_plugin,
            const std::string & plugin_name )  [inline]
```

Attaches a Transformation plugin to be used by RTI Routing **Service** (p. 83) when it is started.

See also

attach_adapter_plugin (p. 85)

8.17.3.6 execute_command() [1/2]

```
CommandReply rti::routing::Service::execute_command (
            const CommandRequest & request )  [inline]
```

Executes an Administration command on this service.

This operation directly executes the specified `request` in this service as if it was directly received from an administration requester.

The request can represent any of the available administration operations as documented in the Administration chapter in the user's manual. This operation gives you an alternative way to control the service using the same remote administration interface but allowing you to call the operation directly.

This operation can also throw an exception if an internal error occurs.

Example:

```
 using namespace rti::routing;
 using namespace RTI::Service;
 using namespace RTI::Service::Admin;
 // Instantiate a RoutingServicee
 RoutingService service(...);
 // Prepare a request for any command, for example stop the service
 CommandRequest request;
 request.action(CommandActionKind::UPDATE_ACTION);
 request.resource_identifier("/routing_services/MyRouter");
 dds::topic::topic_type_support<EntityState>::to_cdr_buffer(
         reinterpret_cast<std::vector<char> &>(request.octet_body()),
         EntityState(EntityStateKind::DISABLED));
// Execute and check result
CommandReply reply = service.execute_command(request);
if (reply.retcode().underlying() != CommandReplyRetcode::OK_RETCODE) {
    Logger.instance().error(
            std::string("Command returned: ") + reply.string_body());
    //handle
    ...
}
```

Parameters

in	*request*	Representation of the command to be executed

Returns

The reply with the result of the command execution.

8.17.3.7 execute_command() [2/2]

```
CommandReply & rti::routing::Service::execute_command (
            CommandReply & reply,
            const CommandRequest & request )  [inline]
```

Executes an Administration command on this service.

Similar to **execute_command(const CommandRequest& request)** (p. 87) except the caller provides the Command↩ Reply object.

Parameters

out	*reply*	Representation of the reply containing the result of the operation
in	*request*	Representation of the command to be executed

Returns

A reference to the same reply object provided as a parameter

8.17.3.8 finalize_globals()

```
static void rti::routing::Service::finalize_globals ( )  [inline], [static]
```

[DEPRECATED] Calling this function at the end of the application is no longer necessary. It will be removed in future versions.

References **rti::routing::Logger::warn()**.

8.18 rti::routing::ServiceProperty Class Reference

Configuration for a RTI Routing **Service** (p. 83) object.

`#include <ServiceProperty.hpp>`

Inherits rti::core::NativeValueType< ServiceProperty >.

Public Member Functions

- **ServiceProperty** ()

  Creates a property object with default settings.
- std::string **cfg_file** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **cfg_file** (const std::string &filename)

  *Path to an RTI Routing **Service** (p. 83) configuration file.*
- const std::vector< std::string > & **cfg_strings** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **cfg_strings** (const std::vector< std::string > &the_cfg_strings)

  XML configuration represented as strings.
- std::string **service_name** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **service_name** (const std::string &name)

  *The name of the RTI Routing **Service** (p. 83) configuration to run.*
- std::string **application_name** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **application_name** (const std::string &name)

  *Assigns a name to the execution of the RTI Routing **Service** (p. 83).*
- bool **enforce_xsd_validation** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **enforce_xsd_validation** (const bool enforce_xsd_validation)

  Controls whether the service applies XSD validation to the loaded configuration.
- int **domain_id_base** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **domain_id_base** (const int domain_id)

  Value that is added to the domain IDs of the domain routes in the XML configuration.
- std::string **plugin_search_path** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **plugin_search_path** (const std::string &path)

  This path is used to look for plugin libraries.
- bool **dont_start_service** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **dont_start_service** (const bool not_start)

  *Set this to true to if you do not want RTI Routing **Service** (p. 83) enabled when RoutingServcie::start is called.*
- bool **enable_administration** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **enable_administration** (const bool enable)

  Set this to true to enable remote administration or false to disable it.
- int **administration_domain_id** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **administration_domain_id** (const int domain_id)

  *If **ServiceProperty::enable_administration** (p. 94) is true, this is the domain ID to use for remote administration.*
- bool **enable_monitoring** () const

  Getter (see setter with the same name)
- **ServiceProperty** & **enable_monitoring** (const bool &enable)

Set it to true to enable remote monitoring or false to disable it.

- int **monitoring_domain_id** () const

  Getter (see setter with the same name)

- **ServiceProperty** & **monitoring_domain_id** (const int &domain_id)

  *If **ServiceProperty::enable_monitoring** (p. 95) is true, this is the domain ID to use for remote monitoring.*

- bool **skip_default_files** () const

  Getter (see setter with the same name)

- **ServiceProperty** & **skip_default_files** (const bool &skip)

  *Set it to true to avoid loading the standard files usually loaded by RTI Routing **Service** (p. 83).*

- bool **identify_execution** () const

  Getter (see setter with the same name)

- **ServiceProperty** & **identify_execution** (const bool &identify)

  *Set this to true to append the host name and process ID to the RTI Routing **Service** (p. 83) execution name.*

- std::string **license_file_name** () const

  Getter (see setter with the same name)

- **ServiceProperty** & **license_file_name** (const std::string &filename)

  *Path to an RTI Routing **Service** (p. 83) license file.*

- std::map< std::string, std::string > **user_environment** () const

  Getter (see setter with the same name)

- **ServiceProperty** & **user_environment** (const std::map< std::string, std::string > &user_environment)

  Dictionary of user variables. The dictionary provides a parallel way to expand XML configuration variables in the form , when they are not defined in the environment.

8.18.1 Detailed Description

Configuration for a RTI Routing **Service** (p. 83) object.

8.18.2 Constructor & Destructor Documentation

8.18.2.1 ServiceProperty()

```
rti::routing::ServiceProperty::ServiceProperty ( )  [inline]
```

Creates a property object with default settings.

8.18.3 Member Function Documentation

8.18.3.1 cfg_file() [1/2]

```
std::string rti::routing::ServiceProperty::cfg_file ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **cfg_file()**.

8.18.3.2 cfg_file() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::cfg_file (
            const std::string & filename )  [inline]
```

Path to an RTI Routing **Service** (p. 83) configuration file.

[default] empty string.

References **cfg_file()**.

8.18.3.3 cfg_strings() [1/2]

```
const std::vector< std::string > & rti::routing::ServiceProperty::cfg_strings ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **cfg_strings()**.

8.18.3.4 cfg_strings() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::cfg_strings (
            const std::vector< std::string > & the_cfg_strings )  [inline]
```

XML configuration represented as strings.

An array of strings that altogether make up an XML document to configure RTI Routing **Service** (p. 83). This parameter is used only if **ServiceProperty::cfg_file** (p. 90) is empty.

The reason for using an array instead of one single string is to get around the limited size of literal strings. In general, if you create the XML string dynamically the vector needs only one element.

[default] empty.

References **cfg_strings()**.

8.18.3.5 service_name() [1/2]

```
std::string rti::routing::ServiceProperty::service_name ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **service_name()**.

8.18.3.6 service_name() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::service_name (
            const std::string & name )  [inline]
```

The name of the RTI Routing **Service** (p. 83) configuration to run.

This is the name used to find the <routing_service> XML tag in the configuration file; the name that will be used to refer to this execution in remote administration and monitoring.

[default] empty string.

References **service_name()**.

8.18.3.7 application_name() [1/2]

```
std::string rti::routing::ServiceProperty::application_name ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **application_name()**.

8.18.3.8 application_name() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::application_name (
            const std::string & name )  [inline]
```

Assigns a name to the execution of the RTI Routing **Service** (p. 83).

Remote commands and status information will refer to the service using this name.

In addition, the name of DomainParticipants created by RTI Routing **Service** (p. 83) will be based on this name.

[default] empty string. If no application name is specified, RTI Routing **Service** (p. 83) will use **ServiceProperty::service_name** (p. 91).

References **application_name()**.

8.18.3.9 enforce_xsd_validation() [1/2]

```
bool rti::routing::ServiceProperty::enforce_xsd_validation ( ) const  [inline]
```

Getter (see setter with the same name)

References **enforce_xsd_validation()**.

Referenced by **enforce_xsd_validation()**.

8.18.3.10 enforce_xsd_validation() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::enforce_xsd_validation (
            const bool enforce_xsd_validation )  [inline]
```

Controls whether the service applies XSD validation to the loaded configuration.

[default] true

References **enforce_xsd_validation()**.

8.18.3.11 domain_id_base() [1/2]

```
int rti::routing::ServiceProperty::domain_id_base ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **domain_id_base()**.

8.18.3.12 domain_id_base() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::domain_id_base (
            const int domain_id )  [inline]
```

Value that is added to the domain IDs of the domain routes in the XML configuration.

By using this, an XML file can use relative domain IDs.

[default] 0

References **domain_id_base()**.

8.18.3.13 plugin_search_path() [1/2]

```
std::string rti::routing::ServiceProperty::plugin_search_path ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **plugin_search_path()**.

8.18.3.14 plugin_search_path() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::plugin_search_path (
            const std::string & path )  [inline]
```

This path is used to look for plugin libraries.

If the plugin class libraries specified in the XML file don't contain a full path, RTI Routing **Service** (p. 83) looks for them in this directory if not NULL.

[default] "." (working directory)

References **plugin_search_path()**.

8.18.3.15 dont_start_service() [1/2]

```
bool rti::routing::ServiceProperty::dont_start_service ( ) const  [inline]
```

Getter (see setter with the same name)

References **dont_start_service()**.

Referenced by **dont_start_service()**.

8.18.3.16 dont_start_service() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::dont_start_service (
            const bool not_start )  [inline]
```

Set this to true to if you do not want RTI Routing **Service** (p. 83) enabled when RoutingServcie::start is called.

RTI Routing **Service** (p. 83) can be enabled afterwards through remote administration.

[default] false

References **dont_start_service()**.

8.18.3.17 enable_administration() [1/2]

```
bool rti::routing::ServiceProperty::enable_administration ( ) const  [inline]
```

Getter (see setter with the same name)

References **enable_administration()**.

Referenced by **enable_administration()**.

8.18.3.18 enable_administration() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::enable_administration (
            const bool enable )  [inline]
```

Set this to true to enable remote administration or false to disable it.

[default] false

References **enable_administration()**.

8.18.3.19 administration_domain_id() [1/2]

```
int rti::routing::ServiceProperty::administration_domain_id ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **administration_domain_id()**.

8.18.3.20 administration_domain_id() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::administration_domain_id (
            const int domain_id )  [inline]
```

If **ServiceProperty::enable_administration** (p. 94) is true, this is the domain ID to use for remote administration.

Takes precedence over the XML configuration. If **ServiceProperty::enable_administration** (p. 94) is false, this value is not used even if remote administration is enabled in the XML configuration.

[default] 0

References **administration_domain_id()**.

8.18.3.21 enable_monitoring() [1/2]

```
bool rti::routing::ServiceProperty::enable_monitoring ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **enable_monitoring()**.

8.18.3.22 enable_monitoring() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::enable_monitoring (
            const bool & enable )  [inline]
```

Set it to true to enable remote monitoring or false to disable it.

[default] false

References **enable_monitoring()**.

8.18.3.23 monitoring_domain_id() [1/2]

```
int rti::routing::ServiceProperty::monitoring_domain_id ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **monitoring_domain_id()**.

8.18.3.24 monitoring_domain_id() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::monitoring_domain_id (
            const int & domain_id )  [inline]
```

If **ServiceProperty::enable_monitoring** (p. 95) is true, this is the domain ID to use for remote monitoring.

Takes precedence over the XML configuration. If **ServiceProperty::enable_monitoring** (p. 95) is false, this value is not used, even if remote monitoring is enabled in the XML configuration.

[default] 0

References **monitoring_domain_id()**.

8.18.3.25 skip_default_files() [1/2]

```
bool rti::routing::ServiceProperty::skip_default_files ( ) const  [inline]
```

Getter (see setter with the same name)

References **skip_default_files()**.

Referenced by **skip_default_files()**.

8.18.3.26 skip_default_files() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::skip_default_files (
            const bool & skip )  [inline]
```

Set it to true to avoid loading the standard files usually loaded by RTI Routing **Service** (p. 83).

Only the configuration in **ServiceProperty::cfg_file** (p. 90) or **ServiceProperty::cfg_strings** (p. 91) will be loaded.

[default] false

References **skip_default_files()**.

8.18.3.27 identify_execution() [1/2]

```
bool rti::routing::ServiceProperty::identify_execution ( ) const  [inline]
```

Getter (see setter with the same name)

References **identify_execution()**.

Referenced by **identify_execution()**.

8.18.3.28 identify_execution() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::identify_execution (
            const bool & identify )  [inline]
```

Set this to true to append the host name and process ID to the RTI Routing **Service** (p. 83) execution name.

Used to get unique names for remote administration and monitoring.

[default] false

References **identify_execution()**.

8.18.3.29 license_file_name() [1/2]

```
std::string rti::routing::ServiceProperty::license_file_name ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **license_file_name()**.

8.18.3.30 license_file_name() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::license_file_name (
            const std::string & filename )  [inline]
```

Path to an RTI Routing **Service** (p. 83) license file.

If not empty, this file is checked for a valid license; otherwise, default location will be used. This parameter is only used if your installation requires a license file.

[default] empty string.

References **license_file_name()**.

8.18.3.31 user_environment() [1/2]

```
std::map< std::string, std::string > rti::routing::ServiceProperty::user_environment ( ) const
[inline]
```

Getter (see setter with the same name)

References **user_environment()**.

Referenced by **user_environment()**.

8.18.3.32 user_environment() [2/2]

```
ServiceProperty & rti::routing::ServiceProperty::user_environment (
            const std::map< std::string, std::string > & user_environment )  [inline]
```

Dictionary of user variables. The dictionary provides a parallel way to expand XML configuration variables in the form , when they are not defined in the environment.

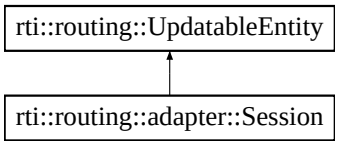
[default] empty

References **user_environment()**.

8.19 rti::routing::adapter::Session Class Reference

`#include <Session.hpp>`

Inheritance diagram for rti::routing::adapter::Session:



Public Member Functions

- virtual ~**Session** ()
  Virtual destructor.

8.19.1 Detailed Description

A **Session** (p. 99) is a concurrency unit within a **Connection** (p. 31) that has an associated set of StreamReaders and **StreamWriter** (p. 109). Access to the **StreamReader** (p. 102) and StreamWriters in the same **Session** (p. 99) is serialized by RTI Routing **Service** (p. 83).

In the XML configuration file, Sessions are associated with the tag <session> within a DomainRoute.

8.19.2 Constructor & Destructor Documentation

8.19.2.1 ~Session()

`virtual rti::routing::adapter::Session::~Session ( ) [inline], [virtual]`

Virtual destructor.

8.20 rti::routing::StreamInfo Class Reference

Definition of the stream information that RTI Routing **Service** (p. 83) needs to manage user data streams.

`#include <StreamInfo.hpp>`

Inherits rti::core::NativeValueType< StreamInfo >.

Public Member Functions

- **StreamInfo** (const std::string &the_stream_name, const std::string &the_type_name)

  *Constructs a **StreamInfo** (p. 99) with the minimum required information.*
- bool **disposed** () const

  Getter (see setter with the same name)
- **StreamInfo** & **disposed** (bool the_disposed)

  Sets the dispose flag in this object.
- std::string **stream_name** () const

  Getter (see setter with the same name)
- **StreamInfo** & **stream_name** (const std::string &the_stream_name)

  Sets the name of the stream.
- **TypeInfo** & **type_info** ()

  Getter (see setter with the same name)
- const **TypeInfo** & **type_info** () const

  Sets the type information associated with the stream.

8.20.1 Detailed Description

Definition of the stream information that RTI Routing **Service** (p. 83) needs to manage user data streams.

A stream in RTI Routing **Service** (p. 83) is a typed data channel to consume/produce data in a data domain.

8.20.2 Constructor & Destructor Documentation

8.20.2.1 StreamInfo()

```
rti::routing::StreamInfo::StreamInfo (
            const std::string & the_stream_name,
            const std::string & the_type_name )  [inline]
```

Constructs a **StreamInfo** (p. 99) with the minimum required information.

Parameters

the_stream_name	<<*in*>> The name of the stream.
the_type_name	<<*in*>> The name of the type associated with the data in the stream.

References **stream_name()**, **type_info()**, and **rti::routing::TypeInfo::type_name()**.

8.20.3 Member Function Documentation

8.20.3.1 disposed() [1/2]

```
bool rti::routing::StreamInfo::disposed ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **disposed()**.

8.20.3.2 disposed() [2/2]

```
StreamInfo & rti::routing::StreamInfo::disposed (
            bool the_disposed )  [inline]
```

Sets the dispose flag in this object.

Indicates whether the stream is a newly discovered stream or a disposed stream that no longer exists.

Parameters

the_disposed	<<*in*>> true if the stream is marked as disposed or false otherwise.

References **disposed()**.

8.20.3.3 stream_name() [1/2]

```
std::string rti::routing::StreamInfo::stream_name ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **stream_name()**, and **StreamInfo()**.

8.20.3.4 stream_name() [2/2]

```
StreamInfo & rti::routing::StreamInfo::stream_name (
            const std::string & the_stream_name )  [inline]
```

Sets the name of the stream.

Parameters

the_stream_name	<<*in*>> Name of the stream.

References **stream_name()**.

8.20.3.5 type_info() [1/2]

```
TypeInfo & rti::routing::StreamInfo::type_info ( )  [inline]
```

Getter (see setter with the same name)

References **type_info()**.

Referenced by **StreamInfo()**, and **type_info()**.

8.20.3.6 type_info() [2/2]

```
const  TypeInfo & rti::routing::StreamInfo::type_info ( ) const  [inline]
```

Sets the type information associated with the stream.

See also

TypeInfo (p. 137).

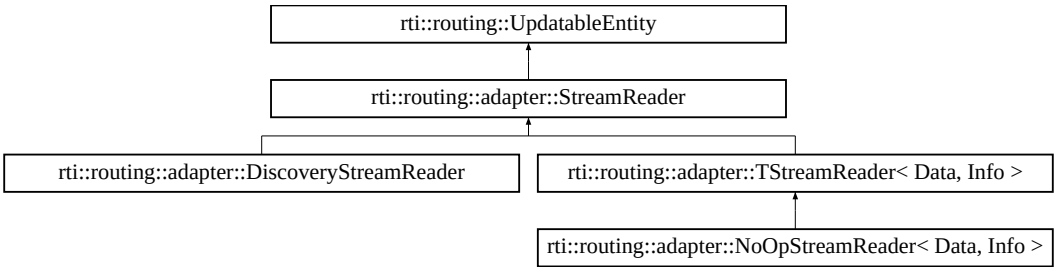
References **type_info()**.

8.21 rti::routing::adapter::StreamReader Class Reference

Provides a way to read samples of a specific type from a data domain. In the XML configuration file, StreamReaders are associated with the tag <input> within <route> and <auto_route>.

```
#include <StreamReader.hpp>
```

Inheritance diagram for rti::routing::adapter::StreamReader:



Public Member Functions

- virtual void **take** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  Takes a collection of all data samples and info samples available from an input stream.
- virtual void **read** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  *Variation of **StreamReader::take** (p. 104) where the returned samples will remain in the **StreamReader** (p. 102)'s cache, so they can be read again by subsequence **StreamReader::take** (p. 104) or **StreamReader::read** (p. 104) operations.*
- virtual void **take** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq, const **Selector**↩ **State** &selector_state)=0

  *Variation of **StreamReader::take** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).*
- virtual void **read** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq, const **Selector**↩ **State** &selector_state)=0

  *Variation of **StreamReader::read** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).*
- virtual void **return_loan** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  Returns a loan on the read or taken data samples and info samples.
- virtual void * **create_content_query** (void *old_query_data, const dds::topic::Filter &filter)=0

  Creates a query object that selects the data with the specified filter.
- virtual void **delete_content_query** (void *query_data)=0

  *Deletes a content query created from this **StreamReader** (p. 102).*
- virtual ~**StreamReader** ()

  Virtual destructor.

8.21.1 Detailed Description

Provides a way to read samples of a specific type from a data domain. In the XML configuration file, StreamReaders are associated with the tag <input> within <route> and <auto_route>.

Multi-threading safety Partially Safe All operations on a concrete **StreamReader** (p. 102) objects are safe and always serialized on a given **Session** (p. 99). Operations on different **StreamReader** (p. 102) objects can be called concurrently if the **StreamReader** (p. 102) objects belong to different Sessions.

8.21.2 Constructor & Destructor Documentation

8.21.2.1 ~StreamReader()

```
virtual rti::routing::adapter::StreamReader::~StreamReader ( )  [inline], [virtual]
```

Virtual destructor.

8.21.3 Member Function Documentation

8.21.3.1 take() [1/2]

```
virtual void rti::routing::adapter::StreamReader::take (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [pure virtual]
```

Takes a collection of all data samples and info samples available from an input stream.

When RTI Routing **Service** (p. 83) is done using the samples, it will 'return the loan' to the **StreamReader** (p. 102) by calling **StreamReader::return_loan** (p. 107). Note that multiple calls to this operation can be made before returning the loans.

The samples provided with this operation are considered 'removed' from the **StreamReader** (p. 102) cache, and they shall no longer be returned by subsequent **StreamReader::take** (p. 104) or **StreamReader::read** (p. 104) operations.

This operation represents the minimum required behavior by RTI Routing **Service** (p. 83) in order to forward data.

Parameters

sample_seq	<<*inout*>> Vector that will hold the output samples. RTI Routing **Service** (p. 83) provides this vector to the **StreamReader** (p. 102) to fill. For each call, the size of the vector is zero. The data representation associated with the samples will be given by the value of TypeInfo::data_representation_kind that is part of the **StreamInfo** (p. 99) object passed at **StreamWriter** (p. 109) creation time. Usually the data representation is dynamic type, which corresponds to dds::core::xtypes::DynamicData.
info_seq	<<*inout*>> Vector that will hold the output info samples. RTI Routing **Service** (p. 83) provides this vector to the **StreamReader** (p. 102) to fill. For each call, the size of the vector is zero. The implementation may leave the vector untouched if it does not support the concept of sample info. The info representation is dependent of the **StreamWriter** (p. 109) implementation. Usually when data representation is dynamic type, the sample info is dds::sub::SampleInfo.

Exceptions

std::exception	

Implemented in **rti::routing::adapter::TStreamReader< Data, Info >** (p. 118), and **rti::routing::adapter::TStreamReader< Data, Info >** (p. 121).

8.21.3.2 read() [1/2]

```
virtual void rti::routing::adapter::StreamReader::read (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [pure virtual]
```

Variation of **StreamReader::take** (p. 104) where the returned samples will remain in the **StreamReader** (p. 102)'s cache, so they can be read again by subsequence **StreamReader::take** (p. 104) or **StreamReader::read** (p. 104) operations.

See also

StreamReader::take (p. 104)

Implemented in **rti::routing::adapter::TStreamReader< Data, Info >** (p. 118), and **rti::routing::adapter::TStreamReader< Data, Info >** (p. 122).

8.21.3.3 take() [2/2]

```
virtual void rti::routing::adapter::StreamReader::take (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq,
            const  SelectorState & selector_state )  [pure virtual]
```

Variation of **StreamReader::take** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).

SelectorState (p. 83) is configured accordingly from the values set on **rti::routing::processor::Selector** (p. 77). This operation will be called only by custom **rti::routing::processor::Processor** (p. 56) implementations.

Parameters

sample_seq	
info_seq	
selector_state	<<*in*>> A description of the subset of samples that shall be returned.

See also

SelectorState (p. 83).

Implemented in **rti::routing::adapter::TStreamReader< Data, Info >** (p. 118), and **rti::routing::adapter::TStreamReader< Data, Info >** (p. 122).

8.21.3.4 read() [2/2]

```
virtual void rti::routing::adapter::StreamReader::read (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq,
            const  SelectorState & selector_state )  [pure virtual]
```

Variation of **StreamReader::read** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).

SelectorState (p. 83) is configured accordingly from the values set on **rti::routing::processor::Selector** (p. 77). This operation will be called only by custom **rti::routing::processor::Processor** (p. 56) implementations.

Parameters

sample_seq	
info_seq	
selector_state	<<*in*>> A description of the subset of samples that shall be returned.

See also

SelectorState (p. 83).

Implemented in **rti::routing::adapter::TStreamReader< Data, Info >** (p. 118), and **rti::routing::adapter::↩ TStreamReader< Data, Info >** (p. 122).

8.21.3.5 return_loan()

```
virtual void rti::routing::adapter::StreamReader::return_loan (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [pure virtual]
```

Returns a loan on the read or taken data samples and info samples.

RTI Routing **Service** (p. 83) calls this method to indicate that it is done accessing the collection of data samples and info samples obtained by an earlier invocation of any variation of **StreamReader::take** (p. 104).

Parameters

sample_seq	<<*in*>> Vector of loaned data samples.
info_seq	<<*in*>> Vector of loaned info samples.

Exceptions

std::exception	

Implemented in **rti::routing::adapter::TStreamReader< Data, Info >** (p. 119), and **rti::routing::adapter::↩ TStreamReader< Data, Info >** (p. 123).

8.21.3.6 create_content_query()

```
virtual void * rti::routing::adapter::StreamReader::create_content_query (
            void * old_query_data,
            const dds::topic::Filter & filter )  [pure virtual]
```

Creates a query object that selects the data with the specified filter.

This operation allows to read data with a **SelectorState** (p. 83) that contains a query object returned by this operation.

A query object type is implementation dependent and it's guaranteed to to be used only within the same **StreamReader** (p. 102) that created it. Because a query object may be a expensive resource, this operation allows to receive a previously created query for a potential reuse and update of its filter.

Parameters

in	*old_query_data*	A previously created query object that that is provided for potential reuse and update of its filter.
in	*filter*	The query content filter

Returns

The new or updated query object.

Exceptions

std::exception	

Implemented in **rti::routing::adapter::NoOpStreamReader< Data, Info >** (p. 53).

8.21.3.7 delete_content_query()

```
virtual void rti::routing::adapter::StreamReader::delete_content_query (
             void * query_data )  [pure virtual]
```

Deletes a content query created from this **StreamReader** (p. 102).

Parameters

in	*query_data*	the query object to be deleted.

Implemented in **rti::routing::adapter::NoOpStreamReader< Data, Info >** (p. 54).

8.22 StreamReaderListener Class Reference

Listener representation used by **StreamReader** (p. 102) to notify RTI Routing **Service** (p. 83) when new data is available.

```
#include <StreamReaderListenerForwarder.hpp>
```

8.22.1 Detailed Description

Listener representation used by **StreamReader** (p. 102) to notify RTI Routing **Service** (p. 83) when new data is available.

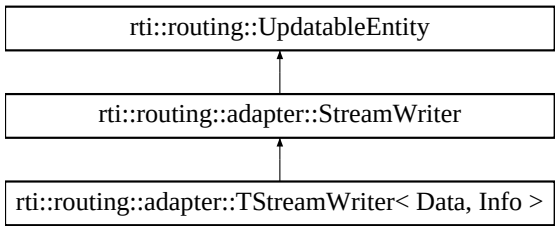
When a **StreamReader** (p. 102) receives new data, it shall call StreamReaderListener::on_data_available to wake up the associated **Session** (p. 99). After that, the **Session** (p. 99) will invoke any of the **StreamReader** (p. 102) operations to read the new data.

8.23 rti::routing::adapter::StreamWriter Class Reference

Provides a way to write samples of a specific type in a data domain.

`#include <StreamWriter.hpp>`

Inheritance diagram for rti::routing::adapter::StreamWriter:



Public Member Functions

- virtual int **write** (const std::vector< SamplePtr > &sample_seq, const std::vector< InfoPtr > &info_seq)=0
  *Writes a collection of data samples to the output stream associated with this **StreamWriter** (p. 109).*
- virtual ~**StreamWriter** ()
  Virtual destructor.

8.23.1 Detailed Description

Provides a way to write samples of a specific type in a data domain.

In the XML configuration file, A **StreamWriter** (p. 109) is associated with the tag <output> within <route> or <auto↩ _route>.

Multi-threading safety Partially Safe All operations on a concrete **StreamWriter** (p. 109) objects are safe and always serialized on a given **Session** (p. 99). Operations on different **StreamWriter** (p. 109) objects can be called concurrently if the **StreamWriter** (p. 109) objects belong to different Sessions.

8.23.2 Constructor & Destructor Documentation

8.23.2.1 ~StreamWriter()

```
virtual rti::routing::adapter::StreamWriter::~StreamWriter ( )  [inline], [virtual]
```

Virtual destructor.

8.23.3 Member Function Documentation

8.23.3.1 write()

```
virtual int rti::routing::adapter::StreamWriter::write (
            const std::vector< SamplePtr > & sample_seq,
            const std::vector< InfoPtr > & info_seq )  [pure virtual]
```

Writes a collection of data samples to the output stream associated with this **StreamWriter** (p. 109).

Parameters

sample_seq	<<*in*>> Vector of sample pointers. The data representation associated with the samples will be given by the value of TypeInfo::data_representation_kind that is part of the **StreamInfo** (p. 99) object passed at **StreamWriter** (p. 109) creation time. Usually the data representation is dynamic type, which corresponds to dds::core::xtypes::DynamicData.
info_seq	<<*in*>> Vector of sample info pointers. The info representation is dependent of the **StreamWriter** (p. 109) implementation. Usually when data representation is dynamic type, the sample info is dds::sub::SampleInfo.

Returns

Number of samples written.

Exceptions

std::exception	

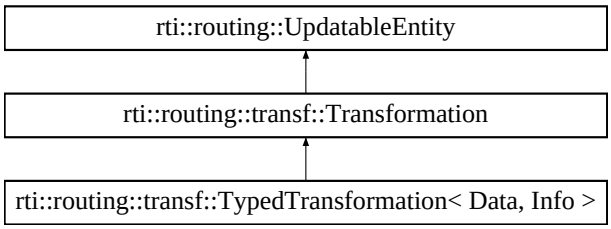
Implemented in **rti::routing::adapter::TStreamWriter< Data, Info >** (p. 125).

8.24 rti::routing::transf::Transformation Class Reference

Provides a way to transform input samples into output samples of a different format and/or content.

`#include <Transformation.hpp>`

Inheritance diagram for rti::routing::transf::Transformation:



Public Member Functions

- virtual void **transform** (std::vector< SamplePtr > &output_sample_seq, std::vector< InfoPtr > &output_info_↩ seq, const std::vector< SamplePtr > &input_sample_seq, const std::vector< InfoPtr > &input_info_seq)=0

  Transforms a vector of input samples into a vector of output samples.
- virtual void **return_loan** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  Returns a loan on the transformed data samples and info samples.

8.24.1 Detailed Description

Provides a way to transform input samples into output samples of a different format and/or content.

Multi-threading safety Partially Safe All operations on a concrete **Transformation** (p. 111) object are safe and always serialized on a given Output. Operations on different Transformations objects may be called concurrently if they belong to different Outputs from different Routes.

8.24.2 Member Function Documentation

8.24.2.1 transform()

```
virtual void rti::routing::transf::Transformation::transform (
            std::vector< SamplePtr > & output_sample_seq,
            std::vector< InfoPtr > & output_info_seq,
            const std::vector< SamplePtr > & input_sample_seq,
            const std::vector< InfoPtr > & input_info_seq )  [pure virtual]
```

Transforms a vector of input samples into a vector of output samples.

When RTI Routing **Service** (p. 83) is done using the output samples, it will 'return the loan' to the transformation by calling **Transformation::return_loan** (p. 112). It is guaranteed RTI Routing **Service** (p. 83) takes one outstanding loan at time.

The number of output samples can be different than the number of input samples.

The format associated with the input and output data samples and info samples depends on the format provided and consumed by the input's **rti::routing::adapter::StreamReader** (p. 102) and the output's **rti::routing::adapter::Stream↩ Writer** (p. 109).

For the built-in DDS adapter, the format of the samples is dds::core::xtypes::DynamicData and the format of the sample info is dds::sub::SampleInfo.

Parameters

output_sample_seq	<<*inout*>> Vector that will hold the output data samples. RTI Routing **Service** (p. 83) provides this vector to the **Transformation** (p. 111) to fill. For each call, the size of the vector is zero.
output_info_seq	<<*inout*>> Vector that will hold the input info samples. RTI Routing **Service** (p. 83) provides this vector to the **Transformation** (p. 111) to fill. For each call, the size of the vector is zero.
input_sample_seq	<<*in*>> Vector of input data samples. This vector contains the list of samples that need to be transformed before calling **rti::routing::adapter::StreamWriter::write** (p. 110).
input_info_seq	<<*in*>> Vector of input info sample. This vector contains the list of samples that need to be transformed before calling **rti::routing::adapter::StreamWriter::write** (p. 110).

Exceptions

std::exception	

Implemented in **rti::routing::transf::TypedTransformation< Data, Info >** (p. 135), and **rti::routing::transf::↩ TypedTransformation< Data, Info >** (p. 136).

8.24.2.2 return_loan()

```
virtual void rti::routing::transf::Transformation::return_loan (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [pure virtual]
```

Returns a loan on the transformed data samples and info samples.

RTI Routing **Service** (p. 83) calls this method to indicate that it is done accessing the collection of data samples and info samples obtained by an earlier invocation of **Transformation::transform** (p. 111).

Parameters

sample_seq	<<*in*>> Vector of loaned data samples.
info_seq	<<*in*>> Vector of loaned info samples.

Exceptions

std::exception	

Implemented in **rti::routing::transf::TypedTransformation< Data, Info >** (p. 135), and **rti::routing::transf::TypedTransformation< Data, Info >** (p. 137).

8.25 rti::routing::transf::TransformationPlugin Class Reference

The top-level plug-in class.

```
#include <TransformationPlugin.hpp>
```

Public Member Functions

- virtual **Transformation** * **create_transformation** (const **rti::routing::TypeInfo** &input_type_info, const **rti::routing::TypeInfo** &output_type_info, const **rti::routing::PropertySet** &properties)=0

  *Creates an Output **Transformation** (p. 111).*
- virtual void **delete_transformation** (**Transformation** *transformation)=0

  *Deletes a **Transformation** (p. 111).*
- virtual rti::config::LibraryVersion **get_version** () const
- virtual ~**TransformationPlugin** ()

  Virtual destructor.

8.25.1 Detailed Description

The top-level plug-in class.

Represents a factory of **Transformation** (p. 111).

8.25.2 Constructor & Destructor Documentation

8.25.2.1 ~TransformationPlugin()

```
virtual rti::routing::transf::TransformationPlugin::~TransformationPlugin ( )  [inline], [virtual]
```

Virtual destructor.

8.25.3 Member Function Documentation

8.25.3.1 create_transformation()

```
virtual  Transformation * rti::routing::transf::TransformationPlugin::create_transformation (
            const  rti::routing::TypeInfo & input_type_info,
            const  rti::routing::TypeInfo & output_type_info,
            const  rti::routing::PropertySet & properties )  [pure virtual]
```

Creates an Output **Transformation** (p. 111).

This function is called when the Output containing the transformation is enabled.

The format associated with the input and output types depends on the format provided by the input and output adapters.

For the built-in DDS adapter, the format of the types is DDS_TypeCode. **Required:** yes

Parameters

input_type_info	<<*in*>> Type information associated with the input samples.
output_type_info	<<*in*>> Type information associated with the output samples.
properties	<<*in*>> Configuration properties for the **Transformation** (p. 111). These properties corresponds to the properties specified within the tag <transformation>.

Returns

New **Transformation** (p. 111) if successful. Cannot return nullptr.

Exceptions

std::exception	

Multi-threading safety Safe

8.25.3.2 delete_transformation()

```
virtual void rti::routing::transf::TransformationPlugin::delete_transformation (
              Transformation * transformation )  [pure virtual]
```

Deletes a **Transformation** (p. 111).

This function is called when the Output containing the transformation is disabled.

Parameters

transformation	<<*in*>> **Transformation** (p. 111) to be deleted.

Exceptions

std::exception	

Multi-threading safety Safe

8.25.3.3 get_version()

```
virtual rti::config::LibraryVersion rti::routing::transf::TransformationPlugin::get_version ( )
const  [inline], [virtual]
```

Returns

The version of this **TransformationPlugin** (p. 113).

Version is used for logging purposes and allows you to track which version of the **TransformationPlugin** (p. 113) RTI Routing **Service** (p. 83) is using.

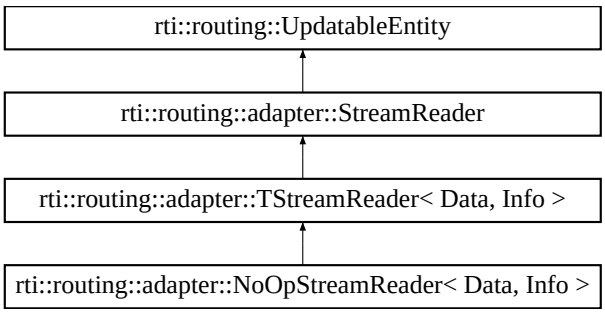
Default implementation of this operation returns the version of the required Connext libraries.

8.26 rti::routing::adapter::TStreamReader< Data, Info > Class Template Reference

A wrapper implementation of a **StreamReader** (p. 102) that provides a strongly-typed interface through template parameters for data and info representation.

```
#include <StreamReader.hpp>
```

Inheritance diagram for rti::routing::adapter::TStreamReader< Data, Info >:



Public Types

- typedef Data **DataRep**

  The data type.
- typedef Info **InfoRep**

  The info type.

Public Member Functions

- void **take** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq) RTI_FINAL

  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- void **read** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq) RTI_FINAL

  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- void **take** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq, const **SelectorState** &selector_state) RTI_FINAL

  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- void **read** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq, const **SelectorState** &selector_state) RTI_FINAL

  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- void **return_loan** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq) RTI_FINAL

  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- virtual void **take** (std::vector< Data * > &sample_seq, std::vector< Info * > &info_seq)=0

  Interface redefinition.
- virtual void **read** (std::vector< Data * > &sample_seq, std::vector< Info * > &info_seq)=0

  Interface redefinition.
- virtual void **take** (std::vector< Data * > &sample_seq, std::vector< Info * > &info_seq, const **SelectorState** &selector_state)=0

  Interface redefinition.
- virtual void **read** (std::vector< Data * > &sample_seq, std::vector< Info * > &info_seq, const **SelectorState** &selector_state)=0

  Interface redefinition.
- virtual void **return_loan** (std::vector< Data * > &sample_seq, std::vector< Info * > &info_seq)=0

  Interface redefinition.
- virtual void **take** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  Takes a collection of all data samples and info samples available from an input stream.

- virtual void **take** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq, const **Selector↩ State** &selector_state)=0

  *Variation of **StreamReader::take** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).*
- virtual void **read** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  *Variation of **StreamReader::take** (p. 104) where the returned samples will remain in the **StreamReader** (p. 102)'s cache, so they can be read again by subsequence **StreamReader::take** (p. 104) or **StreamReader::read** (p. 104) operations.*
- virtual void **read** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq, const **Selector↩ State** &selector_state)=0

  *Variation of **StreamReader::read** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).*
- virtual void **return_loan** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  Returns a loan on the read or taken data samples and info samples.

8.26.1 Detailed Description

template<typename Data, typename Info>
class rti::routing::adapter::TStreamReader< Data, Info >

A wrapper implementation of a **StreamReader** (p. 102) that provides a strongly-typed interface through template parameters for data and info representation.

You can implement this interface as a convenience to manipulate the data and info representation without dealing with opaque pointers.

8.26.2 Member Typedef Documentation

8.26.2.1 DataRep

```
template<typename Data , typename Info >
typedef Data  rti::routing::adapter::TStreamReader< Data, Info >::DataRep
```

The data type.

8.26.2.2 InfoRep

```
template<typename Data , typename Info >
typedef Info  rti::routing::adapter::TStreamReader< Data, Info >::InfoRep
```

The info type.

8.26.3 Member Function Documentation

8.26.3.1 take() [1/6]

```
template<typename Data , typename Info >
void  rti::routing::adapter::TStreamReader< Data, Info >::take (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::adapter::StreamReader** (p. 104).

References **rti::routing::adapter::TStreamReader< Data, Info >::take()**.

Referenced by **rti::routing::adapter::TStreamReader< Data, Info >::take()**.

8.26.3.2 read() [1/6]

```
template<typename Data , typename Info >
void  rti::routing::adapter::TStreamReader< Data, Info >::read (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::adapter::StreamReader** (p. 104).

References **rti::routing::adapter::TStreamReader< Data, Info >::read()**.

Referenced by **rti::routing::adapter::TStreamReader< Data, Info >::read()**.

8.26.3.3 take() [2/6]

```
template<typename Data , typename Info >
void  rti::routing::adapter::TStreamReader< Data, Info >::take (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq,
            const  SelectorState & selector_state )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::adapter::StreamReader** (p. 105).

References **rti::routing::adapter::TStreamReader< Data, Info >::take()**.

8.26.3.4 read() [2/6]

```
template<typename Data , typename Info >
void  rti::routing::adapter::TStreamReader< Data, Info >::read (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq,
            const  SelectorState & selector_state )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::adapter::StreamReader** (p. 105).

References **rti::routing::adapter::TStreamReader< Data, Info >::read()**.

8.26.3.5 return_loan() [1/3]

```
template<typename Data , typename Info >
void  rti::routing::adapter::TStreamReader< Data, Info >::return_loan (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::adapter::StreamReader** (p. 107).

References **rti::routing::adapter::TStreamReader< Data, Info >::return_loan()**.

Referenced by **rti::routing::adapter::TStreamReader< Data, Info >::return_loan()**.

8.26.3.6 take() [3/6]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::TStreamReader< Data, Info >::take (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq )  [pure virtual]
```

Interface redefinition.

See also

StreamReader::take (p. 104)

Implemented in **rti::routing::adapter::NoOpStreamReader< Data, Info >** (p. 52).

8.26.3.7 read() [3/6]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::TStreamReader< Data, Info >::read (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq )  [pure virtual]
```

Interface redefinition.

See also

StreamReader::read (p. 104)

Implemented in **rti::routing::adapter::NoOpStreamReader< Data, Info >** (p. 52).

8.26.3.8 take() [4/6]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::TStreamReader< Data, Info >::take (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq,
            const  SelectorState & selector_state )  [pure virtual]
```

Interface redefinition.

See also

StreamReader::take(std::vector&, std::vector&,const SelectorState&);

Implemented in **rti::routing::adapter::NoOpStreamReader< Data, Info >** (p. 52).

8.26.3.9 read() [4/6]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::TStreamReader< Data, Info >::read (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq,
            const  SelectorState & selector_state )  [pure virtual]
```

Interface redefinition.

See also

StreamReader::read(std::vector&, std::vector&,const SelectorState&);

Implemented in **rti::routing::adapter::NoOpStreamReader< Data, Info >** (p. 53).

8.26.3.10 return_loan() [2/3]

```
template<typename Data , typename Info >
virtual void  rti::routing::adapter::TStreamReader< Data, Info >::return_loan (
             std::vector< Data * > & sample_seq,
             std::vector< Info * > & info_seq )  [pure virtual]
```

Interface redefinition.

See also

StreamReader::return_loan (p. 107)

Implemented in **rti::routing::adapter::NoOpStreamReader< Data, Info >** (p. 53).

8.26.3.11 take() [5/6]

```
template<typename Data , typename Info >
virtual void rti::routing::adapter::StreamReader::take (
             std::vector< SamplePtr > & sample_seq,
             std::vector< InfoPtr > & info_seq )  [virtual]
```

Takes a collection of all data samples and info samples available from an input stream.

When RTI Routing **Service** (p. 83) is done using the samples, it will 'return the loan' to the **StreamReader** (p. 102) by calling **StreamReader::return_loan** (p. 107). Note that multiple calls to this operation can be made before returning the loans.

The samples provided with this operation are considered 'removed' from the **StreamReader** (p. 102) cache, and they shall no longer be returned by subsequent **StreamReader::take** (p. 104) or **StreamReader::read** (p. 104) operations.

This operation represents the minimum required behavior by RTI Routing **Service** (p. 83) in order to forward data.

Parameters

sample_seq	<<*inout*>> Vector that will hold the output samples. RTI Routing **Service** (p. 83) provides this vector to the **StreamReader** (p. 102) to fill. For each call, the size of the vector is zero. The data representation associated with the samples will be given by the value of TypeInfo::data_representation_kind that is part of the **StreamInfo** (p. 99) object passed at **StreamWriter** (p. 109) creation time. Usually the data representation is dynamic type, which corresponds to dds::core::xtypes::DynamicData.
info_seq	<<*inout*>> Vector that will hold the output info samples. RTI Routing **Service** (p. 83) provides this vector to the **StreamReader** (p. 102) to fill. For each call, the size of the vector is zero. The implementation may leave the vector untouched if it does not support the concept of sample info. The info representation is dependent of the **StreamWriter** (p. 109) implementation. Usually when data representation is dynamic type, the sample info is dds::sub::SampleInfo.

Exceptions

std::exception	

Implements **rti::routing::adapter::StreamReader** (p. 104).

8.26.3.12 take() [6/6]

```
template<typename Data , typename Info >
virtual void rti::routing::adapter::StreamReader::take (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq,
            const  SelectorState & selector_state )  [virtual]
```

Variation of **StreamReader::take** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).

SelectorState (p. 83) is configured accordingly from the values set on **rti::routing::processor::Selector** (p. 77). This operation will be called only by custom **rti::routing::processor::Processor** (p. 56) implementations.

Parameters

sample_seq	
info_seq	
selector_state	<<*in*>> A description of the subset of samples that shall be returned.

See also

SelectorState (p. 83).

Implements **rti::routing::adapter::StreamReader** (p. 105).

8.26.3.13 read() [5/6]

```
template<typename Data , typename Info >
virtual void rti::routing::adapter::StreamReader::read (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [virtual]
```

Variation of **StreamReader::take** (p. 104) where the returned samples will remain in the **StreamReader** (p. 102)'s cache, so they can be read again by subsequence **StreamReader::take** (p. 104) or **StreamReader::read** (p. 104) operations.

See also

StreamReader::take (p. 104)

Implements **rti::routing::adapter::StreamReader** (p. 104).

8.26.3.14 read() [6/6]

```
template<typename Data , typename Info >
virtual void rti::routing::adapter::StreamReader::read (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq,
            const  SelectorState & selector_state )  [virtual]
```

Variation of **StreamReader::read** (p. 104) where the returned samples shall represent the subset specified by the **SelectorState** (p. 83).

SelectorState (p. 83) is configured accordingly from the values set on **rti::routing::processor::Selector** (p. 77). This operation will be called only by custom **rti::routing::processor::Processor** (p. 56) implementations.

Parameters

sample_seq	
info_seq	
selector_state	<<*in*>> A description of the subset of samples that shall be returned.

See also

SelectorState (p. 83).

Implements **rti::routing::adapter::StreamReader** (p. 105).

8.26.3.15 return_loan() [3/3]

```
template<typename Data , typename Info >
virtual void rti::routing::adapter::StreamReader::return_loan (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [virtual]
```

Returns a loan on the read or taken data samples and info samples.

RTI Routing **Service** (p. 83) calls this method to indicate that it is done accessing the collection of data samples and info samples obtained by an earlier invocation of any variation of **StreamReader::take** (p. 104).

Parameters

sample_seq	<<*in*>> Vector of loaned data samples.
info_seq	<<*in*>> Vector of loaned info samples.

Exceptions

std::exception	

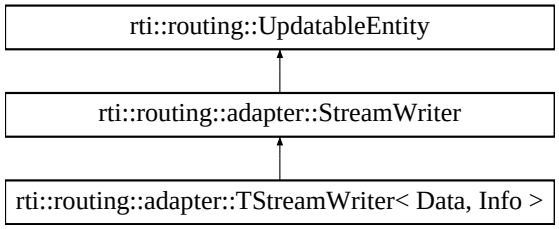
Implements **rti::routing::adapter::StreamReader** (p. 107).

8.27 rti::routing::adapter::TStreamWriter< Data, Info > Class Template Reference

A wrapper implementation of a **StreamWriter** (p. 109) that provides a strongly-typed interface through template parameters for data and info representation.

`#include <StreamWriter.hpp>`

Inheritance diagram for rti::routing::adapter::TStreamWriter< Data, Info >:



Public Types

- typedef Data **DataRep**
  The data type.
- typedef Info **InfoRep**
  The info type.

Public Member Functions

- int **write** (const std::vector< SamplePtr > &sample_seq, const std::vector< InfoPtr > &info_seq) RTI_FINAL
  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- virtual int **write** (const std::vector< Data * > &sample_seq, const std::vector< Info * > &info_seq)=0
  Interface redefinition.
- virtual ~**TStreamWriter** ()
  Virtual destructor.

8.27.1 Detailed Description

template<typename Data, typename Info>
class rti::routing::adapter::TStreamWriter< Data, Info >

A wrapper implementation of a **StreamWriter** (p. 109) that provides a strongly-typed interface through template parameters for data and info representation.

You can implement this interface as a convenience to manipulate the data and info representation without dealing with opaque pointers.

8.27.2 Member Typedef Documentation

8.27.2.1 DataRep

```
template<typename Data , typename Info >
typedef Data  rti::routing::adapter::TStreamWriter< Data, Info >::DataRep
```

The data type.

8.27.2.2 InfoRep

```
template<typename Data , typename Info >
typedef Info  rti::routing::adapter::TStreamWriter< Data, Info >::InfoRep
```

The info type.

8.27.3 Constructor & Destructor Documentation

8.27.3.1 ~TStreamWriter()

```
template<typename Data , typename Info >
virtual  rti::routing::adapter::TStreamWriter< Data, Info >::~ TStreamWriter ( )  [inline], [virtual]
```

Virtual destructor.

8.27.4 Member Function Documentation

8.27.4.1 write() [1/2]

```
template<typename Data , typename Info >
int  rti::routing::adapter::TStreamWriter< Data, Info >::write (
            const std::vector< SamplePtr > & sample_seq,
            const std::vector< InfoPtr > & info_seq )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::adapter::StreamWriter** (p. 110).

References **rti::routing::adapter::TStreamWriter< Data, Info >::write()**.

Referenced by **rti::routing::adapter::TStreamWriter< Data, Info >::write()**.

8.27.4.2 write() [2/2]

```
template<typename Data , typename Info >
virtual int  rti::routing::adapter::TStreamWriter< Data, Info >::write (
             const std::vector< Data * > & sample_seq,
             const std::vector< Info * > & info_seq )  [pure virtual]
```

Interface redefinition.

See also

StreamWriter::write (p. 110)

8.28 rti::routing::processor::TypedInput< Data, Info > Class Template Reference

Representation of an **Input** (p. 39) whose data representation is `DataRep`, whose info representation is `InfoRep`.

```
#include <Input.hpp>
```

Public Member Functions

- const **rti::routing::StreamInfo** & **stream_info** () const

  *Returns the **StreamInfo** (p. 99) associated with this object.*
- const std::string & **name** () const

  Returns the name of this output.
- **LoanedSamples**< Data, Info > **take** ()

  Returns all the available samples in this object.
- **LoanedSamples**< Data, Info > **read** ()

  *Same as **take()** (p. 127) but this calls **rti::routing::adapter::StreamReader::read** (p. 104) instead.*
- **Selector select** ()

  *Gets a **Selector** (p. 77) to perform complex data selections, such as per-instance selection, content and status filtering.*
- bool **active** ()

  Indicates whether this input has received new data since the last time any variant of the read operations was called.
- Data **create_data** ()

  Creates a new data sample from this input.
- dds::sub::DataReader< Data > **dds_data_reader** ()

  Returns the underlying DDS DataReader that is part of this StreamReader implementation, assuming this input holds a DDS StreamReader.

Friends

- class **rti::routing::processor::Selector< Data, Info >**
- class **rti::routing::processor::Route**

8.28.1 Detailed Description

template<typename Data, typename Info>
class rti::routing::processor::TypedInput< Data, Info >

Representation of an **Input** (p. 39) whose data representation is `DataRep`, whose info representation is `InfoRep`.

8.28.2 Member Function Documentation

8.28.2.1 stream_info()

```
template<typename Data , typename Info >
const  rti::routing::StreamInfo &  rti::routing::processor::TypedInput< Data, Info >::stream_info
```

Returns the **StreamInfo** (p. 99) associated with this object.

The associated **StreamInfo** (p. 99) represents an equivalent object to the one used to create the underlying **rti::routing::adapter::StreamReader** (p. 102).

8.28.2.2 name()

```
template<typename Data , typename Info >
const std::string &  rti::routing::processor::TypedInput< Data, Info >::name
```

Returns the name of this output.

The name is given by the name of the corresponding XML configuration tag.

8.28.2.3 take()

```
template<typename Data , typename Info >
LoanedSamples< Data, Info >  rti::routing::processor::TypedInput< Data, Info >::take
```

Returns all the available samples in this object.

This operation will call **rti::routing::adapter::StreamReader::take** (p. 104) on the underlying **rti::routing::adapter::StreamReader** (p. 102).

The returned samples are interpreted to contain data represented as `Data` and info represented as `Info`.

See also

LoanedSamples (p. 40)

rti::routing::adapter::StreamReader::take (p. 104)

8.28.2.4 read()

```
template<typename Data , typename Info >
LoanedSamples< Data, Info >  rti::routing::processor::TypedInput< Data, Info >::read
```

Same as **take()** (p. 127) but this calls **rti::routing::adapter::StreamReader::read** (p. 104) instead.

See also

LoanedSamples (p. 40)

rti::routing::adapter::StreamReader::read (p. 104)

take (p. 127)

8.28.2.5 select()

```
template<typename Data , typename Info >
rti::routing::processor::Selector< Data, Info >  rti::routing::processor::TypedInput< Data, Info
>::select
```

Gets a **Selector** (p. 77) to perform complex data selections, such as per-instance selection, content and status filtering.

The selector can be used as follows:

```
 {.c++}
LoanedSamples<DynamicData> samples = input.select()
    .instance(handle)
    .content(query)
    .state(state)
    .take();
```

This shows how samples can be taken by selecting a specific instance, then filtering by state and content.

Note that when the application wants to access all available samples, it can simply call **read()** (p. 127) or **take()** (p. 127).

Returns

A **Selector** (p. 77), typically used in-line to configure it and finally call **Selector::read()** (p. 82) or **Selector::take()** (p. 82).

See also

Selector (p. 77) for the different parameters that can be set to configure which samples to **read** (p. 127) or **take** (p. 127). Selecting what samples to **read** (p. 127).

8.28.2.6 active()

```
template<typename Data , typename Info >
bool  rti::routing::processor::TypedInput< Data, Info >::active
```

Indicates whether this input has received new data since the last time any variant of the read operations was called.

Note

Given the time decoupling between the notification of new data and when it the data is read, it is possible this function may return true and yet there is no data to be returned.

Returns

True if this input has new data received since the last read access.

8.28.2.7 create_data()

```
template<typename Data , typename Info >
Data  rti::routing::processor::TypedInput< Data, Info >::create_data
```

Creates a new data sample from this input.

The data constructor called in this operation depends on the data representation kind of the **TypeInfo** (p. 137) associated with this input. If the type representation is a dynamic type, then it will assume the following constructor:
`Data(const dds::core::xtypes::DynamicType&)`

and will use the type code object downcasted from the value returned by **TypeInfo::type_representation** (p. 140).

Otherwise:
`Data()`

Returns

A new data sample

8.28.2.8 dds_data_reader()

```
template<typename Data , typename Info >
dds::sub::DataReader< Data >  rti::routing::processor::TypedInput< Data, Info >::dds_data_reader
( )  [inline]
```

Returns the underlying DDS DataReader that is part of this StreamReader implementation, assuming this input holds a DDS StreamReader.

This operation will throw an exception if this input does not hold a DDS StreamReader.

Note

The access to the underlying DataReader is allowed only as long as this **Input** (p. 39) remains enabled. The object representing the DataReader must be invalidated before the input is disabled, otherwise the behavior is undefined.

Exceptions

std::exception	

Returns

dds::sub::DataReader<Data>

8.29 rti::routing::processor::TypedOutput< Data, Info > Class Template Reference

Representation of an **Output** (p. 55) whose data representation is `DataRep`, whose info representation is `InfoRep`.

`#include <Output.hpp>`

Public Member Functions

- const **rti::routing::StreamInfo** & **stream_info** () const

  *Returns the **StreamInfo** (p. 99) associated with this object.*
- const std::string & **name** () const

  Returns the name of this output.
- void **write** (const Data &sample, const Info &info)

  Writes the specified data and info sample in this output.
- Data **create_data** ()

  Creates a new data sample that can be used to write in this output.
- dds::pub::DataWriter< Data > **dds_data_writer** ()

  Returns the underlying DDS DataWriter that is part of this StreamWriter implementation, assuming this input holds a DDS StreamWriter.

Friends

- class **rti::routing::processor::Route**

8.29.1 Detailed Description

template<typename Data, typename Info>
class rti::routing::processor::TypedOutput< Data, Info >

Representation of an **Output** (p. 55) whose data representation is `DataRep`, whose info representation is `InfoRep`.

8.29.2 Member Function Documentation

8.29.2.1 stream_info()

```
template<typename Data , typename Info >
const  rti::routing::StreamInfo &  rti::routing::processor::TypedOutput< Data, Info >::stream_↩
info
```

Returns the **StreamInfo** (p. 99) associated with this object.

The associated **StreamInfo** (p. 99) represents an equivalent object to the one used to create the underlying **rti↩ ::routing::adapter::StreamWriter** (p. 109).

8.29.2.2 name()

```
template<typename Data , typename Info >
const std::string &  rti::routing::processor::TypedOutput< Data, Info >::name
```

Returns the name of this output.

The name is given by the name of the corresponding XML configuration tag.

8.29.2.3 write()

```
template<typename Data , typename Info >
void  rti::routing::processor::TypedOutput< Data, Info >::write (
            const Data & sample,
            const Info & info )
```

Writes the specified data and info sample in this output.

This operation will call **rti::routing::adapter::StreamWriter::write** (p. 110) on the underlying **rti::routing::adapter::↩ StreamWriter** (p. 109).

The provided sample is interpreted to contain data represented as `Data` and the info as `Info`.

See also

rti::routing::adapter::StreamWriter::write (p. 110)

8.29.2.4 create_data()

```
template<typename Data , typename Info >
Data  rti::routing::processor::TypedOutput< Data, Info >::create_data
```

Creates a new data sample that can be used to write in this output.

The data constructor called in this operation depends on the data representation kind of the **TypeInfo** (p. 137) associated with this output. If the type representation is a dynamic type, then it will assume the following constructor:
`Data(const dds::core::xtypes::DynamicType&)`

and will use the type code object downcasted from the value returned by **TypeInfo::type_representation** (p. 140).

Otherwise:
`Data()`

Returns

A new data sample

8.29.2.5 dds_data_writer()

```
template<typename Data , typename Info >
dds::pub::DataWriter< Data >  rti::routing::processor::TypedOutput< Data, Info >::dds_data_↩
writer ( )  [inline]
```

Returns the underlying DDS DataWriter that is part of this StreamWriter implementation, assuming this input holds a DDS StreamWriter.

This operation will throw an exception if this output does not hold a DDS StreamWriter.

Note

The access to the underlying DataWriter is allowed only as long as this **Output** (p. 55) remains enabled. The object representing the DataWriter must be invalidated before the output is disabled, otherwise the behavior is undefined.

Exceptions

std::exception	

Returns

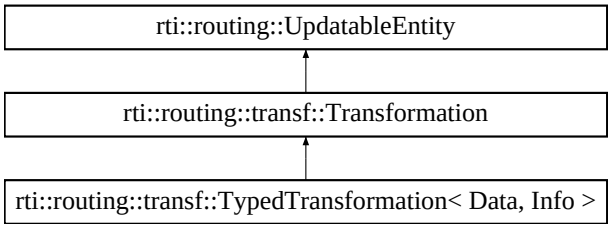
dds::pub::DataWriter<Data>

8.30 rti::routing::transf::TypedTransformation< Data, Info > Class Template Reference

A wrapper implementation of a **Transformation** (p. 111) that provides a strongly-typed interface through template parameters for data and info representation.

`#include <Transformation.hpp>`

Inheritance diagram for rti::routing::transf::TypedTransformation< Data, Info >:



Public Types

- typedef Data **DataRep**

  The data type.
- typedef Info **InfoRep**

  The info type.

Public Member Functions

- void **transform** (std::vector< SamplePtr > &output_sample_seq, std::vector< InfoPtr > &output_info_seq, const std::vector< SamplePtr > &input_sample_seq, const std::vector< InfoPtr > &input_info_seq) RTI_FINAL

  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- void **return_loan** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq) RTI_FINAL

  Performs the conversion between the vector of data and info pointers to strongly-type pointers.
- virtual void **transform** (std::vector< Data * > &output_sample_seq, std::vector< Info * > &output_info_seq, const std::vector< Data * > &input_sample_seq, const std::vector< Info * > &input_info_seq)=0

  Interface redefinition.
- virtual void **return_loan** (std::vector< Data * > &sample_seq, std::vector< Info * > &info_seq)=0

  Interface redefinition.
- virtual ~**TypedTransformation** ()

  Virtual destructor.
- virtual void **transform** (std::vector< SamplePtr > &output_sample_seq, std::vector< InfoPtr > &output_info_↩ seq, const std::vector< SamplePtr > &input_sample_seq, const std::vector< InfoPtr > &input_info_seq)=0

  Transforms a vector of input samples into a vector of output samples.
- virtual void **return_loan** (std::vector< SamplePtr > &sample_seq, std::vector< InfoPtr > &info_seq)=0

  Returns a loan on the transformed data samples and info samples.

8.30.1 Detailed Description

template<typename Data, typename Info>
class rti::routing::transf::TypedTransformation< Data, Info >

A wrapper implementation of a **Transformation** (p. 111) that provides a strongly-typed interface through template parameters for data and info representation.

You can implement this interface as a convenience to manipulate the data and info representation without dealing with opaque pointers.

8.30.2 Member Typedef Documentation

8.30.2.1 DataRep

```
template<typename Data , typename Info >
typedef Data  rti::routing::transf::TypedTransformation< Data, Info >::DataRep
```

The data type.

8.30.2.2 InfoRep

```
template<typename Data , typename Info >
typedef Info  rti::routing::transf::TypedTransformation< Data, Info >::InfoRep
```

The info type.

8.30.3 Constructor & Destructor Documentation

8.30.3.1 ~TypedTransformation()

```
template<typename Data , typename Info >
virtual  rti::routing::transf::TypedTransformation< Data, Info >::~ TypedTransformation ( )  [inline],
[virtual]
```

Virtual destructor.

8.30.4 Member Function Documentation

8.30.4.1 transform() [1/3]

```
template<typename Data , typename Info >
void  rti::routing::transf::TypedTransformation< Data, Info >::transform (
            std::vector< SamplePtr > & output_sample_seq,
            std::vector< InfoPtr > & output_info_seq,
            const std::vector< SamplePtr > & input_sample_seq,
            const std::vector< InfoPtr > & input_info_seq )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::transf::Transformation** (p. 111).

References **rti::routing::transf::TypedTransformation< Data, Info >::transform()**.

Referenced by **rti::routing::transf::TypedTransformation< Data, Info >::transform()**.

8.30.4.2 return_loan() [1/3]

```
template<typename Data , typename Info >
void  rti::routing::transf::TypedTransformation< Data, Info >::return_loan (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [inline], [virtual]
```

Performs the conversion between the vector of data and info pointers to strongly-type pointers.

Implements **rti::routing::transf::Transformation** (p. 112).

References **rti::routing::transf::TypedTransformation< Data, Info >::return_loan()**.

Referenced by **rti::routing::transf::TypedTransformation< Data, Info >::return_loan()**.

8.30.4.3 transform() [2/3]

```
template<typename Data , typename Info >
virtual void  rti::routing::transf::TypedTransformation< Data, Info >::transform (
            std::vector< Data * > & output_sample_seq,
            std::vector< Info * > & output_info_seq,
            const std::vector< Data * > & input_sample_seq,
            const std::vector< Info * > & input_info_seq )  [pure virtual]
```

Interface redefinition.

See also

Transformation::transform (p. 111)

8.30.4.4 return_loan() [2/3]

```
template<typename Data , typename Info >
virtual void  rti::routing::transf::TypedTransformation< Data, Info >::return_loan (
            std::vector< Data * > & sample_seq,
            std::vector< Info * > & info_seq )  [pure virtual]
```

Interface redefinition.

See also

Transformation::return_loan (p. 112)

8.30.4.5 transform() [3/3]

```
template<typename Data , typename Info >
virtual void rti::routing::transf::Transformation::transform (
            std::vector< SamplePtr > & output_sample_seq,
            std::vector< InfoPtr > & output_info_seq,
            const std::vector< SamplePtr > & input_sample_seq,
            const std::vector< InfoPtr > & input_info_seq )  [virtual]
```

Transforms a vector of input samples into a vector of output samples.

When RTI Routing **Service** (p. 83) is done using the output samples, it will 'return the loan' to the transformation by calling **Transformation::return_loan** (p. 112). It is guaranteed RTI Routing **Service** (p. 83) takes one outstanding loan at time.

The number of output samples can be different than the number of input samples.

The format associated with the input and output data samples and info samples depends on the format provided and consumed by the input's **rti::routing::adapter::StreamReader** (p. 102) and the output's **rti::routing::adapter::StreamWriter** (p. 109).

For the built-in DDS adapter, the format of the samples is dds::core::xtypes::DynamicData and the format of the sample info is dds::sub::SampleInfo.

Parameters

output_sample_seq	<<*inout*>> Vector that will hold the output data samples. RTI Routing **Service** (p. 83) provides this vector to the **Transformation** (p. 111) to fill. For each call, the size of the vector is zero.
output_info_seq	<<*inout*>> Vector that will hold the input info samples. RTI Routing **Service** (p. 83) provides this vector to the **Transformation** (p. 111) to fill. For each call, the size of the vector is zero.
input_sample_seq	<<*in*>> Vector of input data samples. This vector contains the list of samples that need to be transformed before calling **rti::routing::adapter::StreamWriter::write** (p. 110).
input_info_seq	<<*in*>> Vector of input info sample. This vector contains the list of samples that need to be transformed before calling **rti::routing::adapter::StreamWriter::write** (p. 110).

Exceptions

std::exception	

Implements **rti::routing::transf::Transformation** (p. 111).

8.30.4.6 return_loan() [3/3]

```
template<typename Data , typename Info >
virtual void rti::routing::transf::Transformation::return_loan (
            std::vector< SamplePtr > & sample_seq,
            std::vector< InfoPtr > & info_seq )  [virtual]
```

Returns a loan on the transformed data samples and info samples.

RTI Routing **Service** (p. 83) calls this method to indicate that it is done accessing the collection of data samples and info samples obtained by an earlier invocation of **Transformation::transform** (p. 111).

Parameters

sample_seq	<<*in*>> Vector of loaned data samples.
info_seq	<<*in*>> Vector of loaned info samples.

Exceptions

std::exception	

Implements **rti::routing::transf::Transformation** (p. 112).

8.31 rti::routing::TypeInfo Class Reference

Definition of the type information associated with a RTI Routing **Service** (p. 83) stream.

```
#include <TypeInfo.hpp>
```

Inherits rti::core::NativeValueType< TypeInfo >.

Public Types

- typedef RTI_RoutingServiceTypeRepresentation **TypeRepresentationType**

  Type representation defined as an opaque pointer.

Public Member Functions

- **TypeInfo** (const std::string &the_type_name, **TypeRepresentationKind** the_type_representation_kind=TypeRepresentationKind::DYNAMIC_TYPE)

  *Constructs a **TypeInfo** (p. 137) with the minimum required information.*
- std::string **type_name** () const

  Getter (see setter with the same name)
- **TypeInfo** & **type_name** (const std::string &the_type_name)

  Sets the registered name of the type associated with the data.
- **TypeRepresentationKind** **type_representation_kind** () const

  Getter (see setter with the same name)
- **TypeInfo** & **type_representation_kind** (**TypeRepresentationKind** the_type_representation_kind)

  Sets the type representation kind.
- **TypeRepresentationType** **type_representation** () const

  Getter (see setter with the same name)
- **TypeInfo** & **type_representation** (**TypeRepresentationType** the_type_representation)

  Sets the type representation value.
- const dds::core::xtypes::DynamicType & **dynamic_type** () const

  Returns the type_representation as DynamicType object.
- void **dynamic_type** (const dds::core::xtypes::DynamicType *dynamic_type)

  Sets the type representation value as DynamicType object.

8.31.1 Detailed Description

Definition of the type information associated with a RTI Routing **Service** (p. 83) stream.

See also

StreamInfo (p. 99).

8.31.2 Member Typedef Documentation

8.31.2.1 TypeRepresentationType

`typedef RTI_RoutingServiceTypeRepresentation` **`rti::routing::TypeInfo::TypeRepresentationType`**

Type representation defined as an opaque pointer.

How this opaque is interpreted depends on the **TypeRepresentationKind** (p. 141).

8.31.3 Constructor & Destructor Documentation

8.31.3.1 TypeInfo()

```
rti::routing::TypeInfo::TypeInfo (
            const std::string & the_type_name,
             TypeRepresentationKind the_type_representation_kind = TypeRepresentationKind:↩
:DYNAMIC_TYPE )  [inline]
```

Constructs a **TypeInfo** (p. 137) with the minimum required information.

Parameters

the_type_name	<<*in*>> Registered name of the type associated with the data.
the_type_representation_kind	<<*in*>> Indicates the representation of the object that describes the type. Default: TypeRepresentationKind::DYNAMIC_TYPE.

References **type_name()**, and **type_representation_kind()**.

8.31.4 Member Function Documentation

8.31.4.1 type_name() [1/2]

```
std::string rti::routing::TypeInfo::type_name ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **rti::routing::StreamInfo::StreamInfo()**, **type_name()**, and **TypeInfo()**.

8.31.4.2 type_name() [2/2]

```
TypeInfo & rti::routing::TypeInfo::type_name (
            const std::string & the_type_name )  [inline]
```

Sets the registered name of the type associated with the data.

RTI Routing **Service** (p. 83) uses the type name to provide the right type information to StreamReader and StreamWriter. Namely, a type match is performed by comparing this type name with the value specified in the tag <registered_type↩
_name> within the <input> and <output> tags.

References **type_name()**.

8.31.4.3 type_representation_kind() [1/2]

```
TypeRepresentationKind rti::routing::TypeInfo::type_representation_kind ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **dynamic_type()**, **type_representation_kind()**, and **TypeInfo()**.

8.31.4.4 type_representation_kind() [2/2]

```
TypeInfo & rti::routing::TypeInfo::type_representation_kind (
            TypeRepresentationKind the_type_representation_kind )  [inline]
```

Sets the type representation kind.

This member indicates how to interpret the type representation.

See also

TypeRepresentationKind (p. 141)

References **type_representation_kind()**.

8.31.4.5 type_representation() [1/2]

```
TypeRepresentationType rti::routing::TypeInfo::type_representation ( ) const  [inline]
```

Getter (see setter with the same name)

Referenced by **dynamic_type()**, and **type_representation()**.

8.31.4.6 type_representation() [2/2]

```
TypeInfo & rti::routing::TypeInfo::type_representation (
            TypeRepresentationType the_type_representation )  [inline]
```

Sets the type representation value.

This member is defined as an opaque pointer that can hold any type representation. How to interpret this pointer is determined by the **TypeRepresentationKind** (p. 141), which shall be consistent with the object set in this member.

References **type_representation()**.

8.31.4.7 dynamic_type() [1/2]

```
const dds::core::xtypes::DynamicType & rti::routing::TypeInfo::dynamic_type ( ) const  [inline]
```

Returns the type_representation as DynamicType object.

The operation throws if type_representation_kind is not DYNAMIC_TYPE

References **type_representation()**, and **type_representation_kind()**.

Referenced by **dynamic_type()**.

8.31.4.8 dynamic_type() [2/2]

```
void rti::routing::TypeInfo::dynamic_type (
            const dds::core::xtypes::DynamicType * dynamic_type )  [inline]
```

Sets the type representation value as DynamicType object.

The provided DynamicType pointer must point to an object that remains valid until the **StreamInfo** (p. 99) object is destroyed.

See also

type_representation(TypeRepresentationType the_type_representation) (p. 140)

References **dynamic_type()**, **type_representation()**, and **type_representation_kind()**.

8.32 TypeRepresentationKind Class Reference

valid type representations that RTI Routing Service accepts.

```
#include <TypeInfo.hpp>
```

8.32.1 Detailed Description

valid type representations that RTI Routing Service accepts.

See also

rti::routing::TypeRepresentationKind_def (p. 142)

8.33 rti::routing::TypeRepresentationKind_def Class Reference

The definition of the rti::routing::TypeRepresentationKind.

`#include <TypeInfo.hpp>`

Public Types

- enum **type** {
  DYNAMIC_TYPE = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_DYNAMIC_TYPE ,
  XML = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_XML ,
  JAVA_OBJECT = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_JAVA_OBJECT ,
  CUSTOM = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_FIRST_CUSTOM_REPRESENTATION }

  Definition of all the possible type representation kinds.

8.33.1 Detailed Description

The definition of the rti::routing::TypeRepresentationKind.

The range [0-100] is reserved for RTI use. Within that range are some predefined type representations.

8.33.2 Member Enumeration Documentation

8.33.2.1 type

`enum` **`rti::routing::TypeRepresentationKind_def::type`**

Definition of all the possible type representation kinds.

Enumerator

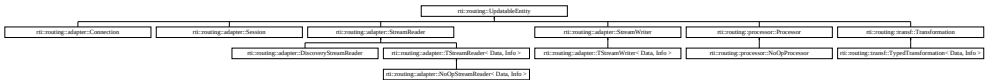
DYNAMIC_TYPE	Dynamic type representation. Types with this representation are provided as RTI Connext TypeCodes. For more information about TypeCodes, see Chapter 3 in the RTI Core Libraries and Utilities User's Manual.
XML	**[Not supported]** XML type representation. Types with this representation are provided as XML strings. The XML schema is the RTI Connext XML schema for type representation. For more information about XML representation, see Chapter 3 in the RTI Core Libraries and Utilities User's Manual.
JAVA_OBJECT	**[Not supported]** Java object type representation. Types with this representation are provided as Java objects.
CUSTOM	**[Not supported]** Custom type representation. Types with this representation are specific to a concrete adapter implementation

8.34 rti::routing::UpdatableEntity Class Reference

Defines a common interface for all the pluggable entities that can be updated at runtime.

`#include <UpdatableEntity.hpp>`

Inheritance diagram for rti::routing::UpdatableEntity:



Public Member Functions

- virtual void **update** (const std::map< std::string, std::string > &properties)

  Updates a pluggable entity.
- virtual ~**UpdatableEntity** ()

  Virtual destructor.

8.34.1 Detailed Description

Defines a common interface for all the pluggable entities that can be updated at runtime.

RTI Routing **Service** (p. 83) allows to update an pluggable entity through remote administration.

Multi-threading safety Safe The **UpdatableEntity::update** (p. 143) operation is called only for one entity at a time. Additionally, no other operations can be made concurrently on a pluggable entity during the **UpdatableEntity::update** (p. 143) call.

8.34.2 Constructor & Destructor Documentation

8.34.2.1 ~UpdatableEntity()

```
virtual rti::routing::UpdatableEntity::~UpdatableEntity ( )  [inline], [virtual]
```

Virtual destructor.

8.34.3 Member Function Documentation

8.34.3.1 update()

```
virtual void rti::routing::UpdatableEntity::update (
            const std::map< std::string, std::string > & properties )  [inline], [virtual]
```

Updates a pluggable entity.

Parameters

properties	<<*in*>> The new configuration properties, obtained as result of parsing the elements within the <property> tag corresponding to the entity being updated.

Chapter 9

File Documentation

9.1 AdapterPlugin.hpp File Reference

RTI Routing Service C++ Adapter API.

```
#include <map>
#include "routingservice/routingservice_infrastructure.h"
#include <rti/config/Version.hpp>
#include <rti/routing/adapter/StreamReaderListener.hpp>
#include <rti/routing/adapter/Connection.hpp>
#include <rti/routing/adapter/detail/AdapterForwarder.hpp>
```

Data Structures

- class **rti::routing::adapter::AdapterPlugin**

  The top-level plug-in class.

Macros

- #define **RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DECL**(ADAPTER_CLASS)

  Utility macro that declares a native extern function that can be used to load an AdapterPlugin through a shared library.
- #define **RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DEF**(ADAPTER_CLASS)

  Utility macro that implements the AdapterPlugin entry point declared with RTI_ADAPTER_PLUGIN_CREATE_↩ FUNCTION_DECL.

9.1.1 Detailed Description

RTI Routing Service C++ Adapter API.

9.2 AdapterPlugin.hpp

Go to the documentation of this file.

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_ADAPTER_HPP_
12 #define RTI_ROUTING_ADAPTER_ADAPTER_HPP_
13
14 #include <map>
15
16 #include "routingservice/routingservice_infrastructure.h"
17 #include <rti/config/Version.hpp>
18 #include <rti/routing/adapter/StreamReaderListener.hpp>
19 #include <rti/routing/adapter/Connection.hpp>
20
25 namespace rti { namespace routing { namespace adapter {
26
34 class AdapterPlugin {
35 public:
59     virtual Connection * create_connection(
60             StreamReaderListener *input_stream_discovery_listener,
61             StreamReaderListener *output_stream_discovery_listener,
62             const PropertySet& properties) = 0;
63
76     virtual void delete_connection(Connection *connection) = 0;
77
87     virtual rti::config::LibraryVersion get_version() const
88     {
89         return rti::config::LibraryVersion();
90     }
91
95     virtual ~AdapterPlugin() {}
96 };
97
98 }}}
99
100 #include <rti/routing/adapter/detail/AdapterForwarder.hpp>
101
102
167 #define RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DECL(ADAPTER_CLASS) \
168 extern "C" RTI_USER_DLL_EXPORT struct RTI_RoutingServiceAdapterPluginExt * \
169     ADAPTER_CLASS ## _create_adapter_plugin(\
170         const struct RTI_RoutingServiceProperties *, \
171         RTI_RoutingServiceEnvironment *); \
172
178 #define RTI_ADAPTER_PLUGIN_CREATE_FUNCTION_DEF(ADAPTER_CLASS) \
179 struct RTI_RoutingServiceAdapterPluginExt * ADAPTER_CLASS ## _create_adapter_plugin( \
180         const struct RTI_RoutingServiceProperties * native_properties, \
181         RTI_RoutingServiceEnvironment *environment) \
182 { \
183     PropertySet properties; \
184     rti::routing::PropertyAdapter::add_properties_from_native(\
185             properties,\
186             native_properties); \
187     try { \
188         return rti::routing::adapter::detail::AdapterForwarder::create_plugin(new
      ADAPTER_CLASS(properties)); \
189     } catch (const std::exception& ex) {\
190         RTI_RoutingServiceEnvironment_set_error(\
191                 environment,\
192                 "%s",\
193                 ex.what());\
194     } catch (...) {}\
195     \
196     return NULL; \
197 }
198
199
200
201
202 #endif // RTI_ROUTING_ADAPTER_ADAPTER_HPP_
```

9.3 Connection.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_CONNECTION_HPP_
12 #define RTI_ROUTING_ADAPTER_CONNECTION_HPP_
13
14 #include <rti/routing/PropertySet.hpp>
15 #include <rti/routing/UpdatableEntity.hpp>
16 #include <rti/routing/adapter/Session.hpp>
17 #include <rti/routing/adapter/StreamWriter.hpp>
18 #include <rti/routing/adapter/StreamReader.hpp>
19 #include <rti/routing/StreamInfo.hpp>
20 #include <rti/routing/adapter/DiscoveryStreamReader.hpp>
21 #include <rti/routing/adapter/StreamReaderListener.hpp>
22
23 namespace rti { namespace routing { namespace adapter {
24
35 class Connection : public UpdatableEntity {
36
37 public:
57     virtual Session * create_session(
58             const PropertySet& properties)
59     {
60         (void) properties;
61
62         return NULL;
63     }
64
77     virtual void delete_session(Session *session)
78     {
79         (void) session;
80     }
81
109     virtual StreamWriter * create_stream_writer(
110             Session *session,
111             const StreamInfo& stream_info,
112             const PropertySet& properties)
113     {
114         (void) session;
115         (void) stream_info;
116         (void) properties;
117
118         return NULL;
119     }
120
135     virtual void delete_stream_writer(
136             StreamWriter *stream_writer)
137     {
138         (void) stream_writer;
139     }
140
172     virtual StreamReader * create_stream_reader(
173             Session *session,
174             const StreamInfo& stream_info,
175             const PropertySet& properties,
176             StreamReaderListener *listener)
177     {
178         (void) session;
179         (void) stream_info;
180         (void) properties;
181         (void) listener;
182
183         return NULL;
184     }
185
200     virtual void delete_stream_reader(
201             StreamReader *stream_reader)
202     {
203         (void) stream_reader;
204     }
205
```

```
224     virtual DiscoveryStreamReader * input_stream_discovery_reader()
225     {
226         return NULL;
227     }
228
247     virtual DiscoveryStreamReader * output_stream_discovery_reader()
248     {
249         return NULL;
250     }
251
255     virtual ~Connection()
256     {
257     }
258
259 };
260
261 }}}
262
263 #endif // RTI_ROUTING_ADAPTER_CONNECTION_HPP_
```

9.4 AdapterForwarder.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_ADAPTER_FORWARDER_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_ADAPTER_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <routingservice/routingservice_adapter_new.h>
17
18 #include <rti/routing/adapter/detail/ConnectionForwarder.hpp>
19
20 namespace rti { namespace routing { namespace adapter { namespace detail {
21
22
23 class AdapterForwarder {
24 public:
25     static RTI_RoutingServiceAdapterPluginExt * create_plugin(
26             AdapterPlugin *adapter_plugin)
27     {
28         RTI_RoutingServiceAdapterPluginExt *native_adapter = NULL;
29         RTIOsapiHeap_allocateStructure(
30                 &native_adapter,
31                 struct RTI_RoutingServiceAdapterPluginExt);
32         rti::core::check_create_entity(
33                 native_adapter,
34                 "RTI_RoutingServiceAdapterPluginExt");
35         RTI_RoutingServiceAdapterPluginExt_initialize(native_adapter);
36
37         // Set adapter version
38         rti::config::LibraryVersion version =  adapter_plugin->get_version();
39         native_adapter->plugin_version.major = version.major_version();
40         native_adapter->plugin_version.minor = version.minor_version();
41         native_adapter->plugin_version.release = version.release_version();
42
43         // Initialize adapter methods
44         native_adapter->adapter_plugin_data =
45                 static_cast<void *>(adapter_plugin);
46         native_adapter->plugin_delete =
47                 AdapterForwarder::delete_plugin;
48         native_adapter->create_connection =
49                 AdapterForwarder::create_connection;
50         native_adapter->delete_connection =
51                 AdapterForwarder::delete_connection;
52
53
54         return native_adapter;
55     }
```

```
56
57     static void delete_plugin(
58         void *native_adapter_data,
59         RTI_RoutingServiceEnvironment *)
60     {
61         RTI_RoutingServiceAdapterPluginExt *native_adapter =
62                 static_cast<RTI_RoutingServiceAdapterPluginExt *>(
63                         native_adapter_data);
64         AdapterPlugin *adapter = static_cast<AdapterPlugin *>(
65                 native_adapter->adapter_plugin_data);
66         // Plug-in is destructor not allowed to throw
67         delete adapter;
68         RTIOsapiHeap_freeStructure(native_adapter);
69     }
70
71     static RTI_RoutingServiceConnectionExt* create_connection(
72             void *native_adapter_data,
73             const char *routing_service_name,
74             const char *routing_service_group_name,
75             const RTI_RoutingServiceStreamReaderListenerExt *native_output_stream_listener,
76             const RTI_RoutingServiceStreamReaderListenerExt *native_input_stream_listener,
77             const RTI_RoutingServiceTypeInfo **registered_types,
78             int registered_type_count,
79             const RTI_RoutingServiceProperties *native_properties,
80             RTI_RoutingServiceEnvironment *environment)
81     {
82         return ConnectionForwarder::create_native(
83                 static_cast<AdapterPlugin *>(native_adapter_data),
84                 routing_service_name,
85                 routing_service_group_name,
86                 native_output_stream_listener,
87                 native_input_stream_listener,
88                 registered_types,
89                 registered_type_count,
90                 native_properties,
91                 environment);
92     }
93
94     static void delete_connection(
95             void *native_adapter_data,
96             RTI_RoutingServiceConnectionExt *native_connection,
97             RTI_RoutingServiceEnvironment *environment)
98     {
99         ConnectionForwarder::delete_native(
100                 static_cast<AdapterPlugin *>(native_adapter_data),
101                 native_connection,
102                 environment);
103      }
104
105 };
106
107
108 }}}}
109
110 #endif // RTI_ROUTING_ADAPTER_DETAIL_ADAPTER_FORWARDER_HPP_
```

9.5 ConnectionForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_CONNECTION_FORWARDER_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_CONNECTION_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <osapi/osapi_heap.h>
17 #include <routingservice/routingservice_adapter_new.h>
18
19 #include <rti/routing/adapter/detail/SessionForwarder.hpp>
```

```
20 #include <rti/routing/adapter/detail/StreamReaderForwarder.hpp>
21 #include <rti/routing/adapter/detail/StreamWriterForwarder.hpp>
22 #include <rti/routing/adapter/detail/DiscoveryStreamReaderForwarder.hpp>
23 #include <rti/routing/detail/UpdatableEntityForwarder.hpp>
24 #include <rti/routing/detail/ForwarderUtils.hpp>
25
26 namespace rti { namespace routing { namespace adapter { namespace detail {
27
28 class ConnectionForwarder {
29 public:
30     static RTI_RoutingServiceConnectionExt * create_native(
31             AdapterPlugin *adapter,
32             const char *,
33             const char *,
34             const RTI_RoutingServiceStreamReaderListenerExt *native_output_stream_listener,
35             const RTI_RoutingServiceStreamReaderListenerExt *native_input_stream_listener,
36             const RTI_RoutingServiceTypeInfo **,
37             int,
38             const RTI_RoutingServiceProperties *native_properties,
39             RTI_RoutingServiceEnvironment *environment)
40     {
41         try {
42             using rti::routing::detail::ScopedForwarder;
43
44             // Set properties
45             std::map<std::string, std::string> properties;
46             rti::routing::PropertyAdapter::add_properties_from_native(
47                     properties,
48                     native_properties);
49
50             ConnectionForwarder *forwarder = new ConnectionForwarder(
51                     native_output_stream_listener,
52                     native_input_stream_listener);
53             ScopedForwarder<AdapterPlugin, ConnectionForwarder> scoped(
54                     adapter,
55                     forwarder,
56                     environment);
57             // Create connection
58             forwarder->connection_ = adapter->create_connection(
59                     &forwarder->input_discovery_listener_,
60                     &forwarder->output_discovery_listener_,
61                     properties);
62             RTI_ROUTING_THROW_ON_NULL(forwarder->connection_);
63
64             // Create built-in SRs
65             forwarder->input_stream_discovery_forwarder_ =
66                     new DiscoveryStreamReaderForwarder(
67                             forwarder->connection_
68                                     ->input_stream_discovery_reader());
69
70             forwarder->output_stream_discovery_forwarder_ =
71                     new DiscoveryStreamReaderForwarder(
72                             forwarder->connection_
73                                     ->output_stream_discovery_reader());
74
75             scoped.release();
76
77             // TRUST-134: the forwarder is not leaked here. It might be leaked
78             // if delete_native() were not to be called, but Routing Service
79             // will make sure that is not the case
80
81             /* coverity[leaked_storage : FALSE] */
82             return forwarder->native();
83         } catch (const std::exception& ex) {
84             RTI_RoutingServiceEnvironment_set_error(
85                     environment,
86                     "%s",
87                     ex.what());
88             return NULL;
89         } catch (...) {
90             RTI_RoutingServiceEnvironment_set_error(
91                     environment,
92                     "unexpected exception");
93             return NULL;
94         }
95     }
96
97     static void delete_native(
98             AdapterPlugin *adapter,
99             RTI_RoutingServiceConnectionExt *native_connection,
100             RTI_RoutingServiceEnvironment *environment)
```

```
101     {
102         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
103                 native_connection->connection_data);
104         try {
105             // delete built-in SRs
106             delete forwarder->input_stream_discovery_forwarder_;
107             delete forwarder->output_stream_discovery_forwarder_;
108 
109             // delete connection
110             if (forwarder->connection_ != NULL) {
111                 adapter->delete_connection(forwarder->connection_);
112                 forwarder->connection_ = NULL;
113             }
114         } catch(const std::exception& ex) {
115             RTI_RoutingServiceEnvironment_set_error(
116                     environment,
117                     "%s",
118                     ex.what());
119         } catch (...) {
120         }
121 
122         delete forwarder;
123     }
124 
125 
126     static RTI_RoutingServiceStreamReaderExt *
127     get_input_stream_discovery_reader(
128             void *native_connection_data,
129             RTI_RoutingServiceEnvironment *)
130     {
131         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
132                 native_connection_data);
133         return forwarder->input_stream_discovery_forwarder_->native();
134     }
135 
136     static RTI_RoutingServiceStreamReaderExt *
137     get_output_stream_discovery_reader(
138             void *native_connection_data,
139             RTI_RoutingServiceEnvironment *)
140     {
141         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
142                 native_connection_data);
143         return forwarder->output_stream_discovery_forwarder_->native();
144     }
145 
146     static RTI_RoutingServiceSessionExt * create_session(
147         void *native_connection_data,
148         const struct RTI_RoutingServiceProperties * native_properties,
149         RTI_RoutingServiceEnvironment *environment)
150     {
151 
152         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
153                 native_connection_data);
154 
155         return SessionForwarder::create_native(
156                 forwarder->connection_,
157                 native_properties,
158                 environment);
159     }
160 
161     static void delete_session(
162             void *native_connection_data,
163             RTI_RoutingServiceSessionExt *native_session,
164             RTI_RoutingServiceEnvironment *environment)
165     {
166         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*> (
167                 native_connection_data);
168 
169         SessionForwarder::delete_native(
170                 forwarder->connection_,
171                 native_session,
172                 environment);
173     }
174 
175     static RTI_RoutingServiceStreamReaderExt* create_stream_reader(
176             void *native_connection_data,
177             void *native_session_data,
178             const struct RTI_RoutingServiceStreamInfo *native_stream_info,
179             const struct RTI_RoutingServiceProperties *native_properties,
180             const struct RTI_RoutingServiceStreamReaderListenerExt *native_listener,
181             RTI_RoutingServiceEnvironment *environment)
```

```
182     {
183         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
184                 native_connection_data);
185         SessionForwarder *session_forwarder = static_cast<SessionForwarder*>(
186                 native_session_data);
187 
188         return StreamReaderForwarder::create_native(
189                 forwarder->connection_,
190                 session_forwarder->session(),
191                 native_stream_info,
192                 native_properties,
193                 native_listener,
194                 environment);
195     }
196 
197     static void delete_stream_reader(
198         void *native_connection_data,
199         RTI_RoutingServiceStreamReaderExt *native_reader,
200         RTI_RoutingServiceEnvironment *environment)
201     {
202         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
203                 native_connection_data);
204         StreamReaderForwarder::delete_native(
205                 forwarder->connection_,
206                 native_reader,
207                 environment);
208     }
209 
210     static RTI_RoutingServiceStreamWriterExt* create_stream_writer(
211             void *native_connection_data,
212             void *native_session_data,
213             const struct RTI_RoutingServiceStreamInfo *native_stream_info,
214             const struct RTI_RoutingServiceProperties *native_properties,
215             const struct RTI_RoutingServiceStreamWriterListenerExt *,
216             RTI_RoutingServiceEnvironment *environment)
217     {
218         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
219                 native_connection_data);
220         SessionForwarder *session_forwarder = static_cast<SessionForwarder*>(
221                 native_session_data);
222 
223         return StreamWriterForwarder::create_native(
224                 forwarder->connection_,
225                 session_forwarder->session(),
226                 native_stream_info,
227                 native_properties,
228                 environment);
229     }
230 
231     static void delete_stream_writer(
232         void *native_connection_data,
233         RTI_RoutingServiceStreamWriterExt *native_writer,
234         RTI_RoutingServiceEnvironment *environment)
235     {
236         ConnectionForwarder *forwarder = static_cast<ConnectionForwarder*>(
237                 native_connection_data);
238         StreamWriterForwarder::delete_native(
239                 forwarder->connection_,
240                 native_writer,
241                 environment);
242     }
243 
244     static void update(
245             void *native_connection_data,
246             const struct RTI_RoutingServiceProperties *native_properties,
247             RTI_RoutingServiceEnvironment *environment)
248     {
249         ConnectionForwarder *forwarder =
250                 static_cast<ConnectionForwarder*> (native_connection_data);
251 
252         rti::routing::detail::UpdatableEntityForwarder::update(
253                 forwarder->connection_,
254                 native_properties,
255                 environment);
256     }
257 
258 
259     Connection * connection()
260     {
261         return connection_;
262     }
```

```
263
264     RTI_RoutingServiceConnectionExt* native()
265     {
266         return &this->native_;
267     }
268
269     RTI_RoutingServiceStreamReaderExt* native_input_discovery_reader()
270     {
271         return input_stream_discovery_forwarder_->native();
272     }
273
274     RTI_RoutingServiceStreamReaderExt* native_output_discovery_reader()
275     {
276         return output_stream_discovery_forwarder_->native();
277     }
278
279 private:
280
281     ConnectionForwarder(
282             const RTI_RoutingServiceStreamReaderListenerExt *native_output_stream_listener,
283             const RTI_RoutingServiceStreamReaderListenerExt *native_input_stream_listener) :
284         connection_(NULL),
285         output_stream_discovery_forwarder_(NULL),
286         output_discovery_listener_(native_output_stream_listener),
287         input_stream_discovery_forwarder_(NULL),
288         input_discovery_listener_(native_input_stream_listener)
289     {
290         RTIOsapiMemory_zero(&native_, sizeof(native_));
291
292         native_.connection_data = static_cast<void*>(this);
293         native_.create_session = ConnectionForwarder::create_session;
294         native_.delete_session = ConnectionForwarder::delete_session;
295         native_.create_stream_reader =
296                 ConnectionForwarder::create_stream_reader;
297         native_.delete_stream_reader =
298                 ConnectionForwarder::delete_stream_reader;
299         native_.create_stream_writer =
300                 ConnectionForwarder::create_stream_writer;
301         native_.delete_stream_writer =
302                 ConnectionForwarder::delete_stream_writer;
303         native_.get_input_stream_discovery_reader =
304                 ConnectionForwarder::get_input_stream_discovery_reader;
305         native_.get_output_stream_discovery_reader =
306                 ConnectionForwarder::get_output_stream_discovery_reader;
307         native_.update = ConnectionForwarder::update;
308     }
309
310     ~ConnectionForwarder()
311     {
312     }
313
314 private:
315     RTI_RoutingServiceConnectionExt native_;
316     Connection *connection_;
317     DiscoveryStreamReaderForwarder *output_stream_discovery_forwarder_;
318     StreamReaderListener output_discovery_listener_;
319     DiscoveryStreamReaderForwarder *input_stream_discovery_forwarder_;
320     StreamReaderListener input_discovery_listener_;
321
322 };
323
324 }}}}
325
326 #endif // RTI_ROUTING_ADAPTER_DETAIL_CONNECTION_FORWARDER_HPP_
```

9.6 DiscoveryStreamReaderForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
```

```
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_DISCOVERY_STREAM_READER_FORWARDER_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_DISCOVERY_STREAM_READER_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <routingservice/routingservice_adapter_new.h>
17
18 #include <rti/routing/adapter/DiscoveryStreamReader.hpp>
19 #include <rti/routing/detail/ForwarderUtils.hpp>
20
21 namespace rti { namespace routing { namespace adapter { namespace detail {
22
23
24 class DiscoveryStreamReaderForwarder {
25 public:
26     static void take(
27             void *native_stream_reader_data,
28             RTI_RoutingServiceSample **sample_array,
29             RTI_RoutingServiceSampleInfo **sample_info_array,
30             int *array_length,
31             RTI_RoutingServiceEnvironment *environment)
32     {
33         DiscoveryStreamReaderForwarder *forwarder =
34                 static_cast<DiscoveryStreamReaderForwarder*>(
35                         native_stream_reader_data);
36
37         *sample_info_array = NULL;
38         *array_length = 0;
39         try {
40             forwarder->stream_reader_->take(forwarder->sample_seq_);
41             *array_length = static_cast<int>(forwarder->sample_seq_.size());
42             if (*array_length > 0) {
43                 *sample_array = reinterpret_cast<RTI_RoutingServiceSample*> (
44                         &forwarder->sample_seq_[0]);
45             }
46         } catch (const std::exception& ex) {
47             RTI_RoutingServiceEnvironment_set_error(
48                     environment,
49                     "%s",
50                     ex.what());
51         } catch (...) {
52             RTI_RoutingServiceEnvironment_set_error(
53                     environment,
54                     "unexpected exception");
55         }
56
57     }
58
59     static void return_loan(
60             void *native_stream_reader_data,
61             RTI_RoutingServiceSample *,
62             RTI_RoutingServiceSampleInfo *,
63             int,
64             RTI_RoutingServiceEnvironment *environment)
65     {
66         DiscoveryStreamReaderForwarder *forwarder =
67                 static_cast<DiscoveryStreamReaderForwarder*>(
68                         native_stream_reader_data);
69
70         try {
71             forwarder->stream_reader_->return_loan(forwarder->sample_seq_);
72         } catch (const std::exception& ex) {
73             RTI_RoutingServiceEnvironment_set_error(
74                     environment,
75                     "%s",
76                     ex.what());
77         } catch (...) {
78             RTI_RoutingServiceEnvironment_set_error(
79                     environment,
80                     "unexpected exception");
81             return;
82         }
83         forwarder->sample_seq_.clear();
84     }
85
86
87     DiscoveryStreamReaderForwarder(
88             DiscoveryStreamReader *stream_reader) :
89             stream_reader_(stream_reader)
90     {
91         RTIOsapiMemory_zero(&native_, sizeof(native_));
```

```
92          native_.stream_reader_data =
93                  static_cast<void *>(this);
94          native_.take =
95                  DiscoveryStreamReaderForwarder::take;
96          native_.return_loan =
97                  DiscoveryStreamReaderForwarder::return_loan;
98      }
99
100
101     ~DiscoveryStreamReaderForwarder()
102     {
103     }
104
105
106     static DiscoveryStreamReaderForwarder* from_native(
107             RTI_RoutingServiceStreamReaderExt *native)
108     {
109         return static_cast<DiscoveryStreamReaderForwarder*>(
110                 native->stream_reader_data);
111     }
112
113     RTI_RoutingServiceStreamReaderExt * native()
114     {
115         return &native_;
116     }
117
118 private:
119     RTI_RoutingServiceStreamReaderExt native_;
120     DiscoveryStreamReader *stream_reader_;
121     std::vector<StreamInfo*> sample_seq_;
122 };
123
124 }}}}
125
126 #endif // RTI_ROUTING_ADAPTER_DETAIL_DISCOVERY_STREAM_READER_FORWARDER_HPP_
```

9.7 ReadProxy.h

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_READ_PROXY_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_READ_PROXY_HPP_
13
14 #include <rti/routing/adapter/StreamReader.hpp>
15
16 namespace rti { namespace routing { namespace adapter { namespace detail {
17
18
19 struct TakeProxy {
20
21     TakeProxy (StreamReader& stream_reader)
22             : stream_reader_(stream_reader)
23     {
24
25     }
26
27     void operator() (
28             std::vector<rti::routing::adapter::StreamReader::SamplePtr>& sample_seq,
29             std::vector<rti::routing::adapter::StreamReader::InfoPtr>& info_seq)
30     {
31         stream_reader_.take(sample_seq, info_seq);
32     }
33
34 private:
35     StreamReader& stream_reader_;
36 };
37
38 struct ReadProxy {
39
```

```
40     ReadProxy (StreamReader& stream_reader)
41             : stream_reader_(stream_reader)
42     {
43
44     }
45
46     void operator() (
47             std::vector<rti::routing::adapter::StreamReader::SamplePtr>& sample_seq,
48             std::vector<rti::routing::adapter::StreamReader::InfoPtr>& info_seq)
49     {
50         stream_reader_.read(sample_seq, info_seq);
51     }
52
53 private:
54     StreamReader& stream_reader_;
55 };
56
57 struct TakeSelectorProxy {
58
59     TakeSelectorProxy(
60             StreamReader& stream_reader,
61             const rti::routing::adapter::SelectorState& state)
62     : stream_reader_(stream_reader), state_(state)
63     {
64
65     }
66
67     void operator() (
68             std::vector<rti::routing::adapter::StreamReader::SamplePtr>& sample_seq,
69             std::vector<rti::routing::adapter::StreamReader::InfoPtr>& info_seq)
70     {
71         stream_reader_.take(sample_seq, info_seq, state_);
72     }
73
74 private:
75     StreamReader& stream_reader_;
76     const rti::routing::adapter::SelectorState& state_;
77 };
78
79 struct ReadSelectorProxy {
80
81     ReadSelectorProxy (
82                 StreamReader& stream_reader,
83                 const rti::routing::adapter::SelectorState& state)
84             : stream_reader_(stream_reader), state_(state)
85     {
86
87     }
88
89     void operator() (
90             std::vector<rti::routing::adapter::StreamReader::SamplePtr>& sample_seq,
91             std::vector<rti::routing::adapter::StreamReader::InfoPtr>& info_seq)
92     {
93         stream_reader_.read(sample_seq, info_seq, state_);
94     }
95
96 private:
97     StreamReader& stream_reader_;
98     const rti::routing::adapter::SelectorState& state_;
99 };
100
101
102 } } } }
103
104
105 #endif /* RTI_ROUTING_ADAPTER_DETAIL_READ_PROXY_HPP_ */
106
```

9.8 SessionForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
```

```
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_SESSION_FORWARDER_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_SESSION_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <rti/routing/adapter/Connection.hpp>
17 #include <rti/routing/detail/UpdatableEntityForwarder.hpp>
18 #include <rti/routing/detail/ForwarderUtils.hpp>
19
20 namespace rti { namespace routing { namespace adapter { namespace detail {
21
22
23 class SessionForwarder {
24 public:
25     static RTI_RoutingServiceSessionExt* create_native(
26             Connection *connection,
27             const RTI_RoutingServiceProperties *native_properties,
28             RTI_RoutingServiceEnvironment *environment)
29     {
30
31         try {
32             using rti::routing::detail::ScopedForwarder;
33
34             // Set properties
35             std::map<std::string, std::string> properties;
36             rti::routing::PropertyAdapter::add_properties_from_native(
37                     properties,
38                     native_properties);
39
40             SessionForwarder *forwarder = new SessionForwarder();
41             ScopedForwarder<Connection, SessionForwarder> scoped(
42                     connection,
43                     forwarder,
44                     environment);
45             // Create session: Note we don't check for NULL because a Session
46             // is not required
47             forwarder->session_ = connection->create_session(properties);
48
49             scoped.release();
50
51             // TRUST-134: the forwarder is not leaked here. It might be leaked
52             // if delete_native() were not to be called, but Routing Service
53             // will make sure that is not the case
54
55             /* coverity[leaked_storage : FALSE] */
56             return forwarder->native();
57         } catch(const std::exception& ex) {
58             RTI_RoutingServiceEnvironment_set_error(
59                     environment,
60                     "%s",
61                     ex.what());
62             return NULL;
63         } catch (...) {
64             RTI_RoutingServiceEnvironment_set_error(
65                     environment,
66                     "unexpected exception");
67             return NULL;
68         }
69     }
70
71     static void delete_native(
72             Connection *connection,
73             RTI_RoutingServiceSessionExt *native_session,
74             RTI_RoutingServiceEnvironment *environment)
75     {
76         SessionForwarder *forwarder = static_cast<SessionForwarder*>(
77                 native_session->session_data);
78         try {
79             // delete session
80             if (forwarder->session_ != NULL) {
81                 connection->delete_session(forwarder->session_);
82                 forwarder->session_ = NULL;
83             }
84
85         } catch(const std::exception& ex) {
86             RTI_RoutingServiceEnvironment_set_error(
87                     environment,
88                     "%s",
```

```
89                      ex.what());
90         } catch (...) {
91         }
92
93         delete forwarder;
94     }
95
96     Session * session()
97     {
98         return session_;
99     }
100
101     RTI_RoutingServiceSessionExt* native()
102     {
103         return &this->native_;
104     }
105
106     static void update(
107             void *native_session_data,
108             const struct RTI_RoutingServiceProperties * native_properties,
109             RTI_RoutingServiceEnvironment * environment)
110     {
111         SessionForwarder *forwarder =
112                 static_cast<SessionForwarder*> (native_session_data);
113
114         rti::routing::detail::UpdatableEntityForwarder::update(
115                 forwarder->session_,
116                 native_properties,
117                 environment);
118     }
119
120 private:
121     SessionForwarder() : session_(NULL)
122     {
123         RTIOsapiMemory_zero(&native_, sizeof(native_));
124         native_.session_data =
125                 static_cast<void*>(this);
126         native_.update =
127                 SessionForwarder::update;
128     }
129
130     ~SessionForwarder()
131     {
132     }
133
134 private:
135     RTI_RoutingServiceSessionExt native_;
136     Session *session_;
137
138 };
139
140 }}}}
141
142 #endif // RTI_ROUTING_ADAPTER_DETAIL_SESSION_FORWARDER_HPP_
```

9.9 StreamReaderForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_STREAM_READER_FORWARDER_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_STREAM_READER_FORWARDER_HPP_
13
14 #include <assert.h>
15 #include <list>
16 #include <rti/core/Exception.hpp>
17
18 #include <rti/routing/adapter/StreamReader.hpp>
19 #include <rti/routing/StreamInfo.hpp>
20 #include <rti/routing/detail/UpdatableEntityForwarder.hpp>
```

```
21 #include <rti/routing/detail/ForwarderUtils.hpp>
22 #include <rti/routing/adapter/detail/DiscoveryStreamReaderForwarder.hpp>
23
24
25 namespace rti { namespace routing { namespace adapter { namespace detail {
26
27 template <bool ReadOrTake, bool HasSelector> struct read_or_take;
28
29
30 template<>
31 struct read_or_take<false, false> {
32
33     static void read(
34             rti::routing::adapter::StreamReader& stream_reader,
35             std::vector<StreamReader::SamplePtr>& sample_seq,
36             std::vector<StreamReader::InfoPtr>& info_seq,
37             const struct RTI_RoutingServiceSelectorState *)
38     {
39         stream_reader.take(sample_seq, info_seq);
40     }
41 };
42
43 template<>
44 struct read_or_take<true, false> {
45
46     static void read(
47             rti::routing::adapter::StreamReader& stream_reader,
48             std::vector<StreamReader::SamplePtr>& sample_seq,
49             std::vector<StreamReader::InfoPtr>& info_seq,
50             const struct RTI_RoutingServiceSelectorState *)
51     {
52         stream_reader.read(sample_seq, info_seq);
53     }
54 };
55
56 template<>
57 struct read_or_take<false, true> {
58
59     static void read(
60             rti::routing::adapter::StreamReader& stream_reader,
61             std::vector<StreamReader::SamplePtr>& sample_seq,
62             std::vector<StreamReader::InfoPtr>& info_seq,
63             const struct RTI_RoutingServiceSelectorState *native_selector)
64     {
65         stream_reader.take(
66                 sample_seq,
67                 info_seq,
68                 SelectorState(*native_selector));
69     }
70 };
71
72
73 template<>
74 struct read_or_take<true, true> {
75
76     static void read(
77             rti::routing::adapter::StreamReader& stream_reader,
78             std::vector<StreamReader::SamplePtr>& sample_seq,
79             std::vector<StreamReader::InfoPtr>& info_seq,
80             const struct RTI_RoutingServiceSelectorState *native_selector)
81     {
82         stream_reader.read(
83                 sample_seq,
84                 info_seq,
85                 SelectorState(*native_selector));
86     }
87 };
88
89
90 struct SamplesHolder {
91 public:
92
93     SamplesHolder()
94     {
95     }
96
97     std::vector<StreamReader::SamplePtr> sample_seq_;
98     std::vector<StreamReader::InfoPtr> info_seq_;
99 };
100
101 class StreamReaderForwarder {
```

```
102 public:
103     static RTI_RoutingServiceStreamReaderExt* create_native(
104             Connection *connection,
105             Session *session,
106             const struct RTI_RoutingServiceStreamInfo *native_stream_info,
107             const struct RTI_RoutingServiceProperties *native_properties,
108             const struct RTI_RoutingServiceStreamReaderListenerExt *native_listener,
109             RTI_RoutingServiceEnvironment *environment)
110     {
111 
112         try {
113             using rti::routing::detail::ScopedForwarder;
114 
115             StreamInfo stream_info(*native_stream_info);
116 
117             std::map<std::string, std::string> properties;
118             rti::routing::PropertyAdapter::add_properties_from_native(
119                     properties,
120                     native_properties);
121 
122             StreamReaderForwarder *forwarder = new StreamReaderForwarder(
123                     native_listener);
124             ScopedForwarder<Connection, StreamReaderForwarder> scoped(
125                     connection,
126                     forwarder,
127                     environment);
128             forwarder->stream_reader_ = connection->create_stream_reader(
129                     session,
130                     stream_info,
131                     properties,
132                     &forwarder->listener_);
133             RTI_ROUTING_THROW_ON_NULL(forwarder->stream_reader_);
134 
135             scoped.release();
136 
137             // TRUST-134: the forwarder is not leaked here. It might be leaked
138             // if delete_native() were not to be called, but Routing Service
139             // will make sure that is not the case
140 
141             /* coverity[leaked_storage : FALSE] */
142             return forwarder->native();
143         } catch(const std::exception& ex) {
144             RTI_RoutingServiceEnvironment_set_error(
145                     environment,
146                     "%s",
147                     ex.what());
148             return NULL;
149         } catch (...) {
150             RTI_RoutingServiceEnvironment_set_error(
151                     environment,
152                     "unexpected exception");
153             return NULL;
154         }
155 
156 
157     }
158 
159     static void delete_native(
160             Connection *connection,
161             RTI_RoutingServiceStreamReaderExt *native_stream_reader,
162             RTI_RoutingServiceEnvironment *environment)
163     {
164         StreamReaderForwarder *stream_reader_forwarder =
165                 from_native(native_stream_reader);
166         try {
167             if (stream_reader_forwarder->stream_reader_ != NULL) {
168                 connection->delete_stream_reader(
169                         stream_reader_forwarder->stream_reader_);
170                 stream_reader_forwarder->stream_reader_ = NULL;
171             }
172         } catch(const std::exception& ex) {
173             RTI_RoutingServiceEnvironment_set_error(
174                     environment,
175                     "%s",
176                     ex.what());
177         } catch (...) {
178             RTI_RoutingServiceEnvironment_set_error(
179                     environment,
180                     "unexpected exception");
181         }
182 
```

```
183         delete stream_reader_forwarder;
184     }
185
186
187     RTI_RoutingServiceStreamReaderExt* native()
188     {
189         return &this->native_;
190     }
191
192
193 private:
194
195     StreamReaderForwarder(
196             const RTI_RoutingServiceStreamReaderListenerExt *native_listener) :
197             stream_reader_(NULL),
198             listener_(native_listener)
199     {
200         RTIOsapiMemory_zero(&native_, sizeof(native_));
201         native_.stream_reader_data =
202                 static_cast<void *>(this);
203         native_.take =
204                 StreamReaderForwarder::take;
205         native_.read =
206                 StreamReaderForwarder::read;
207         native_.take_w_selector =
208                 StreamReaderForwarder::take_with_selector;
209         native_.read_w_selector =
210                 StreamReaderForwarder::read_with_selector;
211         native_.return_loan =
212                 StreamReaderForwarder::return_loan;
213         native_.create_content_query =
214                 StreamReaderForwarder::create_content_query;
215         native_.delete_content_query =
216                 StreamReaderForwarder::delete_content_query;
217         native_.update =
218                 StreamReaderForwarder::update;
219     }
220
221     ~StreamReaderForwarder()
222     {
223         // delete holders
224         for (std::list<SamplesHolder*>::iterator it = holder_pool_.begin();
225              it != holder_pool_.end();
226              ++it) {
227             delete (*it);
228         }
229     }
230
231
232     static void take(
233         void *native_stream_reader_data,
234         RTI_RoutingServiceSample **sample_array,
235         RTI_RoutingServiceSampleInfo **sample_info_array,
236         int *array_length,
237         RTI_RoutingServiceEnvironment *environment)
238     {
239         proxy_read<false, false>(
240                 native_stream_reader_data,
241                 sample_array,
242                 sample_info_array,
243                 array_length,
244                 NULL,
245                 environment);
246     }
247
248     static void read(
249         void *native_stream_reader_data,
250         RTI_RoutingServiceSample **sample_array,
251         RTI_RoutingServiceSampleInfo **sample_info_array,
252         int *array_length,
253         RTI_RoutingServiceEnvironment *environment)
254     {
255         proxy_read<true, false>(
256                 native_stream_reader_data,
257                 sample_array,
258                 sample_info_array,
259                 array_length,
260                 NULL,
261                 environment);
262     }
263
```

```
264     static void take_with_selector(
265         void *native_stream_reader_data,
266         RTI_RoutingServiceSample **sample_array,
267         RTI_RoutingServiceSampleInfo **sample_info_array,
268         int *array_length,
269         const struct RTI_RoutingServiceSelectorState *native_selector,
270         RTI_RoutingServiceEnvironment *environment)
271     {
272         proxy_read<false, true>(
273                 native_stream_reader_data,
274                 sample_array,
275                 sample_info_array,
276                 array_length,
277                 native_selector,
278                 environment);
279     }
280
281     static void read_with_selector(
282         void *native_stream_reader_data,
283         RTI_RoutingServiceSample **sample_array,
284         RTI_RoutingServiceSampleInfo **sample_info_array,
285         int *array_length,
286         const struct RTI_RoutingServiceSelectorState *native_selector,
287         RTI_RoutingServiceEnvironment *environment)
288     {
289         proxy_read<true, true>(
290                 native_stream_reader_data,
291                 sample_array,
292                 sample_info_array,
293                 array_length,
294                 native_selector,
295                 environment);
296     }
297
298     static void return_loan(
299         void *native_stream_reader_data,
300         RTI_RoutingServiceSample *native_samples,
301         RTI_RoutingServiceSampleInfo *,
302         int count,
303         RTI_RoutingServiceEnvironment *environment)
304     {
305         StreamReaderForwarder *forwarder =
306                 static_cast<StreamReaderForwarder*>(native_stream_reader_data);
307         assert(forwarder != NULL);
308
309         SamplesHolder *holder = static_cast<SamplesHolder*>(
310                 native_samples[count]);
311         assert(holder != NULL);
312
313
314         try {
315             forwarder->stream_reader_->return_loan(
316                     holder->sample_seq_,
317                     holder->info_seq_);
318         } catch (const std::exception& ex) {
319             RTI_RoutingServiceEnvironment_set_error(
320                     environment,
321                     "%s",
322                     ex.what());
323         } catch (...) {
324             RTI_RoutingServiceEnvironment_set_error(
325                     environment,
326                     "unexpected exception");
327         }
328
329         forwarder->return_holder(holder);
330     }
331
332     static void update(
333             void *native_stream_reader_data,
334             const struct RTI_RoutingServiceProperties * native_properties,
335             RTI_RoutingServiceEnvironment * environment)
336     {
337
338         StreamReaderForwarder *stream_reader_forwarder =
339                 static_cast<StreamReaderForwarder*>(native_stream_reader_data);
340
341         rti::routing::detail::UpdatableEntityForwarder::update(
342                 stream_reader_forwarder->stream_reader_,
343                 native_properties,
344                 environment);
```

```
345     }
346
347     static void* create_content_query(
348         void *native_stream_reader_data,
349         RTI_RoutingServiceSelectorStateQueryData old_query_data,
350         const struct RTI_RoutingServiceSelectorContent *content,
351         RTI_RoutingServiceEnvironment *environment)
352     {
353         StreamReaderForwarder *forwarder =
354                 static_cast<StreamReaderForwarder*>(native_stream_reader_data);
355
356         void *query_data = NULL;
357         try {
358             dds::topic::Filter filter(
359                     content->expression == NULL ? "" : content->expression,
360                     rti::core::native_conversions::from_native<std::string>(
361                             content->expression_parameters));
362             query_data = forwarder->stream_reader_->create_content_query(
363                     old_query_data,
364                     filter);
365         } catch (const std::exception& ex) {
366             RTI_RoutingServiceEnvironment_set_error(
367                     environment,
368                     "%s",
369                     ex.what());
370         } catch (...) {
371             RTI_RoutingServiceEnvironment_set_error(
372                     environment,
373                     "unexpected exception");
374         }
375
376         return query_data;
377     }
378
379     static void delete_content_query(
380         void *native_stream_reader_data,
381         RTI_RoutingServiceSelectorStateQueryData query_data,
382         RTI_RoutingServiceEnvironment *environment)
383     {
384         StreamReaderForwarder *forwarder =
385                 static_cast<StreamReaderForwarder*>(native_stream_reader_data);
386         try {
387             forwarder->stream_reader_->delete_content_query(query_data);
388         } catch (const std::exception& ex) {
389             RTI_RoutingServiceEnvironment_set_error(
390                     environment,
391                     "%s",
392                     ex.what());
393         } catch (...) {
394             RTI_RoutingServiceEnvironment_set_error(
395                     environment,
396                     "unexpected exception");
397         }
398     }
399
400     static StreamReaderForwarder* from_native(
401             RTI_RoutingServiceStreamReaderExt *native)
402     {
403         return static_cast<StreamReaderForwarder*>(native->stream_reader_data);
404     }
405
406     template <bool ReadOrTake, bool HasSelector>
407     static void proxy_read(
408         void *native_stream_reader_data,
409         RTI_RoutingServiceSample **sample_array,
410         RTI_RoutingServiceSampleInfo **sample_info_array,
411         int *array_length,
412         const struct RTI_RoutingServiceSelectorState *native_selector,
413         RTI_RoutingServiceEnvironment *environment)
414     {
415         StreamReaderForwarder *forwarder =
416                 static_cast<StreamReaderForwarder*>(native_stream_reader_data);
417         assert(forwarder != NULL);
418
419         SamplesHolder *holder = forwarder->get_holder();
420
421         *array_length = 0;
422         try {
423             read_or_take<ReadOrTake, HasSelector>::read(
424                     *forwarder->stream_reader_,
425                     holder->sample_seq_,
```

```
426                     holder->info_seq_,
427                     native_selector);
428             if (!holder->info_seq_.empty()
429                     && (holder->info_seq_.size() != holder->sample_seq_.size())) {
430                 throw dds::core::PreconditionNotMetError(
431                         "sample and info sequences length mismatch");
432             }
433             /*
434              * We reserve a hidden slot in the sample_seq to keep the original
435              * loaned returned from the native StreamReader. This allows us
436              * to quickly access it on return_loan(), instead of having to search
437              * in a list.
438              *
439              * TRUST-1466: reserving just 1 more memory spot can be inefficient,
440              * and we're doing this for every read/take operation called in C++.
441              */
442             if (holder->sample_seq_.size() + 1
443                     >= holder->sample_seq_.capacity()) {
444                 holder->sample_seq_.reserve(
445                         holder->sample_seq_.capacity() == 0
446                                 ? 1
447                                 : 2 * holder->sample_seq_.capacity());
448             }
449             *(holder->sample_seq_.data() + holder->sample_seq_.size()) = holder;
450         } catch (const std::exception& ex) {
451             RTI_RoutingServiceEnvironment_set_error(
452                     environment,
453                     "%s",
454                     ex.what());
455             forwarder->return_holder(holder);
456             return;
457         } catch (...) {
458             RTI_RoutingServiceEnvironment_set_error(
459                     environment,
460                     "unexpected exception");
461             forwarder->return_holder(holder);
462             return;
463         }
464
465         // direct mapping
466         *array_length = static_cast<int>(holder->sample_seq_.size());
467         *sample_array = holder->sample_seq_.data();
468         if (holder->info_seq_.empty()) {
469             *sample_info_array = NULL;
470         } else {
471             *sample_info_array = &holder->info_seq_[0];
472         }
473     }
474
475     SamplesHolder* get_holder()
476     {
477         if (holder_pool_.size() == 0) {
478             return new SamplesHolder();
479         }
480
481         SamplesHolder *holder = holder_pool_.front();
482         holder_pool_.pop_front();
483
484         return holder;
485     }
486
487     void return_holder(SamplesHolder *holder)
488     {
489         holder->sample_seq_.clear();
490         holder->info_seq_.clear();
491         holder_pool_.push_front(holder);
492     }
493
494 private:
495
496     RTI_RoutingServiceStreamReaderExt native_;
497     StreamReader *stream_reader_;
498     StreamReaderListener listener_;
499     std::list<SamplesHolder*> holder_pool_;
500 };
501
502 }}}}
503
504 #endif // RTI_ROUTING_ADAPTER_DETAIL_STREAM_READER_FORWARDER_HPP_
```

9.10 StreamReaderListenerForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_STREAM_READER_LISTENER_FORWARDER_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_STREAM_READER_LISTENER_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <routingservice/routingservice_adapter_new.h>
17
18 #include <rti/routing/adapter/StreamReader.hpp>
19 #include <rti/routing/adapter/DiscoveryStreamReader.hpp>
20
21 namespace rti { namespace routing { namespace adapter { namespace detail {
22
36 class StreamReaderListener {
37 public:
38
39 public:
48     void on_data_available(
49             rti::routing::adapter::StreamReader *stream_reader)
50     {
51         native_listener_.on_data_available(
52                 static_cast<void*>(stream_reader),
53                 native_listener_.listener_data);
54     }
55
56
57     StreamReaderListener(
58             const RTI_RoutingServiceStreamReaderListenerExt *native_listener) :
59             native_listener_(*native_listener)
60     {
61     }
62
63     ~StreamReaderListener()
64     {
65     }
66
67 private:
68
69     RTI_RoutingServiceStreamReaderListenerExt native_listener_;
70
71 };
72
73 }}}}
74
75 #endif // RTI_ROUTING_ADAPTER_DETAIL_STREAM_READER_LISTENER_FORWARDER_HPP_
```

9.11 StreamWriterForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_DETAIL_STREAM_WRITER_FORWARDER_HPP_
12 #define RTI_ROUTING_ADAPTER_DETAIL_STREAM_WRITER_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <rti/routing/adapter/StreamWriter.hpp>
17 #include <rti/routing/detail/UpdatableEntityForwarder.hpp>
18 #include <rti/routing/detail/ForwarderUtils.hpp>
```

```
19
20 namespace rti { namespace routing { namespace adapter { namespace detail {
21
22
23 class StreamWriterForwarder {
24 public:
25
26     static RTI_RoutingServiceStreamWriterExt* create_native(
27             Connection *connection,
28             Session *session,
29             const struct RTI_RoutingServiceStreamInfo *native_stream_info,
30             const struct RTI_RoutingServiceProperties *native_properties,
31             RTI_RoutingServiceEnvironment *environment)
32     {
33         try {
34             using rti::routing::detail::ScopedForwarder;
35
36             StreamInfo stream_info(*native_stream_info);
37
38             std::map<std::string, std::string> properties;
39             rti::routing::PropertyAdapter::add_properties_from_native(
40                     properties,
41                     native_properties);
42
43             StreamWriterForwarder *forwarder = new StreamWriterForwarder(
44                     NULL);
45             ScopedForwarder<Connection, StreamWriterForwarder> scoped(
46                     connection,
47                     forwarder,
48                     environment);
49             forwarder->stream_writer_ = connection->create_stream_writer(
50                     session,
51                     stream_info,
52                     properties);
53             RTI_ROUTING_THROW_ON_NULL(forwarder->stream_writer_);
54
55             scoped.release();
56
57             // TRUST-134: the forwarder is not leaked here. It might be leaked
58             // if delete_native() were not to be called, but Routing Service
59             // will make sure that is not the case
60
61             /* coverity[leaked_storage : FALSE] */
62             return forwarder->native();
63         } catch(const std::exception& ex) {
64             RTI_RoutingServiceEnvironment_set_error(
65                     environment,
66                     "%s",
67                     ex.what());
68             return NULL;
69         } catch (...) {
70             RTI_RoutingServiceEnvironment_set_error(
71                     environment,
72                     "unexpected exception");
73             return NULL;
74         }
75
76     }
77
78     static void delete_native(
79             Connection *connection,
80             RTI_RoutingServiceStreamWriterExt *native_stream_writer,
81             RTI_RoutingServiceEnvironment *environment)
82     {
83         StreamWriterForwarder *stream_writer_forwarder =
84                 static_cast<StreamWriterForwarder*>(
85                     native_stream_writer->stream_writer_data);
86         try {
87             if (stream_writer_forwarder->stream_writer_ != NULL) {
88                 connection->delete_stream_writer(
89                         stream_writer_forwarder->stream_writer_);
90                 stream_writer_forwarder->stream_writer_ = NULL;
91             }
92         } catch(const std::exception& ex) {
93             RTI_RoutingServiceEnvironment_set_error(
94                     environment,
95                     "%s",
96                     ex.what());
97         } catch (...) {
98         }
99
```

```
100         delete stream_writer_forwarder;
101     }
102 
103 
104     static int write(
105         void *native_stream_writer_data,
106         const RTI_RoutingServiceSample *sample_array,
107         const RTI_RoutingServiceSampleInfo *sample_info_array,
108         int array_length,
109         RTI_RoutingServiceEnvironment *environment)
110     {
111 
112         StreamWriterForwarder *forwarder =
113                 static_cast<StreamWriterForwarder*>(native_stream_writer_data);
114 
115         try {
116             RTI_ROUTING_SAMPLE_VECTOR_COPY_FROM_NATIVE(
117                     forwarder->sample_seq_,
118                     sample_array,
119                     array_length);
120             if (sample_info_array != NULL) {
121                 RTI_ROUTING_SAMPLE_VECTOR_COPY_FROM_NATIVE(
122                         forwarder->info_seq_,
123                         sample_info_array,
124                         array_length);
125             } else {
126                 forwarder->info_seq_.clear();
127             }
128             return forwarder->stream_writer_->write(
129                     forwarder->sample_seq_,
130                     forwarder->info_seq_);
131         } catch (const std::exception& ex) {
132             RTI_RoutingServiceEnvironment_set_error(
133                     environment,
134                     "%s",
135                     ex.what());
136         } catch (...) {
137              RTI_RoutingServiceEnvironment_set_error(
138                     environment,
139                     "unexpected exception");
140         }
141 
142         return 0;
143     }
144 
145     static void update(
146             void *native_stream_writer_data,
147             const struct RTI_RoutingServiceProperties * native_properties,
148             RTI_RoutingServiceEnvironment * environment)
149     {
150 
151         StreamWriterForwarder *stream_writer_forwarder =
152                 static_cast<StreamWriterForwarder*> (native_stream_writer_data);
153 
154         rti::routing::detail::UpdatableEntityForwarder::update(
155                 stream_writer_forwarder->stream_writer_,
156                 native_properties,
157                 environment);
158     }
159 
160      RTI_RoutingServiceStreamWriterExt* native()
161     {
162         return &this->native_;
163     }
164 
165 private:
166 
167     StreamWriterForwarder(
168             StreamWriter *stream_writer) :
169             stream_writer_(stream_writer)
170     {
171         RTIOsapiMemory_zero(&native_, sizeof(native_));
172         native_.stream_writer_data =
173                 static_cast<void*>(this);
174         native_.write =
175                 StreamWriterForwarder::write;
176         native_.update =
177                 StreamWriterForwarder::update;
178 
179     }
180 
```

```
181     ~StreamWriterForwarder()
182     {
183     }
184
185 private:
186     RTI_RoutingServiceStreamWriterExt native_;
187     StreamWriter *stream_writer_;
188     std::vector<StreamWriter::SamplePtr> sample_seq_;
189     std::vector<StreamWriter::InfoPtr> info_seq_;
190
191 };
192
193 }}}}
194
195 #endif // RTI_ROUTING_ADAPTER_DETAIL_STREAM_WRITER_FORWARDER_HPP_
```

9.12 DiscoveryStreamReader.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_DISCOVERY_STREAM_READER_HPP_
12 #define RTI_ROUTING_ADAPTER_DISCOVERY_STREAM_READER_HPP_
13
14 #include <stdlib.h>
15
16 #include <rti/routing/UpdatableEntity.hpp>
17 #include <rti/routing/StreamInfo.hpp>
18 #include <rti/routing/detail/ForwarderUtils.hpp>
19
20
21 namespace rti { namespace routing { namespace adapter {
22
34 class DiscoveryStreamReader : public StreamReader{
35
36     using StreamReader::take;
37     using StreamReader::return_loan;
42     void take(
43             std::vector<SamplePtr>& sample_seq,
44             std::vector<InfoPtr>&) RTI_FINAL
45     {
46         take(sample_seq_);
47         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(sample_seq, sample_seq_);
48     }
49
54     void return_loan(
55             std::vector<SamplePtr>& sample_seq,
56             std::vector<InfoPtr>&) RTI_FINAL
57     {
58         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(sample_seq_, sample_seq);
59         return_loan(sample_seq_);
60         sample_seq_.clear();
61     }
62
63     void read(
64             std::vector<SamplePtr>&,
65             std::vector<InfoPtr>&) RTI_FINAL
66     {
67     }
68
69     void take(
70             std::vector<SamplePtr>&,
71             std::vector<InfoPtr>&,
72             const SelectorState&) RTI_FINAL
73     {
74     }
75
76     void read(
77             std::vector<SamplePtr>&,
78             std::vector<InfoPtr>&,
```

```
79              const SelectorState&) RTI_FINAL
80      {
81      }
82
83       virtual void* create_content_query(
84              void *,
85              const dds::topic::Filter&) RTI_FINAL
86      {
87          return NULL;
88      }
89
90
91      virtual void delete_content_query(void*) RTI_FINAL
92      {
93
94      }
95
96
97  public:
98
116      virtual void take(
117              std::vector<rti::routing::StreamInfo*>& sample_seq) = 0;
118
130      virtual void return_loan(
131              std::vector<rti::routing::StreamInfo*>& sample_seq) = 0;
132
133      /*
134       * @brief Virtual destructor
135       */
136      virtual ~DiscoveryStreamReader()
137      {
138      }
139
140  private:
141      std::vector<StreamInfo*> sample_seq_;
142  };
143
144  }}}
145
146  #endif // RTI_ROUTING_ADAPTER_DISCOVERY_STREAM_READER_HPP_
147
```

9.13 Session.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10  #ifndef RTI_ROUTING_ADAPTER_SESSION_HPP_
11  #define RTI_ROUTING_ADAPTER_SESSION_HPP_
12
13  #include <rti/routing/UpdatableEntity.hpp>
14
15  namespace rti { namespace routing { namespace adapter {
16
28  class Session : public UpdatableEntity {
29
30  public:
31
35      virtual ~Session()
36      {
37      }
38  };
39
40  }}}
41
42  #endif // RTI_ROUTING_ADAPTER_SESSION_HPP_
```

9.14 StreamReader.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_STREAM_READER_HPP_
12 #define RTI_ROUTING_ADAPTER_STREAM_READER_HPP_
13
14 #include <vector>
15
16 #include <dds/core/SafeEnumeration.hpp>
17 #include <dds/sub/status/DataState.hpp>
18 #include <dds/core/InstanceHandle.hpp>
19 #include <dds/core/types.hpp>
20 #include <dds/topic/Filter.hpp>
21 #include <dds/core/xtypes/DynamicData.hpp>
22 #include <dds/sub/SampleInfo.hpp>
23 #include <rti/core/NativeValueType.hpp>
24 #include <rti/routing/UpdatableEntity.hpp>
25 #include <rti/routing/detail/ForwarderUtils.hpp>
26
27 namespace rti { namespace routing { namespace adapter {
28
29 class SelectorState;
30
45 class StreamReader : public UpdatableEntity{
46
47 public:
48     typedef void* SamplePtr;
49     typedef void* InfoPtr;
50
89     virtual void take(
90             std::vector<SamplePtr>& sample_seq,
91             std::vector<InfoPtr>& info_seq) = 0;
92
100     virtual void read(
101             std::vector<SamplePtr>& sample_seq,
102             std::vector<InfoPtr>& info_seq) = 0;
103
120     virtual void take(
121             std::vector<SamplePtr>& sample_seq,
122             std::vector<InfoPtr>& info_seq,
123             const SelectorState& selector_state) = 0;
124
141     virtual void read(
142             std::vector<SamplePtr>& sample_seq,
143             std::vector<InfoPtr>& info_seq,
144             const SelectorState& selector_state) = 0;
145
161     virtual void return_loan(
162             std::vector<SamplePtr>& sample_seq,
163             std::vector<InfoPtr>& info_seq) = 0;
164
187     virtual void* create_content_query(
188             void *old_query_data,
189             const dds::topic::Filter& filter) = 0;
190
196     virtual void delete_content_query(void* query_data) = 0;
197
201     virtual ~StreamReader()
202     {
203     }
204
205 };
206
207
219 template <typename Data, typename Info>
220 class TStreamReader : public StreamReader {
221
222 public:
223
227     typedef Data DataRep;
231     typedef Info InfoRep;
```

```
232
233     using StreamReader::take;
234     using StreamReader::read;
235     using StreamReader::return_loan;
236
241     void take(
242             std::vector<SamplePtr>& sample_seq,
243             std::vector<InfoPtr>& info_seq) RTI_FINAL
244     {
245         take(sample_seq_, info_seq_);
246         convert_samples(sample_seq, info_seq);
247     }
248
253     void read(
254             std::vector<SamplePtr>& sample_seq,
255             std::vector<InfoPtr>& info_seq) RTI_FINAL
256     {
257         read(sample_seq_, info_seq_);
258         convert_samples(sample_seq, info_seq);
259     }
260
265     void take(
266             std::vector<SamplePtr>& sample_seq,
267             std::vector<InfoPtr>& info_seq,
268             const SelectorState& selector_state) RTI_FINAL
269     {
270         take(sample_seq_, info_seq_, selector_state);
271         convert_samples(sample_seq, info_seq);
272     }
273
278     void read(
279             std::vector<SamplePtr>& sample_seq,
280             std::vector<InfoPtr>& info_seq,
281             const SelectorState& selector_state) RTI_FINAL
282     {
283         read(sample_seq_, info_seq_, selector_state);
284         convert_samples(sample_seq, info_seq);
285     }
286
291     void return_loan(
292             std::vector<SamplePtr>& sample_seq,
293             std::vector<InfoPtr>& info_seq) RTI_FINAL
294     {
295         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(sample_seq_, sample_seq);
296         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(info_seq_, info_seq);
297         return_loan(sample_seq_, info_seq_);
298         sample_seq_.clear();
299         info_seq_.clear();
300     }
301
307     virtual void take(
308             std::vector<Data*>& sample_seq,
309             std::vector<Info*>& info_seq) = 0;
310
316     virtual void read(
317             std::vector<Data*>& sample_seq,
318             std::vector<Info*>& info_seq) = 0;
319
325     virtual void take(
326             std::vector<Data*>& sample_seq,
327             std::vector<Info*>& info_seq,
328             const SelectorState& selector_state) = 0;
329
335     virtual void read(
336             std::vector<Data*>& sample_seq,
337             std::vector<Info*>& info_seq,
338             const SelectorState& selector_state) = 0;
339
345     virtual void return_loan(
346             std::vector<Data*>& sample_seq,
347             std::vector<Info*>& info_seq) = 0;
348
349     /*
350      * @brief Virtual destructor
351      */
352     virtual ~TStreamReader()
353     {
354
355     }
356
357 private:
```

```
358     void convert_samples(
359             std::vector<SamplePtr>& sample_seq,
360             std::vector<InfoPtr>& info_seq)
361     {
362         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(sample_seq, sample_seq_);
363         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(info_seq, info_seq_);
364     }
365
366 private:
367     std::vector<Data*> sample_seq_;
368     std::vector<Info*> info_seq_;
369
370 };
371
380 template <typename Data, typename Info>
381 class NoOpStreamReader : public TStreamReader<Data, Info> {
382
383 public:
384     virtual void take(
385             std::vector<Data*>&,
386             std::vector<Info*>&) RTI_OVERRIDE
387     {
388     }
389
390     virtual void read(
391             std::vector<Data*>&,
392             std::vector<Info*>&) RTI_OVERRIDE
393     {
394     }
395
396     virtual void take(
397             std::vector<Data*>&,
398             std::vector<Info*>&,
399             const SelectorState&) RTI_OVERRIDE
400     {
401     }
402
403     virtual void read(
404             std::vector<Data*>&,
405             std::vector<Info*>&,
406             const SelectorState&) RTI_OVERRIDE
407     {
408     }
409
410     virtual void return_loan(
411             std::vector<Data*>&,
412             std::vector<Info*>&) RTI_OVERRIDE
413     {
414     }
415
416     virtual void* create_content_query(
417             void *,
418             const dds::topic::Filter&) RTI_OVERRIDE
419     {
420         return NULL;
421     }
422
423
424     virtual void delete_content_query(void*) RTI_OVERRIDE
425     {
426
427     }
428
429
430
431     virtual ~NoOpStreamReader()
432     {
433     }
434
435 };
436
444 typedef NoOpStreamReader<dds::core::xtypes::DynamicData, dds::sub::SampleInfo>
445         DynamicDataStreamReader;
446
447
448
449 class SelectorStateAdapter {
450 public:
451     typedef RTI_RoutingServiceSelectorState native_type;
452
453     static void initialize(native_type& native_value)
```

```
454     {
455         RTI_RoutingServiceSelectorState native_state =
456                 RTI_RoutingServiceSelectorState_INITIALIZER;
457         memcpy(&native_value, &native_state, sizeof(native_state));
458     }
459
460     static void finalize(native_type& native_value)
461     {
462         if (native_value.content.expression != NULL) {
463             DDS_String_free(native_value.content.expression);
464         }
465         DDS_StringSeq_finalize(&native_value.content.expression_parameters);
466     }
467
468     static void copy(native_type& destination, const native_type& source)
469     {
470         destination.sample_state = source.sample_state;
471         destination.view_state = source.view_state;
472         destination.instance_state = source.instance_state;
473         destination.instance_handle = source.instance_handle;
474         destination.instance_selection = source.instance_selection;
475         destination.sample_count_max = source.sample_count_max;
476         destination.query_data = source.query_data;
477         char *result = DDS_String_replace(
478                 &destination.content.expression,
479                 source.content.expression);
480         if (source.content.expression != NULL && result == NULL) {
481             throw std::bad_alloc();
482         }
483         if (DDS_StringSeq_copy(
484                 &destination.content.expression_parameters,
485                 &source.content.expression_parameters) == NULL) {
486             throw std::bad_alloc();
487         }
488     }
489
490     static bool equals(const native_type& first, const native_type& second)
491     {
492         if (first.sample_state != second.sample_state) {
493             return false;
494         }
495         if (first.view_state != second.view_state) {
496             return false;
497         }
498         if (first.instance_state != second.instance_state) {
499             return false;
500         }
501         if (first.instance_selection != second.instance_selection) {
502             return false;
503         }
504         if (first.sample_count_max != second.sample_count_max) {
505             return false;
506         }
507         if (first.query_data != second.query_data) {
508             return false;
509         }
510         if (first.content.expression != NULL
511                 && second.content.expression != NULL
512                 && strcmp(first.content.expression, second.content.expression) != 0) {
513             return false;
514         }
515         return DDS_StringSeq_equals(
516                 &first.content.expression_parameters,
517                 &second.content.expression_parameters);
518     }
519
520 };
521
522 } }
523
524 // native_type_traits needs to be defined in rti::core
525 namespace core {
526
527 template <>
528 struct native_type_traits<rti::routing::adapter::SelectorState> {
529     typedef rti::routing::adapter::SelectorStateAdapter adapter_type;
530     typedef RTI_RoutingServiceSelectorState native_type;
531 };
532
533 }
534
```

```
535 namespace routing { namespace adapter {
536
547 class SelectorState : public rti::core::NativeValueType<SelectorState> {
548 public:
549     RTI_NATIVE_VALUE_TYPE_DEFINE_DEFAULT_MOVE_OPERATIONS(SelectorState)
550
551     typedef rti::core::NativeValueType<SelectorState> Base;
552 public:
553     SelectorState()
554     {
555
556     }
557
558     SelectorState(const RTI_RoutingServiceSelectorState &native)
559             : Base(native)
560     {
561     }
562
563     dds::sub::status::DataState state() const
564     {
565         return dds::sub::status::DataState(
566                 dds::sub::status::SampleState(native().sample_state),
567                 dds::sub::status::ViewState(native().view_state),
568                 dds::sub::status::InstanceState(native().instance_state));
569     }
570
571     SelectorState& state(const dds::sub::status::DataState& state)
572     {
573          native().sample_state = static_cast<DDS_SampleStateMask>(
574                  state.sample_state().to_ulong());
575          native().view_state = static_cast<DDS_ViewStateMask>(
576                  state.view_state().to_ulong());
577          native().instance_state = static_cast<DDS_InstanceStateMask>(
578                  state.instance_state().to_ulong());
579          return *this;
580     }
581
582     int32_t max_samples() const
583     {
584        return native().sample_count_max;
585     }
586
587     SelectorState& max_samples(const int32_t count)
588     {
589         native().sample_count_max = count;
590         return *this;
591     }
592
593     dds::core::InstanceHandle instance() const
594     {
595         if (native().instance_selection
596                 == RTI_ROUTING_SERVICE_THIS_INSTANCE_SELECTION) {
597             return rti::core::native_conversions::from_native_handle(
598                         native().instance_handle);
599         }
600
601         return dds::core::InstanceHandle::nil();
602     }
603
604     SelectorState& instance(const dds::core::InstanceHandle& handle)
605     {
606         native().instance_handle =
607                 static_cast<DDS_InstanceHandle_t>(handle->native());
608         native().instance_selection = RTI_ROUTING_SERVICE_THIS_INSTANCE_SELECTION;
609         return *this;
610     }
611
612     dds::core::InstanceHandle next_instance() const
613     {
614         if (native().instance_selection
615                 == RTI_ROUTING_SERVICE_NEXT_INSTANCE_SELECTION) {
616             return rti::core::native_conversions::from_native_handle(
617                         native().instance_handle);
618         }
619
620         return dds::core::InstanceHandle::nil();
621     }
622
623     SelectorState& next_instance(
624             const dds::core::InstanceHandle& handle)
625     {
```

```
626         native().instance_handle =
627                 static_cast<DDS_InstanceHandle_t>(handle->native());
628         native().instance_selection = RTI_ROUTING_SERVICE_NEXT_INSTANCE_SELECTION;
629         return *this;
630     }
631
632     void* content()
633     {
634         return native().query_data;
635     }
636
637     SelectorState& content(void *query_data)
638     {
639         native().query_data = query_data;
640         return *this;
641     }
642
643     dds::topic::Filter filter() const
644     {
645         return dds::topic::Filter(
646                 native().content.expression == NULL ? "" : native().content.expression,
647                 rti::core::native_conversions::from_native<std::string>(
648                         native().content.expression_parameters));
649     }
650
651     SelectorState& filter(const dds::topic::Filter& filter)
652     {
653         DDS_String_replace(
654                 &native().content.expression,
655                 filter.expression().c_str());
656         rti::core::native_conversions::to_native(
657                 native().content.expression_parameters,
658                 filter->parameters());
659         return *this;
660     }
661
662 };
663
664 }}}
665
666 #endif // RTI_ROUTING_ADAPTER_STREAM_READER_HPP_
```

9.15 StreamReaderListener.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_ADAPTER_STREAM_READER_LISTENER_HPP_
12 #define RTI_ROUTING_ADAPTER_STREAM_READER_LISTENER_HPP_
13
14 #include <rti/routing/adapter/detail/StreamReaderListenerForwarder.hpp>
15
16 namespace rti { namespace routing { namespace adapter {
17
18 using detail::StreamReaderListener;
19
20 }}}
21
22 #endif // RTI_ROUTING_ADAPTER_STREAM_READER_LISTENER_HPP_
```

9.16 StreamWriter.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
```

```
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_STREAM_WRITER_HPP_
12 #define RTI_ROUTING_ADAPTER_STREAM_WRITER_HPP_
13
14 #include <vector>
15
16 #include <rti/routing/UpdatableEntity.hpp>
17 #include <rti/routing/detail/ForwarderUtils.hpp>
18
19 namespace rti { namespace routing { namespace adapter {
20
36 class StreamWriter : public UpdatableEntity {
37
38 public:
39
40     typedef void* SamplePtr;
41     typedef void* InfoPtr;
42
64     virtual int write(
65             const std::vector<SamplePtr>& sample_seq,
66             const std::vector<InfoPtr>& info_seq) = 0;
67
71     virtual ~StreamWriter()
72     {
73     }
74 };
75
87 template <typename Data, typename Info>
88 class TStreamWriter : public StreamWriter {
89
90 public:
91
95     typedef Data DataRep;
96     typedef DataRep* DataRepPtr;
100     typedef Info InfoRep;
101     typedef InfoRep* InfoRepPtr;
102
107     int write(
108             const std::vector<SamplePtr>& sample_seq,
109             const std::vector<InfoPtr>& info_seq) RTI_FINAL
110     {
111         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(sample_seq_, sample_seq);
112         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(info_seq_, info_seq);
113
114         return write(sample_seq_, info_seq_);
115     }
116
122     virtual int write(
123             const std::vector<Data*>& sample_seq,
124             const std::vector<Info*>& info_seq) = 0;
125
129     virtual ~TStreamWriter()
130     {
131     }
132
133 private:
134     std::vector<Data*> sample_seq_;
135     std::vector<Info*> info_seq_;
136 };
137
138
146 typedef TStreamWriter<dds::core::xtypes::DynamicData, dds::sub::SampleInfo>
147         DynamicDataStreamWriter;
148
149 }}}
150
151 #endif // RTI_ROUTING_ADAPTER_STREAM_WRITER_HPP_
```

9.17 ForwarderUtils.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
```

```
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_DETAIL_FORWARDER_UTILS_UTILS_HPP_
12 #define RTI_ROUTING_DETAIL_FORWARDER_UTILS_UTILS_HPP_
13
14 #include <map>
15
16 #include <dds/topic/Topic.hpp>
17 #include <rti/core/Exception.hpp>
18 #include <routingservice/routingservice_infrastructure.h>
19
20 namespace rti { namespace routing { namespace detail {
21
22 #define RTI_ROUTING_THROW_ON_NULL(pointer) \
23     if ((pointer) == NULL) { \
24         throw dds::core::Error("invalid return of NULL"); \
25     }
26
27 #define RTI_ROUTING_THROW_ON_ENV_ERROR(NATIVE_ENV) \
28     if (RTI_RoutingServiceEnvironment_error_occurred((NATIVE_ENV))) { \
29             dds::core::Error rex( \
30                     RTI_RoutingServiceEnvironment_get_error_message((NATIVE_ENV))); \
31             RTI_RoutingServiceEnvironment_clear_error((NATIVE_ENV)); \
32             throw rex; \
33     }
34
35 #define RTI_ROUTING_SAMPLE_VECTOR_COPY_FROM_NATIVE( \
36         VECTOR, \
37         NATIVE_ARRAY, \
38         ARRAY_LENGTH) \
39     (VECTOR).resize((ARRAY_LENGTH)); \
40     memcpy(&((VECTOR)[0]), \
41             (NATIVE_ARRAY), \
42             sizeof (void*) * (ARRAY_LENGTH));
43
44 #define RTI_ROUTING_SAMPLE_VECTOR_COPY_FROM_NATIVE_W_OFFSET( \
45         VECTOR, \
46         NATIVE_ARRAY, \
47         OFFSET, \
48         ARRAY_LENGTH) \
49     (VECTOR).resize((ARRAY_LENGTH) + (OFFSET)); \
50     memcpy(&((VECTOR)[OFFSET]), \
51             (NATIVE_ARRAY), \
52             sizeof (void*) * (ARRAY_LENGTH));
53
54
55 #define RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS( \
56         TO_VECTOR, \
57         FROM_VECTOR) \
58     (TO_VECTOR).resize((FROM_VECTOR).size()); \
59     if ((FROM_VECTOR).size() > 0) { \
60         memcpy( \
61                 &(TO_VECTOR[0]), \
62                 &(FROM_VECTOR)[0], \
63                 sizeof(void*) * (FROM_VECTOR).size()); \
64     }
65
66
67 template <typename OwnerType, typename ForwarderType>
68 struct ScopedForwarder {
69
70     ScopedForwarder(
71             OwnerType *owner,
72             ForwarderType *forwarder,
73             RTI_RoutingServiceEnvironment *environment)
74         :owner_(owner), forwarder_(forwarder), environment_(environment)
75     {
76     }
77
78     void release()
79     {
80         forwarder_ = NULL;
81     }
82
83     ~ScopedForwarder()
```

```
84     {
85         if (forwarder_ != NULL) {
86             ForwarderType::delete_native(
87                     owner_,
88                     forwarder_->native(),
89                     environment_);
90         }
91     }
92
93 private:
94
95     OwnerType *owner_;
96     ForwarderType *forwarder_;
97     RTI_RoutingServiceEnvironment *environment_;
98
99 };
100
101
102 } } }
103
104
105 #endif // RTI_ROUTING_DETAIL_FORWARDER_UTILS_UTILS_HPP_
```

9.18 RoutingServiceImpl.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_ROUTING_SERVICE_IMPL_HPP_
12 #define RTI_ROUTING_ROUTING_SERVICE_IMPL_HPP_
13
14 #include <dds/core/refmacros.hpp>
15 #include <rti/core/detail/SelfReference.hpp>
16 #include "routingservice/routingservice_adapter.h"
17 #include "routingservice/routingservice_service.h"
18 #include "routingservice/routingservice_log.h"
19 #include "routingservice/ServiceAdmin.h"
20
21 #include <rti/routing/ServiceProperty.hpp>
22 #include <rti/routing/adapter/AdapterPlugin.hpp>
23 #include <rti/routing/transf/TransformationPlugin.hpp>
24 #include <rti/routing/processor/ProcessorPlugin.hpp>
25 #include <rti/routing/detail/ShutdownHookForwarder.hpp>
26 #include <rti/idlgen/ServiceAdmin.hpp>
27
28 namespace rti { namespace routing {
29 class RoutingServiceImpl
30         : public rti::core::detail::RetainableType<RoutingServiceImpl> {
31 public:
32
33     typedef RTI::Service::Admin::CommandRequest CommandRequest;
34     typedef RTI::Service::Admin::CommandReply CommandReply;
35
36     template <typename HookFunc>
37     explicit RoutingServiceImpl(
38             const ServiceProperty& property,
39             HookFunc& shutdown_hook)
40             : native_(RTI_RoutingService_new(&property.native())),
41             shutdown_hook_fwd_(
42                     new rti::routing::detail::ShutdownHookForwarder<HookFunc>(
43                             shutdown_hook))
44     {
45         rti::core::check_create_entity(native_, "RoutingService");
46         RTI_RoutingService_set_remote_shutdown_hook(
47                 native_,
48                 shutdown_hook_fwd_->native());
49     }
50
51     explicit RoutingServiceImpl(const ServiceProperty& property)
52             : native_(RTI_RoutingService_new(&property.native())),
```

```
53                shutdown_hook_fwd_(nullptr)
54      {
55          rti::core::check_create_entity(native_, "RoutingService");
56      }
57
58      RoutingServiceImpl(const RTI_RoutingServiceProperty& property)
59              : native_(RTI_RoutingService_new(&property)),
60              shutdown_hook_fwd_(NULL)
61      {
62          rti::core::check_create_entity(native_, "RoutingService");
63      }
64
65      RoutingServiceImpl(RTI_RoutingService *native)
66              : native_(native), shutdown_hook_fwd_(NULL)
67      {
68          rti::core::check_create_entity(native_, "RoutingService");
69      }
70
71      ~RoutingServiceImpl()
72      {
73          RTI_RoutingService_delete(native_);
74          delete shutdown_hook_fwd_;
75      }
76
77      void start()
78      {
79          if (!RTI_RoutingService_start(native_)) {
80              throw dds::core::Error("failed to start RoutingService");
81          }
82      }
83
84      void stop()
85      {
86          if (!RTI_RoutingService_stop(native_)) {
87              throw dds::core::Error("failed to stop RoutingService");
88          }
89      }
90
91      void attach_adapter_plugin(
92              rti::routing::adapter::AdapterPlugin *adapter_plugin,
93              const std::string& registered_name)
94      {
95          RTI_RoutingServiceAdapterPluginExt *native_plugin =
96                  adapter::detail::AdapterForwarder::create_plugin(adapter_plugin);
97          if (!RTI_RoutingService_attach_adapter_plugin(
98                  native_,
99                  native_plugin,
100                  registered_name.c_str())) {
101             throw dds::core::Error("failed to attach native adapter");
102         }
103     }
104
105     void attach_transformation_plugin(
106             rti::routing::transf::TransformationPlugin *transformation_plugin,
107             const std::string& registered_name)
108     {
109         RTI_RoutingServiceTransformationPlugin *native_plugin =
110                 transf::detail::TransformationPluginForwarder::create_plugin(
111                         transformation_plugin);
112         if (!RTI_RoutingService_attach_transformation_plugin(
113                 native_,
114                 native_plugin,
115                 registered_name.c_str())) {
116             throw dds::core::Error("failed to attach native transformation");
117         }
118     }
119
120     void attach_processor_plugin(
121             rti::routing::processor::ProcessorPlugin *processor_plugin,
122             const std::string& registered_name)
123     {
124         RTI_RoutingServiceProcessorPlugin *native_plugin =
125                 processor::detail::ProcessorPluginForwarder::create_plugin(
126                         processor_plugin);
127         if (!RTI_RoutingService_attach_processor_plugin(
128                 native_,
129                 native_plugin,
130                 registered_name.c_str())) {
131             throw dds::core::Error("failed to attach native processor");
132         }
133     }
```

```
134
135     CommandReply& execute_command(
136             CommandReply& reply,
137             const CommandRequest& request)
138     {
139         RTI_Service_Admin_CommandRequest native_request;
140         RTI_Service_Admin_CommandReply *native_reply_ptr = nullptr;
141
142         if (!RTI_Service_Admin_CommandRequest_initialize(&native_request)) {
143             RTI_Service_Admin_CommandRequest_finalize(&native_request);
144             throw dds::core::Error("failed to initialize CommandRequest");
145         }
146         std::shared_ptr<RTI_Service_Admin_CommandRequest> request_ptr(
147                 &native_request,
148                 [](RTI_Service_Admin_CommandRequest *native_req_ptr) {
149                     RTI_Service_Admin_CommandRequest_finalize(native_req_ptr);
150                 });
151
152         to_native(native_request, request);
153         RTI_RoutingService_execute_command(
154                 native_,
155                 &native_reply_ptr,
156                 &native_request);
157         from_native(reply, *native_reply_ptr);
158         RTI_RoutingService_return_reply(native_, native_reply_ptr);
159         return reply;
160     }
161
162     CommandReply execute_command(const CommandRequest& request)
163     {
164         CommandReply reply;
165         return execute_command(reply, request);
166     }
167
168
169     RTI_RoutingService* native() const
170     {
171         return native_;
172     }
173
174 private:
175
176     RTI_RoutingService *native_;
177
178     rti::routing::detail::AbstractShutdownHookForwarder *shutdown_hook_fwd_;
179
180     static RTI_Service_Admin_CommandRequest& to_native(
181             RTI_Service_Admin_CommandRequest& native,
182             const RTI::Service::Admin::CommandRequest& request)
183     {
184         native.action = static_cast<RTI_Service_Admin_CommandActionKind>(
185                 request.action());
186         if (request.application_name().is_set()) {
187             if (DDS_String_replace(
188                     &native.application_name,
189                     request.application_name().get().c_str()) == NULL) {
190                 throw dds::core::Error("failed to set application name");
191             }
192         }
193         native.instance_id = request.instance_id();
194         if (DDS_String_replace(
195                 &native.resource_identifier,
196                 request.resource_identifier().c_str()) == NULL) {
197             throw dds::core::Error("failed to set resource identifier");
198         }
199         if (DDS_String_replace(
200                 &native.string_body,
201                 request.string_body().c_str()) == NULL) {
202             throw dds::core::Error("failed to set string body");
203         }
204         if (!DDS_OctetSeq_from_array(
205                 &native.octet_body,
206                 (DDS_Octet *) request.octet_body().data(),
207                 (DDS_Long) request.octet_body().size())) {
208             throw dds::core::Error("failed to set octet body");
209         }
210         return native;
211     }
212
213     static RTI::Service::Admin::CommandReply& from_native(
214             RTI::Service::Admin::CommandReply& reply,
```

```
215             const RTI_Service_Admin_CommandReply& native)
216     {
217         using namespace RTI::Service::Admin;
218 
219         reply.retcode(static_cast<CommandReplyRetcode>(native.retcode));
220         reply.native_retcode(native.native_retcode);
221         DDS_Octet *native_octet_body = DDS_OctetSeq_get_contiguous_buffer(
222                 (DDS_OctetSeq *) &native.octet_body);
223         if (DDS_OctetSeq_get_length(&native.octet_body) < 0) {
224             throw dds::core::Error("invalid octet_body length");
225         }
226         reply.octet_body().resize(
227                 (size_t) DDS_OctetSeq_get_length(&native.octet_body));
228         std::copy(
229                 native_octet_body,
230                 native_octet_body + DDS_OctetSeq_get_length(&native.octet_body),
231                 reply.octet_body().begin());
232         reply.string_body(native.string_body);
233 
234         return reply;
235     }
236 
237 };
238 
239 }}
240 
241 #endif // RTI_ROUTINGv_ROUTING_SERVICE_IMPL_HPP_
```

9.19 ShutdownHookForwarder.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10 
11 #ifndef RTI_ROUTING_SHUTDOWN_HOOK_HPP_
12 #define RTI_ROUTING_SHUTDOWN_HOOK_HPP_
13 
14 
15 #include "routingservice/routingservice_service.h"
16 
17 namespace rti { namespace routing { namespace detail {
18 
19 class AbstractShutdownHookForwarder {
20 public:
21 
22     AbstractShutdownHookForwarder()
23     {
24         native_.on_shutdown = on_shutdown_forwarder;
25         native_.shutdown_hook_data = this;
26     }
27 
28     virtual ~AbstractShutdownHookForwarder()
29     {
30     }
31 
32     virtual void on_shutdown()
33     {
34     }
35 
36     static void on_shutdown_forwarder(void *hook_data)
37     {
38         AbstractShutdownHookForwarder *self =
39                 static_cast<AbstractShutdownHookForwarder*>(hook_data);
40         try {
41             self->on_shutdown();
42         } catch(...) {}
43     }
44 
45 
46     const RTI_RoutingServiceRemoteShutdownHook* native() const
47     {
```

```
48         return &native_;
49     }
50
51 private:
52     RTI_RoutingServiceRemoteShutdownHook native_;
53 };
54
55 template<typename HookFunc>
56 class ShutdownHookForwarder : public AbstractShutdownHookForwarder {
57 public:
58     ShutdownHookForwarder(HookFunc& shutdown_hook)
59         : shutdown_hook_(shutdown_hook)
60     {
61
62     }
63
64     void on_shutdown()
65     {
66        shutdown_hook_();
67     };
68
69 private:
70     HookFunc& shutdown_hook_;
71 };
72
73 }
74
75 } }
76
77
78 #endif /* RTI_ROUTING_SHUTDOWN_HOOK_HPP_ */
79
```

9.20 UpdatableEntityForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_DETAIL_UPDATABLE_ENTITY_FORWARDER_HPP_
12 #define RTI_ROUTING_DETAIL_UPDATABLE_ENTITY_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15 #include <rti/routing/UpdatableEntity.hpp>
16 #include <rti/routing/ServiceProperty.hpp>
17 #include <rti/routing/detail/ForwarderUtils.hpp>
18
19 namespace rti { namespace routing { namespace detail {
20
21
22 class UpdatableEntityForwarder {
23 public:
24
25     static void update(
26             UpdatableEntity *updatable_entity,
27             const struct RTI_RoutingServiceProperties *native_properties,
28             RTI_RoutingServiceEnvironment *environment)
29     {
30
31         try {
32             // Set properties
33             PropertySet properties;
34             rti::routing::PropertyAdapter::add_properties_from_native(
35                     properties,
36                     native_properties);
37             updatable_entity->update(properties);
38         } catch (const std::exception& ex) {
39             RTI_RoutingServiceEnvironment_set_error(
40                     environment,
41                     "%s",
42                     ex.what());
```

```
43         } catch (...) {
44             RTI_RoutingServiceEnvironment_set_error(
45                     environment,
46                     "unexpected exception");
47             return;
48         }
49     }
50
51 };
52
53 }}}
54
55 #endif // RTI_ROUTING_DETAIL_UPDATABLE_ENTITY_FORWARDER_HPP_
```

9.21 EntityListener.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ENTITY_LISTENER_HPP_
12 #define RTI_ROUTING_ENTITY_LISTENER_HPP_
13
14 #include "routingservice/routingservice_service_impl.h"
15
16 #include <rti/apputils/ServiceException.hpp>
17 #include <rti/apputils/log/LogConfig.hpp>
18 #include <rti/routing/StreamInfo.hpp>
19
20 namespace rti { namespace routing {
21
24 struct EntityKind_def {
28     enum type {
29         SERVICE = RTI_ROUTING_SERVICE_SERVICE_ENTITY,
30         DOMAIN_ROUTE = RTI_ROUTING_SERVICE_DOMAIN_ROUTE_ENTITY,
31         CONNECTION = RTI_ROUTING_SERVICE_CONNECTION_ENTITY,
32         SESSION = RTI_ROUTING_SERVICE_SESSION_ENTITY,
33         AUTO_TOPIC_ROUTE = RTI_ROUTING_SERVICE_AUTO_TOPIC_ROUTE_ENTITY,
34         TOPIC_ROUTE = RTI_ROUTING_SERVICE_TOPIC_ROUTE_ENTITY,
35         INPUT = RTI_ROUTING_SERVICE_INPUT_ENTITY,
36         OUTPUT = RTI_ROUTING_SERVICE_OUTPUT_ENTITY,
37         count_
38     };
39
40 };
41
44 typedef dds::core::safe_enum<EntityKind_def> EntityKind;
45
46
49 struct EntityStateKind_def {
53     enum type {
54         INVALID = RTI_SERVICE_INVALID_ENTITY_STATE,
55         ENABLED = RTI_SERVICE_ENABLED_ENTITY_STATE,
56         DISABLED = RTI_SERVICE_DISABLED_ENTITY_STATE,
57         STARTED = RTI_SERVICE_STARTED_ENTITY_STATE,
58         STOPPED = RTI_SERVICE_STOPPED_ENTITY_STATE,
59         RUNNING = RTI_SERVICE_RUNNING_ENTITY_STATE,
60         PAUSED = RTI_SERVICE_PAUSED_ENTITY_STATE
61     };
62
63 };
64
65 typedef dds::core::safe_enum<EntityStateKind_def> EntityStateKind;
66
67 /*
68  * @brief string representation of EntityStateKind
69  */
70 inline
71 const std::string to_string(const rti::routing::EntityStateKind& kind)
72 {
73     static const char * ENTITY_STATE_KIND_AS_STRING[] = {
```

```
74          "INVALID",
75          "ENABLED",
76          "DISABLED",
77          "STARTED",
78          "STOPPED",
79          "RUNNING",
80          "PAUSED"
81      };
82
83      return ENTITY_STATE_KIND_AS_STRING[kind.underlying()];
84  }
85
86  inline
87  const char* to_action_native_string(const rti::routing::EntityStateKind& kind)
88  {
89      static const char * ENTITY_STATE_ACTION_AS_STRING[] = {
90          "INVALID",
91          "ENABLE",
92          "DISABLE",
93          "START",
94          "STOP",
95          "RUN",
96          "PAUSE"
97          ""
98      };
99
100      return ENTITY_STATE_ACTION_AS_STRING[kind.underlying()];
101 }
102
103 inline
104 const std::string to_action_string(const rti::routing::EntityStateKind& kind)
105 {
106     return to_action_native_string(kind);
107 }
108
109 class EntityListener {
110 public:
111
112     class Context {
113     public:
114
115         friend EntityListener::Context from_native(
116                 RTI_RoutingServiceEntityListenerContext*);
117
118         const rti::routing::EntityKind& entity_kind() const
119         {
120             return entity_kind_;
121         }
122
123         const std::string& entity_name() const
124         {
125             return entity_name_;
126         }
127
128         const rti::routing::EntityStateKind& entity_state_kind() const
129         {
130             return entity_state_kind_;
131         }
132
133         bool ok()
134         {
135             return ok_;
136         }
137
138         rti::advlog::Entries logging_context_entries()
139         {
140             return logging_context_entries_;
141         }
142
143     private:
144
145         rti::routing::EntityKind entity_kind_;
146
147         std::string entity_name_;
148
149         rti::routing::EntityStateKind entity_state_kind_;
150
151         bool ok_;
152
153         /*
154          * We need to keep the entries of the context because they are entered
```

```
155          * in a function and left in another function.
156          * The number of entries is two:
157          *  - An entry for the resource.
158          *  - An entry for the activity.
159          *
160          * Recorder:
161          *  - The entries are entered into the context using the macro
162          *   RTI_ENTITY_LISTENER_CONTEXT_ENTER_ON_BEFORE in a function.
163          *  - The entries are left using the macro
164          *   RTI_ENTITY_LISTENER_CONTEXT_LEAVE_ON_AFTER in another function.
165          *
166          * For example:
167          *      - Enter in EntityListener::on_before_create
168          *      - Leave in EntityListener::on_after_create
169          */
170         rti::advlog::Entries logging_context_entries_;
171     };
172
173     virtual void on_before_create( EntityListener::Context&)
174     {
175     }
176
177     virtual void on_after_create(EntityListener::Context&)
178     {
179     }
180
181     virtual void on_before_state_change(EntityListener::Context&)
182     {
183     }
184
185     virtual void on_after_state_change(EntityListener::Context&)
186     {
187     }
188
189     virtual void on_before_delete(EntityListener::Context&)
190     {
191     }
192
193     virtual void on_after_delete(EntityListener::Context&)
194     {
195     }
196
197     virtual void on_before_stream_event(
198             EntityListener::Context&,
199             const rti::routing::StreamInfo&)
200     {
201     }
202
203     virtual void on_after_stream_event(
204             EntityListener::Context&,
205             const rti::routing::StreamInfo&)
206     {
207     }
208
209     virtual ~EntityListener() {}
210
211 };
212
213     inline
214     EntityListener::Context from_native(
215             RTI_RoutingServiceEntityListenerContext *native)
216     {
217         EntityListener::Context context;
218         context.entity_kind_ = static_cast<EntityKind::type> (
219                 native->entity_kind);
220         context.entity_name_ = native->entity_name;
221         context.entity_state_kind_ = static_cast<EntityStateKind::type> (
222                 native->entity_state_kind);
223         context.ok_ = native->ok ? true : false;
224         context.logging_context_entries_.from_native(
225                 native->logging_context_entries,
226                 native->logging_context_params);
227
228         return context;
229     }
230
231 class EntityListenerForwarder  {
232 public:
233
234     typedef RTI_RoutingServiceEntityListener native;
235
```

```
236     static native create(
237             EntityListener *listener)
238     {
239         native native_listener = RTI_RoutingServiceEntityListener_INITIALIZER;
240         native_listener.on_before_create =
241                 on_before_create_forwarder;
242         native_listener.on_after_create =
243                 on_after_create_forwarder;
244         native_listener.on_before_state_change =
245                 on_before_state_change_forwarder;
246         native_listener.on_after_state_change =
247                 on_after_state_change_forwarder;
248         native_listener.on_before_delete =
249                 on_before_delete_forwarder;
250         native_listener.on_after_delete =
251                 on_after_delete_forwarder;
252         native_listener.on_before_stream_event =
253                 on_before_stream_event_forwarder;
254         native_listener.on_after_stream_event =
255                 on_after_stream_event_forwarder;
256         native_listener.listener_data = static_cast<void*>(listener);
257
258         return native_listener;
259     }
260
261 private:
262     static void on_before_create_forwarder(
263             void *listener_data,
264             RTI_RoutingServiceEntityListenerContext *context_native)
265     {
266         EntityListener *listener =
267                 static_cast<EntityListener *>(listener_data);
268         try {
269             EntityListener::Context context =
270                     rti::routing::from_native(context_native);
271             listener->on_before_create(context);
272         } catch (rti::apputils::ServiceException& ex) {
273             SERVICELog_fromException(ex);
274         } catch (std::exception& ex) {
275             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
276         } catch (...) {
277             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
278         }
279     }
280     static void on_after_create_forwarder(
281             void *listener_data,
282             RTI_RoutingServiceEntityListenerContext *context_native)
283     {
284         EntityListener *listener =
285                 static_cast<EntityListener *>(listener_data);
286         try {
287             EntityListener::Context context =
288                     rti::routing::from_native(context_native);
289             listener->on_after_create(context);
290         } catch (rti::apputils::ServiceException& ex) {
291             SERVICELog_fromException(ex);
292         } catch (std::exception& ex) {
293             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
294         } catch (...) {
295             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
296         }
297     }
298
299     static void on_before_state_change_forwarder(
300             void *listener_data,
301             RTI_RoutingServiceEntityListenerContext *context_native)
302     {
303         EntityListener *listener =
304                 static_cast<EntityListener *>(listener_data);
305         try {
306             EntityListener::Context context =
307                     rti::routing::from_native(context_native);
308             listener->on_before_state_change(context);
309         } catch (rti::apputils::ServiceException& ex) {
310             SERVICELog_fromException(ex);
311         } catch (std::exception& ex) {
312             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
313         } catch (...) {
314             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
315         }
316     }
```

```
317
318     static void on_after_state_change_forwarder(
319             void *listener_data,
320             RTI_RoutingServiceEntityListenerContext *context_native)
321     {
322         EntityListener *listener =
323                 static_cast<EntityListener *>(listener_data);
324         try {
325             EntityListener::Context context =
326                     rti::routing::from_native(context_native);
327             listener->on_after_state_change(context);
328         } catch (rti::apputils::ServiceException& ex) {
329             SERVICELog_fromException(ex);
330         } catch (std::exception& ex) {
331             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
332         } catch (...) {
333             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
334         }
335     }
336
337     static void on_before_delete_forwarder(
338             void *listener_data,
339             RTI_RoutingServiceEntityListenerContext *context_native)
340     {
341         EntityListener *listener =
342                 static_cast<EntityListener *>(listener_data);
343         try {
344             EntityListener::Context context =
345                     rti::routing::from_native(context_native);
346             listener->on_before_delete(context);
347         } catch (rti::apputils::ServiceException& ex) {
348             SERVICELog_fromException(ex);
349         } catch (std::exception& ex) {
350             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
351         } catch (...) {
352             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
353         }
354     }
355
356     static void on_after_delete_forwarder(
357             void *listener_data,
358             RTI_RoutingServiceEntityListenerContext *context_native)
359     {
360         EntityListener *listener =
361                 static_cast<EntityListener *>(listener_data);
362         try {
363             EntityListener::Context context =
364                     rti::routing::from_native(context_native);
365             listener->on_after_delete(context);
366         } catch (rti::apputils::ServiceException& ex) {
367             SERVICELog_fromException(ex);
368         } catch (std::exception& ex) {
369             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
370         } catch (...) {
371             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
372         }
373     }
374
375     static void on_before_stream_event_forwarder(
376             void *listener_data,
377             RTI_RoutingServiceEntityListenerContext *context_native,
378             const RTI_RoutingServiceStreamInfo *stream_info)
379     {
380         EntityListener *listener =
381                 static_cast<EntityListener *>(listener_data);
382         try {
383             EntityListener::Context context =
384                     rti::routing::from_native(context_native);
385             listener->on_before_stream_event(
386                     context,
387                     *stream_info);
388         } catch (rti::apputils::ServiceException& ex) {
389             SERVICELog_fromException(ex);
390         } catch (std::exception& ex) {
391             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
392         } catch (...) {
393             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
394         }
395     }
396
397     static void on_after_stream_event_forwarder(
```

```
398             void *listener_data,
399             RTI_RoutingServiceEntityListenerContext *context_native,
400             const RTI_RoutingServiceStreamInfo *stream_info)
401     {
402         EntityListener *listener =
403                 static_cast<EntityListener *>(listener_data);
404         try {
405             EntityListener::Context context =
406                     rti::routing::from_native(context_native);
407             listener->on_after_stream_event(
408                     context,
409                     *stream_info);
410         } catch (rti::apputils::ServiceException& ex) {
411             SERVICELog_fromException(ex);
412         } catch (std::exception& ex) {
413             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
414         } catch (...) {
415             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
416         }
417     }
418 };
419
420 }}
421
422
423
424
425
426 #endif // RTI_ROUTING_ENTITY_LISTENER_HPP_
```

9.22 LogConfig.hpp

```
1  /*
2   (c) Copyright, Real-Time Innovations, 2017-
3   All rights reserved.
4
5   No duplications, whole or partial, manual or electronic, may be made without
6   express written permission.  Any such copies, or revisions thereof, must
7   display this notice unaltered.
8
9   This code contains trade secrets of Real-Time Innovations, Inc.
10  */
11
12  #ifndef HPP_ROUTING_LOG_LOG_CONFIG_HPP_
13  #define HPP_ROUTING_LOG_LOG_CONFIG_HPP_
14
15  #include <cstdio>
16  #include <cstdarg>
17
18  #include "log/log_common.h"
19  #include "routingservice/routingservice_log.h"
20
21  #include "ndds/ndds_config_c.h"
22
23
24  namespace rti { namespace routing { namespace log {
25
26  class LogConfig {
27  public:
28      static rti::routing::log::LogConfig& instance()
29      {
30          static rti::routing::log::LogConfig singleton;
31          return singleton;
32      }
33
34      // Deleted constructor to avoid multiple instances.
35      LogConfig(const LogConfig&) = delete;
36      LogConfig(LogConfig&&) = delete;
37
38      RTILogBitmap instrumentation_mask() const
39      {
40          NDDS_Config_LogVerbosity verbosity =
41                  NDDS_Config_Logger_get_verbosity_by_service(
42                          NDDS_Config_Logger_get_instance(),
43                          NDDS_CONFIG_LOG_SERVICE_ROUTING);
44          return (RTILogBitmap) verbosity;
45      }
```

```
46
47      LogConfig& instrumentation_mask(RTILogBitmap logmask)
48      {
49          NDDS_Config_Logger_set_verbosity_by_service(
50                  NDDS_Config_Logger_get_instance(),
51                  NDDS_CONFIG_LOG_SERVICE_ROUTING,
52                  (NDDS_Config_LogVerbosity) logmask);
53          return *this;
54      }
55
56      void router_instrumentation_mask(RTILogBitmap logmask)
57      {
58          instrumentation_mask(logmask);
59      }
60
61      void update_native(RTILogBitmap logmask)
62      {
63          router_instrumentation_mask(logmask);
64      }
65
66  private:
67      // Constructor is private: this class is a singleton
68      LogConfig() = default;
69  };
70
71  }}} // rti::routing::log
72
73  #endif /* HPP_ROUTING_LOG_LOG_CONFIG_HPP_ */
```

9.23 Logger.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11  #ifndef RTI_ROUTING_LOGGER_HPP_
12  #define RTI_ROUTING_LOGGER_HPP_
13
14  #include <dds/core/Reference.hpp>
15  #include <rti/config/Logger.hpp>
16
17  #include "routingservice/routingservice_log.h"
18  #include "routingservice/routingservice_service.h"
19
20  namespace rti { namespace routing {
21
22
30  class Logger
31  {
32  public:
33
34      typedef rti::config::Verbosity Verbosity;
35      typedef rti::config::LogCategory LogCategory;
36      typedef rti::config::PrintFormat PrintFormat;
37
38      static Logger& instance()
39      {
40          static Logger singleton;
41          return singleton;
42      }
43
51      void service_verbosity(rti::config::Verbosity verbosity)
52      {
53          NDDS_Config_Logger_set_verbosity_by_service(
54                  NDDS_Config_Logger_get_instance(),
55                  NDDS_CONFIG_LOG_SERVICE_ROUTING,
56                  static_cast<NDDS_Config_LogVerbosity>(verbosity.underlying()));
57      }
58
64       rti::config::Verbosity service_verbosity()
65      {
```

```
66          RTILogBitmap instrumentation_mask =
67                  NDDS_Config_Logger_get_verbosity_by_service(
68                          NDDS_Config_Logger_get_instance(),
69                          NDDS_CONFIG_LOG_SERVICE_ROUTING);
70
71          return static_cast<rti::config::Verbosity::type>(instrumentation_mask);
72      }
73
86      void error(const std::string& msg)
87      {
88          this->error(msg.c_str());
89      }
90
94      void error(const char *msg)
95      {
96          this->message(rti::config::LogLevel::EXCEPTION, msg);
97      }
98
110     void warn(const std::string& msg)
111     {
112         this->warn(msg.c_str());
113     }
114
118     void warn(const char *msg)
119     {
120         this->message(rti::config::LogLevel::WARNING, msg);
121     }
122
134     void local(const std::string& msg)
135     {
136         this->local(msg.c_str());
137     }
138
142     void local(const char *msg)
143     {
144         this->message(rti::config::LogLevel::STATUS_LOCAL, msg);
145     }
146
158     void remote(const std::string& msg)
159     {
160         this->remote(msg.c_str());
161     }
162
166     void remote(const char *msg)
167     {
168         this->message(rti::config::LogLevel::STATUS_REMOTE, msg);
169     }
170
182     void debug(const std::string& msg)
183     {
184         this->debug(msg.c_str());
185     }
186
190     void debug(const char *msg)
191     {
192         this->message(rti::config::LogLevel::STATUS_ALL, msg);
193     }
194
195     virtual ~Logger()
196     {
197     }
198
199 private:
200
201     void message(const rti::config::LogLevel& level, const char* msg)
202     {
203         RTI_RoutingServiceLogger_log(
204                 static_cast<NDDS_Config_LogLevel> (level.underlying()),
205                 "%s",
206                 msg);
207     }
208
209     Logger()
210     {
211     }
212      // Disable copy
213     Logger(const Logger&);
214     Logger& operator=(const Logger&);
215 };
216
217 }}
```

```
218
219 #endif // RTI_ROUTING_LOGGER_HPP_
```

9.24 ProcessorForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_PROCESSOR_DETAIL_PROCESSOR_FORWARDER_HPP_
12 #define RTI_ROUTING_PROCESSOR_DETAIL_PROCESSOR_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include "routingservice/routingservice_processor_impl.h"
17 #include <rti/routing/detail/ForwarderUtils.hpp>
18 #include <rti/routing/detail/UpdatableEntityForwarder.hpp>
19 #include <rti/routing/processor/Route.hpp>
20 #include <rti/routing/processor/Input.hpp>
21 #include <rti/routing/processor/Output.hpp>
22
23 namespace rti { namespace routing { namespace processor { namespace detail {
24
25 class ProcessorForwarder  {
26
27 private:
28
29     template <typename PORT, typename NATIVE>
30     class ScopedPort {
31
32     public:
33         ScopedPort(Route& route,  int32_t index, NATIVE *native)
34             : route_(route),
35               port_(new PORT(native, index, route.native_route_))
36         {
37         }
38
39         PORT* get()
40         {
41             return port_;
42         }
43
44         void clear()
45         {
46             port_ = NULL;
47         }
48
49         ~ScopedPort()
50         {
51             if (port_ != NULL) {
52                 delete port_;
53                 port_ = NULL;
54             }
55         }
56
57     private:
58         Route& route_;
59         PORT *port_;
60     };
61
62 public:
63
64     static RTI_RoutingServiceProcessor * create_native(
65             ProcessorPlugin *plugin,
66             RTI_RoutingServiceRoute *native_route,
67             const struct RTI_RoutingServiceProperties *native_properties,
68             RTI_RoutingServiceEnvironment *environment)
69     {
70         try {
71             using rti::routing::detail::ScopedForwarder;
72
```

```
73              std::map<std::string, std::string> properties;
74              rti::routing::PropertyAdapter::add_properties_from_native(
75                      properties,
76                      native_properties);
77
78              ProcessorForwarder *forwarder = new ProcessorForwarder(
79                      native_route,
80                      environment);
81              ScopedForwarder<ProcessorPlugin, ProcessorForwarder> scoped(
82                      plugin,
83                      forwarder,
84                      environment);
85              forwarder->processor_ = plugin->create_processor(
86                      forwarder->route(),
87                      properties);
88              RTI_ROUTING_THROW_ON_NULL(forwarder->processor_);
89
90              scoped.release();
91
92              // TRUST-134: the forwarder is not leaked here. It might be leaked
93              // if delete_native() were not to be called, but Routing Service
94              // will make sure that is not the case
95
96              /* coverity[leaked_storage : FALSE] */
97              return forwarder->native();
98          } catch(const std::exception& ex) {
99              RTI_RoutingServiceEnvironment_set_error(
100                     environment,
101                     "%s",
102                     ex.what());
103             return NULL;
104         } catch (...) {
105             RTI_RoutingServiceEnvironment_set_error(
106                     environment,
107                     "unexpected exception");
108             return NULL;
109         }
110
111
112     }
113
114     static void delete_native(
115             ProcessorPlugin *plugin,
116             RTI_RoutingServiceProcessor *native_processor,
117             RTI_RoutingServiceEnvironment *environment)
118     {
119         ProcessorForwarder *processor_forwarder =
120                 static_cast<ProcessorForwarder*>(native_processor->processor_data);
121         try {
122             if (processor_forwarder->processor_ != NULL) {
123                 plugin->delete_processor(
124                         processor_forwarder->route_,
125                         processor_forwarder->processor_);
126                 processor_forwarder->processor_ = NULL;
127             }
128         } catch(const std::exception& ex) {
129             RTI_RoutingServiceEnvironment_set_error(
130                     environment,
131                     "%s",
132                     ex.what());
133         } catch (...) {
134             RTI_RoutingServiceEnvironment_set_error(
135                     environment,
136                     "unexpected exception");
137         }
138
139         delete processor_forwarder;
140     }
141
142
143     static void forward_on_route_event(
144         void *native_processor_data,
145         RTI_RoutingServiceRouteEvent *native_route_event,
146         RTI_RoutingServiceEnvironment *environment)
147     {
148
149         ProcessorForwarder *forwarder =
150                 static_cast<ProcessorForwarder*>(native_processor_data);
151
152         try {
153
```

```
154             // build up wrapper objects based on the event
155             switch (RTI_RoutingServiceRouteEvent_get_kind(native_route_event)) {
156
157             case RTI_ROUTING_SERVICE_ROUTE_EVENT_DATA_ON_INPUTS:
158                 forwarder->processor_->on_data_available(forwarder->route());
159                 break;
160
161             case RTI_ROUTING_SERVICE_ROUTE_EVENT_PERIODIC_ACTION:
162
163                 forwarder->processor_->on_periodic_action(forwarder->route());
164                 break;
165
166             case RTI_ROUTING_SERVICE_ROUTE_EVENT_INPUT_ENABLED:
167             {
168                 void *affected_entity =
169                         RTI_RoutingServiceRouteEvent_get_affected_entity(native_route_event);
170                 void *index =
171                         RTI_RoutingServiceRouteEvent_get_event_data(native_route_event);
172                 /* ScopedPort helps to delete the Input if an exception occurs */
173                 ScopedPort<Input, RTI_RoutingServiceInput> port(
174                         forwarder->route_,
175                         *(static_cast<int32_t*>(index)),
176                         static_cast<RTI_RoutingServiceInput *>(affected_entity));
177                 forwarder->processor_->on_input_enabled(
178                         forwarder->route(),
179                         *port.get());
180                 RTI_RoutingServiceInput_set_user_data(
181                         static_cast<RTI_RoutingServiceInput *>(affected_entity),
182                         port.get());
183                 /*
184                  * Input successfully enabled, so clear the ScopedPort so it
185                  * doesn't delete it
186                  */
187                 port.clear();
188
189             }
190                 break;
191
192             case RTI_ROUTING_SERVICE_ROUTE_EVENT_INPUT_DISABLED:
193             {
194
195                 void *affected_entity =
196                         RTI_RoutingServiceRouteEvent_get_affected_entity(native_route_event);
197                 RTI_RoutingServiceInput *native_input =
198                         static_cast<RTI_RoutingServiceInput *>(affected_entity);
199                 Input *input = reinterpret_cast<Input*>(
200                         RTI_RoutingServiceInput_get_user_data(native_input));
201                 forwarder->processor_->on_input_disabled(
202                         forwarder->route(),
203                         *input);
204                 RTI_RoutingServiceInput_set_user_data(native_input, NULL);
205                 delete input;
206             }
207                 break;
208
209             case RTI_ROUTING_SERVICE_ROUTE_EVENT_OUTPUT_ENABLED:
210             {
211                 void *affected_entity =
212                         RTI_RoutingServiceRouteEvent_get_affected_entity(native_route_event);
213                 void *index =
214                         RTI_RoutingServiceRouteEvent_get_event_data(native_route_event);
215                 /* ScopedPort helps to delete the Output if an exception occurs */
216                 ScopedPort<Output, RTI_RoutingServiceOutput> port(
217                         forwarder->route_,
218                         *(static_cast<int32_t*>(index)),
219                         static_cast<RTI_RoutingServiceOutput *>(affected_entity));
220                 forwarder->processor_->on_output_enabled(
221                         forwarder->route(),
222                         *port.get());
223                 RTI_RoutingServiceOutput_set_user_data(
224                         static_cast<RTI_RoutingServiceOutput *>(affected_entity),
225                         port.get());
226                 /*
227                  * Output successfully enabled, so clear the ScopedPort so it
228                  * doesn't delete it
229                  */
230                 port.clear();
231
232             }
233                 break;
234
```

```
235             case RTI_ROUTING_SERVICE_ROUTE_EVENT_OUTPUT_DISABLED:
236             {
237                 void *affected_entity =
238                         RTI_RoutingServiceRouteEvent_get_affected_entity(native_route_event);
239                 RTI_RoutingServiceOutput *native_output =
240                         static_cast<RTI_RoutingServiceOutput *>(affected_entity);
241                 Output *output = reinterpret_cast<Output*>(
242                         RTI_RoutingServiceOutput_get_user_data(native_output));
243                 forwarder->processor_->on_output_disabled(
244                         forwarder->route(),
245                         *output);
246                 RTI_RoutingServiceOutput_set_user_data(native_output, NULL);
247                 delete output;
248             }
249                 break;
250
251             case RTI_ROUTING_SERVICE_ROUTE_EVENT_ROUTE_STARTED:
252
253                 forwarder->processor_->on_start(forwarder->route());
254                 break;
255
256             case RTI_ROUTING_SERVICE_ROUTE_EVENT_ROUTE_STOPPED:
257
258                 forwarder->processor_->on_stop(forwarder->route());
259                 break;
260
261             case RTI_ROUTING_SERVICE_ROUTE_EVENT_ROUTE_RUNNING:
262
263                 forwarder->processor_->on_run(forwarder->route());
264                 break;
265
266             case RTI_ROUTING_SERVICE_ROUTE_EVENT_ROUTE_PAUSED:
267
268                 forwarder->processor_->on_pause(forwarder->route());
269                 break;
270
271             default:
272                 break;
273             };
274
275         } catch (const std::exception& ex) {
276             RTI_RoutingServiceEnvironment_set_error(
277                     environment,
278                     "%s",
279                     ex.what());
280         } catch (...) {
281             RTI_RoutingServiceEnvironment_set_error(
282                     environment,
283                     "%s",
284                     "unexpected exception");
285         }
286     }
287
288     static void forward_update(
289             void *native_processor_data,
290             const struct RTI_RoutingServiceProperties *native_properties,
291             RTI_RoutingServiceEnvironment *environment)
292     {
293
294         ProcessorForwarder *processorForwarder =
295                 static_cast<ProcessorForwarder*> (native_processor_data);
296
297         rti::routing::detail::UpdatableEntityForwarder::update(
298                 processorForwarder->processor_,
299                 native_properties,
300                 environment);
301     }
302
303     RTI_RoutingServiceProcessor* native()
304     {
305         return &native_;
306     }
307
308 private:
309
310     ProcessorForwarder(
311             RTI_RoutingServiceRoute *native_route,
312             RTI_RoutingServiceEnvironment *native_env) :
313             processor_(NULL),
314             route_(native_route, native_env)
315     {
```

```
316         RTIOsapiMemory_zero(&native_, sizeof(native_));
317         /* initialize native implementation */
318         native_.processor_data = static_cast<void *>(this);
319         native_.on_route_event = ProcessorForwarder::forward_on_route_event;
320         native_.update = ProcessorForwarder::forward_update;
321     }
322
323     rti::routing::processor::Route& route()
324     {
325         return route_;
326     }
327
328     ~ProcessorForwarder()
329     {
330     }
331
332 private:
333     RTI_RoutingServiceProcessor native_;
334     Processor *processor_;
335     rti::routing::processor::Route route_;
336
337 };
338
339 }}}}
340
341 #endif // RTI_ROUTING_PROCESSOR_DETAIL_PROCESSOR_FORWARDER_HPP_
```

9.25 ProcessorPluginForwarder.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_PROCESSOR_PLUGIN_DETAIL_PROCESSOR_PLUGIN_FORWARDER_HPP_
12 #define RTI_ROUTING_PROCESSOR_PLUGIN_DETAIL_PROCESSOR_PLUGIN_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <rti/routing/processor/detail/ProcessorForwarder.hpp>
17 #include <rti/routing/detail/ForwarderUtils.hpp>
18
19 namespace rti { namespace routing { namespace processor { namespace detail {
20
21
22 class ProcessorPluginForwarder {
23 public:
24
25     static RTI_RoutingServiceProcessorPlugin * create_plugin(
26             ProcessorPlugin *plugin)
27     {
28         RTI_RoutingServiceProcessorPlugin *native_plugin = NULL;
29         RTIOsapiHeap_allocateStructure(
30                 &native_plugin,
31                 struct RTI_RoutingServiceProcessorPlugin);
32         rti::core::check_create_entity(
33                 native_plugin,
34                 "RTI_RoutingServiceProcessorPlugin");
35         RTI_RoutingServiceProcessorPlugin_initialize(native_plugin);
36
37         // Set adapter version
38         rti::config::LibraryVersion version =  plugin->get_version();
39         native_plugin->plugin_version.major = version.major_version();
40         native_plugin->plugin_version.minor = version.minor_version();
41         native_plugin->plugin_version.release = version.release_version();
42
43         // Initialize native implementation
44         native_plugin->processor_plugin_data =
45                 static_cast<void *>(plugin);
46         native_plugin->plugin_delete =
47                 ProcessorPluginForwarder::delete_plugin;
48         native_plugin->create_processor =
```

```
49                 ProcessorPluginForwarder::forward_create_processor;
50         native_plugin->delete_processor =
51                 ProcessorPluginForwarder::forward_delete_processor;
52 
53         return native_plugin;
54     }
55 
56     static void delete_plugin(
57         RTI_RoutingServiceProcessorPlugin *native_plugin,
58         RTI_RoutingServiceEnvironment *)
59     {
60         ProcessorPlugin *plugin = static_cast<ProcessorPlugin*>(
61                 native_plugin->processor_plugin_data);
62         // Plug-in is destructor not allowed to throw
63         delete plugin;
64         RTIOsapiHeap_freeStructure(native_plugin);
65     }
66 
67 
68     static RTI_RoutingServiceProcessor * forward_create_processor(
69         void *native_plugin_data,
70         RTI_RoutingServiceRoute *native_route,
71         const struct RTI_RoutingServiceProperties *native_properties,
72         RTI_RoutingServiceEnvironment *environment)
73     {
74         return ProcessorForwarder::create_native(
75                 static_cast<ProcessorPlugin *>(native_plugin_data),
76                 native_route,
77                 native_properties,
78                 environment);
79 
80     }
81 
82     static void forward_delete_processor(
83             void *native_plugin_data,
84             struct RTI_RoutingServiceProcessor *native_processor,
85             RTI_RoutingServiceRoute *,
86             RTI_RoutingServiceEnvironment *environment)
87     {
88 
89         ProcessorForwarder::delete_native(
90                 static_cast<ProcessorPlugin *>(native_plugin_data),
91                 native_processor,
92                 environment);
93     }
94 
95 };
96 
97 }}}}
98 
99 #endif // RTI_ROUTING_PROCESSOR_PLUGIN_DETAIL_PROCESSOR_PLUGIN_FORWARDER_HPP_
```

9.26 Input.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10 
11 #ifndef RTI_ROUTING_PROCESSOR_INPUT_HPP_
12 #define RTI_ROUTING_PROCESSOR_INPUT_HPP_
13 
14 #include <dds/core/Value.hpp>
15 #include <dds/core/SafeEnumeration.hpp>
16 #include <rti/core/NativeValueType.hpp>
17 #include <rti/core/Entity.hpp>
18 #include <dds/sub/DataReader.hpp>
19 #include <dds/core/xtypes/DynamicData.hpp>
20 
21 #include "routingservice/routingservice_processor.h"
22 #include "routingservice/routingservice_adapter_new.h"
23 #include <rti/routing/StreamInfo.hpp>
```

```
24 #include <rti/routing/adapter/StreamReader.hpp>
25 #include <rti/routing/processor/LoanedSamples.hpp>
26 #include <rti/routing/processor/Query.hpp>
27
28 namespace rti { namespace routing { namespace processor {
29
30 class Route;
31
32 template <typename Data, typename Info> class TypedInput;
33
34 template <typename Data, typename Info = dds::sub::SampleInfo>
35 class TypedInput;
36
49 class Input {
50 public:
51
52     /*i
53      */
54     Input(RTI_RoutingServiceInput *native,
55             int32_t index,
56             RTI_RoutingServiceRoute *native_route)
57         : native_(native),
58           index_(index),
59           native_route_(native_route),
60           stream_info_("", ""),
61           name_(RTI_RoutingServiceInput_get_name(native))
62     {
63         const RTI_RoutingServiceStreamInfo *native_stream_info =
64                 RTI_RoutingServiceInput_get_stream_info(native);
65         if (native_stream_info == nullptr) {
66             throw dds::core::InvalidArgumentError(
67                     "invalid argument: invalid StreamInfo for input");
68         }
69         stream_info_ = *native_stream_info;
70     }
71
72     /*i
73      *
74      */
75     const rti::routing::StreamInfo& stream_info() const
76     {
77         return stream_info_;
78     }
79
80      /*i
81      *
82      */
83     const std::string& name() const
84     {
85         return name_;
86     }
87
88     /*i
89      *
90      */
91     int32_t index()
92     {
93         return index_;
94     }
95
106     template <typename Data, typename Info>
107     TypedInput<Data, Info> get()
108     {
109         return this;
110     }
111
121     template <typename Data>
122     TypedInput<Data, dds::sub::SampleInfo> get()
123     {
124         return this;
125     }
126
127     /*i
128      *
129      */
130     RTI_RoutingServiceInput* native()
131     {
132         return native_;
133     }
134
135     // Implementation details
```

```
136 private:
137
138     template <typename Data, typename Info> friend class TypedInput;
139     friend class Route;
140     friend class std::allocator<Input>;
141     RTI_RoutingServiceInput *native_;
142     int32_t index_;
143     RTI_RoutingServiceRoute *native_route_;
144     rti::routing::StreamInfo stream_info_;
145     std::string name_;
146 };
147
148 template <typename Data, typename Info = dds::sub::SampleInfo>
149 class Selector;
150
159 template <typename Data, typename Info>
160 class TypedInput{
161 public:
162
163     typedef rti::routing::processor::Selector<Data, Info> Selector;
164
165     TypedInput(Input *input);
166
173     const rti::routing::StreamInfo& stream_info() const;
174
180     const std::string& name() const;
181
194     LoanedSamples<Data, Info> take();
195
204     LoanedSamples<Data, Info> read();
205
232     Selector select();
233
234     rti::routing::processor::Query query(
235             const dds::topic::Filter& filter);
236
248     bool active();
249
267     Data create_data();
268
269     /*i
270      *
271      */
272     TypedInput<Data, Info>* operator->()
273     {
274         return this;
275     }
276
277
294     dds::sub::DataReader<Data> dds_data_reader()
295     {
296         using rti::core::detail::create_from_native_entity;
297
298         DDS_DataReader *native_reader =
299                 RTI_RoutingServiceInput_get_dds_reader(input_->native_);
300         if (native_reader == NULL) {
301             throw dds::core::InvalidArgumentError(
302                     "invalid argument: input does not hold a DDS StreamReader");
303         }
304
305         typedef dds::sub::DataReader<Data> data_reader_type;
306         return rti::core::detail::create_from_native_entity<data_reader_type>(
307                 native_reader);
308     }
309
310 private:
311     TypedInput();
312
313     friend class rti::routing::processor::Selector<Data, Info>;
314
315     LoanedSamples<Data, Info> take(
316             const rti::routing::adapter::SelectorState& selector_state);
317
318     LoanedSamples<Data, Info> read(
319             const rti::routing::adapter::SelectorState& selector_state);
320
321     friend class rti::routing::processor::Route;
322     Input *input_;
323 };
324
325
```

```
360 template <typename Data, typename Info>
361 class Selector {
362 public:
363
372     Selector(const rti::routing::processor::TypedInput<Data, Info> input)
373         : typed_input_(input), query_(dds::core::null)
374     {
375
376     }
377
381     Selector(const Selector& other)
382         : typed_input_(other.typed_input_),
383           state_(other.state_),
384           query_(other.query_)
385     {
386
387     }
388
400     Selector& state(
401             const dds::sub::status::DataState& the_state)
402     {
403         state_.state(the_state);
404         return *this;
405     }
406
413     Selector& max_samples(
414             const int32_t count)
415     {
416         state_.max_samples(count);
417         return *this;
418     }
419
439     Selector& instance(
440             const dds::core::InstanceHandle& the_handle)
441     {
442         state_.instance(the_handle);
443         return *this;
444     }
445
475     Selector& next_instance(
476             const dds::core::InstanceHandle& the_handle)
477     {
478         state_.next_instance(the_handle);
479         return *this;
480     }
481
495     Selector& filter(
496             const dds::topic::Filter& the_filter)
497     {
498         state_.filter(the_filter);
499         return *this;
500     }
501
502
518     Selector& query(const rti::routing::processor::Query& the_query)
519     {
520         query_ = the_query;
521         state_.content(the_query.delegate().get()->query_data_);
522         return *this;
523     }
524
535     LoanedSamples<Data, Info> take()
536     {
537         return typed_input_.take(state_);
538     }
539
550     LoanedSamples<Data, Info> read()
551     {
552         return typed_input_.read(state_);
553     }
554
555 private:
556     rti::routing::processor::TypedInput<Data, Info> typed_input_;
557     rti::routing::adapter::SelectorState state_;
558     rti::routing::processor::Query query_;
559 };
560
561
562 template <typename Data, typename Info>
563 struct create_data_from_input {
564
```

```
565     static Data get(TypedInput<Data, Info>& )
566     {
567         return Data();
568     }
569 };
570
571 template <typename Info>
572 struct create_data_from_input<dds::core::xtypes::DynamicData, Info> {
573
574     static dds::core::xtypes::DynamicData get(
575             TypedInput<dds::core::xtypes::DynamicData, Info>& input)
576     {
577         if (input->stream_info().type_info().type_representation_kind()
578                 != TypeRepresentationKind::DYNAMIC_TYPE) {
579             throw dds::core::PreconditionNotMetError(
580                     "inconsistent data representation kind");
581         }
582         dds::core::xtypes::DynamicType *type_code =
583                 static_cast<dds::core::xtypes::DynamicType *> (
584                 input->stream_info().type_info().type_representation());
585         return dds::core::xtypes::DynamicData(*type_code);
586     }
587 };
588
589 template <typename Data, typename Info>
590 TypedInput<Data, Info>::TypedInput(Input *input)
591 : input_(input)
592 {
593 }
594
595 template <typename Data, typename Info>
596 const rti::routing::StreamInfo& TypedInput<Data, Info>::stream_info() const
597 {
598     return input_->stream_info_;
599 }
600
601 template <typename Data, typename Info>
602 const std::string& TypedInput<Data, Info>::name() const
603 {
604     return input_->name_;
605 }
606
607 template <typename Data, typename Info>
608 LoanedSamples<Data, Info> TypedInput<Data, Info>::take()
609 {
610     RTI_RoutingServiceLoanedSamples native_samples =
611             RTI_RoutingServiceLoanedSamples_INITIALIZER;
612     if (!RTI_RoutingServiceInput_take(
613             input_->native_,
614             &native_samples)) {
615         throw dds::core::Error("error taking samples from native input");
616     }
617     return LoanedSamples<Data, Info>(input_->native_, native_samples);
618 }
619
620 template <typename Data, typename Info>
621 LoanedSamples<Data, Info> TypedInput<Data, Info>::read()
622 {
623     RTI_RoutingServiceLoanedSamples native_samples =
624             RTI_RoutingServiceLoanedSamples_INITIALIZER;
625     if (!RTI_RoutingServiceInput_read(
626             input_->native_,
627             &native_samples)) {
628         throw dds::core::Error("error reading samples from native input");
629     }
630     return LoanedSamples<Data, Info>(input_->native_, native_samples);
631 }
632
633 template <typename Data, typename Info>
634 rti::routing::processor::Selector<Data, Info>
635 TypedInput<Data, Info>::select()
636 {
637     return rti::routing::processor::Selector<Data, Info>(*this);
638 }
639
640 template <typename Data, typename Info>
641 rti::routing::processor::Query TypedInput<Data, Info>::query(
642         const dds::topic::Filter& filter)
643 {
644     return rti::routing::processor::Query(input_->native_, filter);
645
```

```
646 }
647
648 template <typename Data, typename Info>
649 LoanedSamples<Data, Info> TypedInput<Data, Info>::take(
650         const rti::routing::adapter::SelectorState& selector_state)
651 {
652     RTI_RoutingServiceLoanedSamples native_samples =
653             RTI_RoutingServiceLoanedSamples_INITIALIZER;
654     if (!RTI_RoutingServiceInput_take_w_selector(
655             input_->native_,
656             &native_samples,
657             &selector_state.native())) {
658         throw dds::core::Error("error taking samples with selector from native input");
659     }
660     return LoanedSamples<Data, Info>(input_->native_, native_samples);
661 }
662
663 template <typename Data, typename Info>
664 LoanedSamples<Data, Info> TypedInput<Data, Info>::read(
665         const rti::routing::adapter::SelectorState& selector_state)
666 {
667     RTI_RoutingServiceLoanedSamples native_samples =
668             RTI_RoutingServiceLoanedSamples_INITIALIZER;
669     if (!RTI_RoutingServiceInput_read_w_selector(
670             input_->native_,
671             &native_samples,
672             &selector_state.native())) {
673         throw dds::core::Error("error reading samples with selector from native input");
674     }
675     return LoanedSamples<Data, Info>(input_->native_, native_samples);
676 }
677
678 template <typename Data, typename Info>
679 bool TypedInput<Data, Info>::active()
680 {
681     return RTI_RoutingServiceInput_is_active(input_->native_) ? true : false;
682 }
683
684 template <typename Data, typename Info>
685 Data TypedInput<Data,Info>::create_data()
686 {
687     return create_data_from_input<Data, Info>::get(*this);
688 }
689
697 typedef TypedInput<dds::core::xtypes::DynamicData> DynamicDataInput;
698 typedef dds::sub::DataReader<dds::core::xtypes::DynamicData> DynamicDataReader;
699
700 } } }
701
702
703 #endif // RTI_ROUTING_PROCESSOR_INPUT_HPP_
```

9.27 LoanedSample.hpp

```
1 /* $Id$
2
3 (c) Copyright, Real-Time Innovations, 2015-2016.
4 All rights reserved.
5
6 No duplications, whole or partial, manual or electronic, may be made
7 without express written permission.  Any such copies, or
8 revisions thereof, must display this notice unaltered.
9 This code contains trade secrets of Real-Time Innovations, Inc.
10
11
12 ================================================================================ */
13
14 #ifndef RTI_ROUTING_PROCESSOR_LOANED_SAMPLE_HPP_
15 #define RTI_ROUTING_PROCESSOR_LOANED_SAMPLE_HPP_
16
17 #include <exception>
18
19 namespace rti { namespace routing { namespace processor {
20
27 template <typename T, typename U>
28 class LoanedSample
29 {
```

```
30 public:
34     typedef T DataType;
38     typedef U InfoType;
39
40     LoanedSample()
41         : _data_ptr(NULL),
42           _info_ptr(NULL)
43     {}
44
45     LoanedSample(const DataType * the_data, const InfoType * the_info)
46         : _data_ptr(the_data),
47           _info_ptr(the_info)
48     {}
49
50
51     LoanedSample(const void * the_data, const void * the_info)
52         : _data_ptr(static_cast<const T*>(the_data)),
53           _info_ptr(static_cast<const U*>(the_info))
54     {}
55
56
57     const DataType & data() const
58     {
59         if (_info_ptr != NULL && !info().valid()) {
60             throw dds::core::PreconditionNotMetError("Invalid data");
61         }
62
63         return *_data_ptr;
64     }
65
69     const InfoType & info() const {
70         if (_info_ptr == NULL) {
71             throw dds::core::PreconditionNotMetError(
72                     "Sample Info not available");
73         }
74
75         return *_info_ptr;
76     }
77
78
79     operator const DataType & () const
80     {
81         return data();
82     }
83
84     LoanedSample<DataType, InfoType> * operator ->() {
85         return this;
86     }
87
88     const LoanedSample<DataType, InfoType> * operator ->() const {
89         return this;
90     }
91
95     bool operator==(const LoanedSample& other) const
96     {
97         if (info() != other.info()) {
98             return false;
99         }
100
101          if (info().valid()) {
102              if (data() != other.data()) {
103                  return false;
104              }
105          }
106
107          return true;
108      }
109
110      bool operator!=(const LoanedSample& other) const
111      {
112          return !(*this == other);
113      }
114
115      void swap(LoanedSample & other) throw()
116      {
117          using std::swap;
118
119          swap(_data_ptr, other._data_ptr);
120          swap(_info_ptr, other._info_ptr);
121      }
122
```

```
123 private:
124     const DataType * _data_ptr;
125     const InfoType * _info_ptr;
126 };
127
128 } } }
129
130 #endif // RTI_ROUTING_PROCESSOR_LOANED_SAMPLE_HPP_
```

9.28 LoanedSamples.hpp

```
1 /* $Id$
2
3 (c) Copyright, Real-Time Innovations, 2013-2016.
4 All rights reserved.
5
6 No duplications, whole or partial, manual or electronic, may be made
7 without express written permission.  Any such copies, or
8 revisions thereof, must display this notice unaltered.
9 This code contains trade secrets of Real-Time Innovations, Inc.
10
11
12 ============================================================================= */
13
14 #ifndef RTI_ROUTING_PROCESSOR_LOANED_SAMPLES_HPP_
15 #define RTI_ROUTING_PROCESSOR_LOANED_SAMPLES_HPP_
16
17 #include <utility>  // std::move
18 #include <iterator>
19
20 #include "routingservice/routingservice_adapter_new.h"
21 #include <dds/core/types.hpp>
22 #include <rti/routing/processor/LoanedSample.hpp>
23 #include <rti/routing/adapter/StreamReader.hpp>
24 #include <rti/routing/processor/SampleIterator.hpp>
25
26 namespace rti { namespace routing { namespace processor {
27
28 template <typename T, typename U> class TypedInput;
29
30
77 template <typename T, typename U = dds::sub::SampleInfo>
78 class LoanedSamples {
79 public:
80
81     typedef rti::routing::adapter::StreamReader StreamReader;
82     typedef typename std::vector<StreamReader::SamplePtr> SampleSeqType;
83     typedef typename std::vector<StreamReader::InfoPtr> InfoSeqType;
84
88     typedef SampleIterator<T, U> iterator;
89     typedef SampleIterator<T, U> const_iterator;
90     typedef typename SampleIterator<T, U>::value_type value_type;
91     typedef std::ptrdiff_t difference_type;
92
96     LoanedSamples() : input_(NULL)
97     {
98         native_samples_.data_array = NULL;
99         native_samples_.info_array = NULL;
100         native_samples_.length = 0;
101
102     }
103
104 public:
105     // Private c-tor only to be used by static method create_from_loans()
106     LoanedSamples(
107             RTI_RoutingServiceInput *native_input,
108             RTI_RoutingServiceLoanedSamples& native_samples)
109         : input_(native_input),
110           native_samples_(native_samples)
111     {
112     }
113
114 public:
115
122     ~LoanedSamples() throw()
123     {
124         try {
```

```
125             return_loan();
126         } catch (const std::exception&) { // Do not throw in destructor
127
128         }
129     }
130
131
142     LoanedSample<T,U> operator [] (size_t index)
143     {
144         return native_samples_.info_array == NULL
145                 ? LoanedSample<T, U>(native_samples_.data_array[index], NULL)
146                 : LoanedSample<T, U>(
147                         native_samples_.data_array[index],
148                         native_samples_.info_array[index]);
149     }
150
154     int32_t length() const
155     {
156         return native_samples_.length;
157     }
158
181     void return_loan()
182     {
183         if (input_) {
184             if (!RTI_RoutingServiceInput_return_loan(
185                     input_,
186                     &native_samples_)) {
187                 throw dds::core::Error("error returning loaned samples to native input");
188             }
189             input_ = NULL; // Indicate that this object doesn't hold a loan anymore
190         }
191     }
192
193
198     bool has_infos()
199     {
200         return (native_samples_.info_array != NULL);
201     }
202
203
207     iterator begin()
208     {
209         return iterator(
210                 native_samples_.data_array,
211                 native_samples_.info_array,
212                 native_samples_.length);
213     }
214
218     iterator end()
219     {
220         return iterator(
221                 native_samples_.data_array,
222                 native_samples_.info_array,
223                 native_samples_.length,
224                 native_samples_.length);
225     }
226
230     const_iterator begin() const
231     {
232         return const_iterator(
233                 native_samples_.data_array,
234                 native_samples_.info_array,
235                 native_samples_.length);
236     }
237
241     const_iterator end() const
242     {
243         return const_iterator(
244                 native_samples_.data_array,
245                 native_samples_.info_array,
246                 native_samples_.length,
247                 native_samples_.length);
248     }
249
253     void swap(LoanedSamples& other) throw()
254     {
255         std::swap(input_, other.input_);
256         std::swap(native_samples_, other.native_samples_);
257     }
258
259     LoanedSamples(LoanedSamples&& other)
```

```
260         :input_(NULL)
261     {
262         native_samples_.data_array = NULL;
263         native_samples_.info_array = NULL;
264         native_samples_.length = 0;
265         other.swap(*this);
266     }
267 
268 
269     LoanedSamples& operator= (LoanedSamples&& other) throw ()
270     {
271         // clean up existing values
272         LoanedSamples temp(std::move(other));
273         temp.swap(*this);
274         return *this;
275     }
276 
277 private:
278    friend class TypedInput<T, U>;
279 
280     void release()
281     {
282         input_ = NULL;
283     }
284 
285     RTI_RoutingServiceLoanedSamples& native_samples()
286     {
287         return native_samples_;
288     }
289 
290 private:
291     // reference to the reader that created this LoanedSamples object
292     RTI_RoutingServiceInput *input_;
293     RTI_RoutingServiceLoanedSamples native_samples_;
294 
295 };
296 
297 template <typename T, typename U>
298 LoanedSamples<T, U> move(LoanedSamples<T,U> & ls) OMG_NOEXCEPT
299 {
300     return std::move(ls);
301 }
302 
303 
304 } } }
305 
306 #endif // RTI_ROUTING_PROCESSOR_LOANED_SAMPLES_HPP_
```

9.29 Output.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10 
11 #ifndef RTI_ROUTING_PROCESSOR_OUTPUT_HPP_
12 #define RTI_ROUTING_PROCESSOR_OUTPUT_HPP_
13 
14 #include <dds/core/Value.hpp>
15 #include <dds/core/SafeEnumeration.hpp>
16 #include <rti/core/NativeValueType.hpp>
17 #include <dds/pub/DataWriter.hpp>
18 #include <dds/core/xtypes/DynamicData.hpp>
19 
20 #include "routingservice/routingservice_processor.h"
21 #include "routingservice/routingservice_adapter_new.h"
22 #include <rti/routing/adapter/StreamWriter.hpp>
23 
24 namespace rti { namespace routing { namespace processor {
25 
26 class Route;
27 
```

```
28 template <typename Data, typename Info> class TypedOutput;
29
30 template <typename Data, typename Info = dds::sub::SampleInfo>
31 class TypedOutput;
32
44 class Output {
45 public:
46
47     /*i
48      */
49     Output(
50         RTI_RoutingServiceOutput *native,
51         int32_t index,
52         RTI_RoutingServiceRoute *native_route) :
53             native_(native),
54             index_(index),
55             native_route_(native_route),
56             stream_info_(*RTI_RoutingServiceOutput_get_stream_info(native)),
57             name_(RTI_RoutingServiceOutput_get_name(native))
58     {
59     }
60
61     /*i
62      */
63     const rti::routing::StreamInfo& stream_info() const
64     {
65         return stream_info_;
66     }
67
68     /*i
69      *
70      */
71     const std::string& name() const
72     {
73         return name_;
74     }
75
76     /*i
77      *
78      */
79     int32_t index()
80     {
81         return index_;
82     }
83
93     template <typename Data, typename Info>
94     TypedOutput<Data, Info> get()
95     {
96         return this;
97     }
98
108     template <typename Data>
109     TypedOutput<Data, dds::sub::SampleInfo> get()
110     {
111         return this;
112     }
113
114 private:
115     template <typename Data, typename Info> friend class TypedOutput;
116     friend class Route;
117     RTI_RoutingServiceOutput *native_;
118     int32_t index_;
119     RTI_RoutingServiceRoute *native_route_;
120     rti::routing::StreamInfo stream_info_;
121     std::string name_;
122 };
123
132 template <typename Data, typename Info>
133 class TypedOutput{
134 public:
135     TypedOutput(Output *output) ;
136
137     TypedOutput<Data, Info>* operator->();
138
145     const rti::routing::StreamInfo& stream_info() const;
146
152     const std::string& name() const;
153
165     void write(const Data& sample, const Info& info);
166
167     /*
```

```
168      * @brief Writes a sample given its data only.
169      *
170      * This operation will call rti::routing::adapter::StreamWriter::write
171      * on the underlying rti::routing::adapter::StreamWriter, providing a null
172      * value for the info sample. The underlying adapter will compute any
173      * associated metadata from the data sample.
174      */
175     void write(const Data& sample);
176 
195     Data create_data();
196 
213     dds::pub::DataWriter<Data> dds_data_writer()
214     {
215         DDS_DataWriter *native_writer =
216                 RTI_RoutingServiceOutput_get_dds_writer(output_->native_);
217         if (native_writer == NULL) {
218             throw dds::core::InvalidArgumentError(
219                     "invalid argument: input does not hold a DDS StreamWriter");
220         }
221 
222         typedef dds::pub::DataWriter<Data> data_writer_type;
223         return rti::core::detail::create_from_native_entity<data_writer_type>(
224                 native_writer);
225     }
226 
227 private:
228     friend class rti::routing::processor::Route;
229 
230     Output *output_;
231 };
232 
233 template <typename Data, typename Info>
234 struct create_data_from_output {
235 
236     static Data get(TypedOutput<Data, Info>& )
237     {
238         return Data();
239     }
240 };
241 
242 
243 template <typename Info>
244 struct create_data_from_output<dds::core::xtypes::DynamicData, Info> {
245 
246     static dds::core::xtypes::DynamicData get(TypedOutput<dds::core::xtypes::DynamicData, Info>& output)
247     {
248         if (output->stream_info().type_info().type_representation_kind()
249                 != TypeRepresentationKind::DYNAMIC_TYPE) {
250             throw dds::core::PreconditionNotMetError(
251                     "inconsistent data representation kind");
252         }
253         dds::core::xtypes::DynamicType *type_code =
254                 static_cast<dds::core::xtypes::DynamicType *> (
255                 output->stream_info().type_info().type_representation());
256         return dds::core::xtypes::DynamicData(*type_code);
257     }
258 };
259 
260 
261 template <typename Data, typename Info>
262 TypedOutput<Data,Info>::TypedOutput(Output* output) : output_(output)
263 {
264 }
265 
266 template <typename Data, typename Info>
267 const rti::routing::StreamInfo& TypedOutput<Data, Info>::stream_info() const
268 {
269     return output_->stream_info_;
270 }
271 
272 template <typename Data, typename Info>
273 const std::string& TypedOutput<Data, Info>::name() const
274 {
275     return output_->name_;
276 }
277 
278 template <typename Data, typename Info>
279 TypedOutput<Data, Info>* TypedOutput<Data,Info>::operator->()
280 {
281     return this;
282 }
```

```
283
284 template <typename Data, typename Info>
285 void TypedOutput<Data, Info>::write(const Data& sample, const Info& info)
286 {
287     if (!RTI_RoutingServiceOutput_write_sample(
288             output_->native_,
289             const_cast<void *> (reinterpret_cast<const void*> (&sample)),
290             const_cast<void *> (reinterpret_cast<const void*> (&info)))) {
291         throw dds::core::Error("error writing sample to native output");
292     }
293 }
294
295 template <typename Data, typename Info>
296 void TypedOutput<Data,Info>::write(const Data& sample)
297 {
298     if (!RTI_RoutingServiceOutput_write_sample(
299             output_->native_,
300             const_cast<void *> (reinterpret_cast<const void*> (&sample)),
301             NULL)) {
302         throw dds::core::Error("error writing sample to native output");
303     }
304 }
305
306 template <typename Data, typename Info>
307 Data TypedOutput<Data,Info>::create_data()
308 {
309     return create_data_from_output<Data, Info>::get(*this);
310 }
311
312
320 typedef TypedOutput<dds::core::xtypes::DynamicData> DynamicDataOutput;
321 typedef dds::pub::DataWriter<dds::core::xtypes::DynamicData> DynamicDataWriter;
322
323 } } }
324
325
326 #endif // RTI_ROUTING_PROCESSOR_OUTPUT_HPP_
```

9.30 PortIterator.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_PROCESSOR_PORTS_ITERATOR_HPP_
12 #define RTI_ROUTING_PROCESSOR_PORTS_ITERATOR_HPP_
13
14 #include <dds/core/Value.hpp>
15 #include <dds/core/SafeEnumeration.hpp>
16 #include <rti/core/NativeValueType.hpp>
17
18
19 #include <dds/core/xtypes/DynamicData.hpp>
20 #include <dds/sub/SampleInfo.hpp>
21 #include <rti/routing/processor/Input.hpp>
22 #include <rti/routing/processor/Output.hpp>
23
24 namespace rti { namespace routing { namespace processor {
25
26 template <typename PORT, typename DataRep, typename InfoRep> struct TypedPortType;
27
28 template <typename DataRep, typename InfoRep>
29 struct TypedPortType<Input, DataRep, InfoRep>
30 {
31     typedef TypedInput<DataRep, InfoRep> type;
32     typedef RTI_RoutingServiceInput native_type;
33
34     static native_type * begin(RTI_RoutingServiceRoute *native_route)
35     {
36         return  RTI_RoutingServiceRoute_get_first_input(native_route);
37     }
```

```
38
39      static native_type * next(
40              RTI_RoutingServiceRoute *native_route,
41              native_type *native)
42      {
43          return  RTI_RoutingServiceRoute_get_next_input(native_route, native);
44      }
45
46      static Input* from_native(native_type *native)
47      {
48          return static_cast<Input*>(RTI_RoutingServiceInput_get_user_data(
49                  native));
50      }
51
52  };
53
54  template <typename DataRep, typename InfoRep>
55  struct TypedPortType<Output, DataRep, InfoRep>
56  {
57      typedef TypedOutput<DataRep, InfoRep> type;
58      typedef RTI_RoutingServiceOutput native_type;
59
60      static native_type * begin(RTI_RoutingServiceRoute *native_route)
61      {
62          return  RTI_RoutingServiceRoute_get_first_output(native_route);
63      }
64
65      static native_type * next(
66              RTI_RoutingServiceRoute *native_route,
67              native_type *native)
68      {
69          return  RTI_RoutingServiceRoute_get_next_output(native_route, native);
70      }
71
72      static Output* from_native(native_type *native)
73      {
74          return static_cast<Output*>(RTI_RoutingServiceOutput_get_user_data(
75                  native));
76      }
77  };
78
79  template <typename PORT,
80            typename DataRep,
81            typename InfoRep = dds::sub::SampleInfo>
82  class PortIterator  {
83  public:
84      // iterator traits
85     typedef std::random_access_iterator_tag iterator_category;
86     typedef typename TypedPortType<PORT, DataRep, InfoRep>::type value_type;
87     typedef value_type reference;
88     typedef value_type pointer;
89     typedef std::ptrdiff_t difference_type;
90
91
92      explicit PortIterator(
93              RTI_RoutingServiceRoute *native_route)
94          : native_route_(native_route)
95      {
96         native_ = TypedPortType<PORT, DataRep, InfoRep>::begin(native_route);
97      }
98
99      explicit PortIterator()
100         : native_route_(NULL), native_(NULL)
101      {
102      }
103
104      PortIterator (const PortIterator<PORT, DataRep, InfoRep>& other)
105          : native_route_(other.native_route_),
106            native_(other.native_)
107      {}
108
109      PortIterator & operator = (const PortIterator<PORT, DataRep, InfoRep>& other)
110      {
111          native_route_ = other.native_route_;
112          native_ = other.native_;
113
114          return *this;
115      }
116
117      value_type operator*()
118      {
```

```
119         return TypedPortType<PORT, DataRep, InfoRep>::from_native(native_);
120     }
121
122     value_type operator->()
123     {
124         return *(*this);
125     }
126
127     PortIterator & operator++()
128     {
129         native_ = TypedPortType<PORT, DataRep, InfoRep>::next(
130                 native_route_,
131                 native_);
132
133         return *this;
134     }
135
136     PortIterator operator++(int)
137     {
138         PortIterator temp(*this);
139         ++(*this);
140         return temp;
141     }
142
143
144     friend bool operator<(
145             const PortIterator & s1,
146             const PortIterator & s2)
147     {
148         return s1.index() < s2.index();
149     }
150
151     friend bool operator>(
152             const PortIterator & s1,
153             const PortIterator & s2)
154     {
155         return s1.index() > s2.index();
156     }
157
158     friend bool operator<=(
159             const PortIterator & s1,
160             const PortIterator & s2)
161     {
162         return s1.index() <= s2.index();
163     }
164
165     friend bool operator>=(
166             const PortIterator & s1,
167             const PortIterator & s2)
168     {
169         return s1.index() >= s2.index();
170     }
171
172     friend bool operator==(
173             const PortIterator & s1,
174             const PortIterator & s2)
175     {
176         return (s1.index() == s2.index());
177     }
178
179     friend bool operator!=(
180             const PortIterator & s1,
181             const PortIterator & s2)
182     {
183         return !(s1 == s2);
184     }
185
186 private:
187     int32_t index() const
188     {
189         return (native_ == NULL)
190                 ? RTI_INT32_MAX
191                 : TypedPortType<PORT, DataRep, InfoRep>::from_native(native_)->index();
192     }
193
194     RTI_RoutingServiceRoute *native_route_;
195     typename TypedPortType<PORT, DataRep, InfoRep>::native_type *native_;
196
197 };
198
199
```

```
200 } } }
201
202 #endif // RTI_ROUTING_PROCESSOR_PORTS_ITERATOR_HPP_
```

9.31 Processor.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_PROCESSOR_PROCESSOR_HPP_
12 #define RTI_ROUTING_PROCESSOR_PROCESSOR_HPP_
13
14 #include <dds/core/Value.hpp>
15 #include <dds/core/SafeEnumeration.hpp>
16 #include <rti/core/NativeValueType.hpp>
17
18 #include "routingservice/routingservice_processor.h"
19
20 #include <rti/routing/processor/Route.hpp>
21 #include <rti/routing/UpdatableEntity.hpp>
22
23 namespace rti { namespace routing { namespace processor {
24
54 class Processor : public rti::routing::UpdatableEntity {
55 public:
56
57
83     virtual void on_input_enabled(
84             Route& route,
85             rti::routing::processor::Input& input) = 0;
86
112     virtual void on_input_disabled(
113             Route& route,
114             rti::routing::processor::Input& input) = 0;
115
141     virtual void on_output_enabled(
142             Route& route,
143             rti::routing::processor::Output& output) = 0;
144
170     virtual void on_output_disabled(
171             Route& route,
172             rti::routing::processor::Output& output) = 0;
173
195     virtual void on_start(Route& route) = 0;
196
219     virtual void on_stop(Route& route) = 0;
220
247     virtual void on_run(Route& route) = 0;
248
271     virtual void on_pause(Route& route) = 0;
272
300     virtual void on_periodic_action(Route& route) = 0;
301
328     virtual void on_data_available(Route& route) = 0;
329
333     virtual ~Processor() {}
334 };
335
336
347 class NoOpProcessor : public Processor {
348
349
350     virtual void on_input_enabled(Route&, rti::routing::processor::Input&)
351     {
352     }
353
354     virtual void on_input_disabled(Route&, rti::routing::processor::Input&)
355     {
356     }
357
```

```
358     virtual void on_output_enabled(Route&, rti::routing::processor::Output&)
359     {
360     }
361
362     virtual void on_output_disabled(Route&, rti::routing::processor::Output&)
363     {
364     }
365
366     virtual void on_start(Route&)
367     {
368     }
369
370     virtual void on_stop(Route&)
371     {
372     }
373
374     virtual void on_run(Route&)
375     {
376     }
377
378     virtual void on_pause(Route&)
379     {
380     }
381
382
383     virtual void on_data_available(Route&)
384     {
385     }
386
387
388     virtual void on_periodic_action(Route&)
389     {
390     }
391
392 };
393
394 } } }
395
396
397 #endif // RTI_ROUTING_PROCESSOR_PROCESSOR_HPP_
```

9.32 ProcessorPlugin.hpp File Reference

RTI Routing Service C++ Processor API.

```
#include "routingservice/routingservice_log.h"
#include <rti/config/Version.hpp>
#include <rti/routing/PropertySet.hpp>
#include <rti/routing/processor/Route.hpp>
#include <rti/routing/processor/Processor.hpp>
#include <rti/routing/processor/detail/ProcessorPluginForwarder.hpp>
```

Data Structures

- class **rti::routing::processor::ProcessorPlugin**

  The top-level plug-in class.

Macros

- #define **RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DECL**(PROCESSOR_CLASS)

  Utility macro that declares a native extern function that can be used to load a ProcessorPlugin through a shared library.
- #define **RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DEF**(PROCESSOR_CLASS)

  Utility macro that implements the ProcessorPlugin entry point declared with RTI_PROCESSOR_PLUGIN_CREATE_↩ FUNCTION_DECL.

9.32.1 Detailed Description

RTI Routing Service C++ Processor API.

9.33 ProcessorPlugin.hpp

Go to the documentation of this file.

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_PROCESSOR_PROCESSOR_PLUGIN_HPP_
12 #define RTI_ROUTING_PROCESSOR_PROCESSOR_PLUGIN_HPP_
13
14 #include "routingservice/routingservice_log.h"
15
16 #include <rti/config/Version.hpp>
17 #include <rti/routing/PropertySet.hpp>
18 #include <rti/routing/processor/Route.hpp>
19 #include <rti/routing/processor/Processor.hpp>
20
193 namespace rti { namespace routing { namespace processor {
194
202 class ProcessorPlugin {
203 public:
204
232     virtual Processor * create_processor(
233             Route& route,
234             const rti::routing::PropertySet& properties) = 0;
235
253     virtual void delete_processor(
254             Route& route,
255             Processor *processor) = 0;
256
266     virtual rti::config::LibraryVersion get_version() const
267     {
268         return rti::config::LibraryVersion();
269     }
270
274     virtual ~ProcessorPlugin() {}
275
276 };
277
278 } } }
279
280
281 #include <rti/routing/processor/detail/ProcessorPluginForwarder.hpp>
282
283
354 #define RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DECL(PROCESSOR_CLASS) \
355 extern "C" RTI_USER_DLL_EXPORT struct RTI_RoutingServiceProcessorPlugin* \
356     PROCESSOR_CLASS ## _create_processor_plugin(\
357         const struct RTI_RoutingServiceProperties *, \
358         RTI_RoutingServiceEnvironment *); \
359
365 #define RTI_PROCESSOR_PLUGIN_CREATE_FUNCTION_DEF(PROCESSOR_CLASS) \
366 struct RTI_RoutingServiceProcessorPlugin * PROCESSOR_CLASS ## _create_processor_plugin( \
367         const struct RTI_RoutingServiceProperties * native_properties, \
368         RTI_RoutingServiceEnvironment *environment) \
369 { \
370     PropertySet properties; \
371     rti::routing::PropertyAdapter::add_properties_from_native(\
372             properties,\
373             native_properties); \
374     try { \
375         return rti::routing::processor::detail::ProcessorPluginForwarder::create_plugin(\
376                 new PROCESSOR_CLASS(properties)); \
```

```
377     } catch (const std::exception& ex) {\
378         RTI_RoutingServiceEnvironment_set_error(\
379                 environment,\
380                 "%s",\
381                 ex.what());\
382     } catch (...) {}\
383     \
384     return NULL; \
385 }
386
387
388
389 #endif // RTI_ROUTING_PROCESSOR_PROCESSOR_PLUGIN_HPP_
```

9.34 Query.hpp

```
1 /* $Id$
2
3 (c) Copyright, Real-Time Innovations, 2015-2016.
4 All rights reserved.
5
6 No duplications, whole or partial, manual or electronic, may be made
7 without express written permission.  Any such copies, or
8 revisions thereof, must display this notice unaltered.
9 This code contains trade secrets of Real-Time Innovations, Inc.
10
11
12 ============================================================================= */
13
14 #ifndef RTI_ROUTING_PROCESSOR_QUERY_HPP_
15 #define RTI_ROUTING_PROCESSOR_QUERY_HPP_
16
17 #include "routingservice/routingservice_processor.h"
18
19 #include <dds/core/Reference.hpp>
20 #include <rti/core/detail/SelfReference.hpp>
21
22
23 namespace rti { namespace routing { namespace processor {
24
25 class Input;
26
27 class QueryHolder : public rti::core::detail::RetainableType<QueryHolder>{
28 public:
29
30     QueryHolder() : input_(NULL), query_data_(NULL)
31     {
32     }
33
34     QueryHolder(RTI_RoutingServiceInput *input, const dds::topic::Filter& the_filter)
35             : input_(input),
36               query_data_(NULL)
37     {
38         filter(the_filter);
39     }
40
41     ~QueryHolder()
42     {
43         if (query_data_ != NULL) {
44             /*
45              * Ignore the error. We can't throw in the destructor and this
46              * layer does not log anything
47              */
48             (void) RTI_RoutingServiceInput_delete_content_query(
49                     input_,
50                     query_data_);
51         }
52     }
53
54     void filter(const dds::topic::Filter& filter)
55     {
56         RTI_RoutingServiceSelectorContent content =
57                 RTI_RoutingServiceSelectorContent_INITIALIZER;
58         content.expression = (char *) filter.expression().c_str();
59         rti::core::native_conversions::to_native(
60                 content.expression_parameters,
61                 filter->parameters());
```

```
62          query_data_ = RTI_RoutingServiceInput_create_content_query(
63                  input_,
64                  query_data_,
65                  &content);
66          DDS_StringSeq_finalize(&content.expression_parameters);
67          if (query_data_ == NULL) {
68              throw dds::core::Error("error creating content query from native input");
69          }
70      }
71
72      RTI_RoutingServiceInput *input_;
73      void* query_data_;
74  };
75
92  class Query : public dds::core::Reference<rti::routing::processor::QueryHolder> {
93  public:
94      typedef dds::core::Reference<QueryHolder> Base;
95      OMG_DDS_REF_TYPE_NOTYPENAME(
96              Query,
97              dds::core::Reference,
98              rti::routing::processor::QueryHolder);
99
100      explicit Query(Base::DELEGATE_REF_T reference) : Base(reference)
101      {
102          if (this->delegate()) {
103              this->delegate()->remember_reference(this->delegate());
104          }
105      }
106
116      Query(RTI_RoutingServiceInput *input, const dds::topic::Filter& filter)
117              : Base(new QueryHolder(input, filter))
118      {
119          this->delegate()->remember_reference(this->delegate());
120      }
121
122
130      void filter(const dds::topic::Filter& filter)
131      {
132          this->delegate()->filter(filter);
133      }
134  };
135
136  } } }
137
138  #endif // RTI_ROUTING_PROCESSOR_QUERY_HPP_
```

9.35 Route.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11  #ifndef RTI_ROUTING_PROCESSOR_ROUTE_HPP_
12  #define RTI_ROUTING_PROCESSOR_ROUTE_HPP_
13
14  #include <dds/core/Value.hpp>
15  #include <dds/core/SafeEnumeration.hpp>
16  #include <rti/core/NativeValueType.hpp>
17
18  #include "routingservice/routingservice_processor.h"
19  #include "routingservice/routingservice_processor_impl.h"
20
21  #include <dds/core/xtypes/DynamicData.hpp>
22  #include <dds/sub/SampleInfo.hpp>
23  #include <rti/routing/adapter/AdapterPlugin.hpp>
24  #include <rti/routing/processor/Input.hpp>
25  #include <rti/routing/processor/Output.hpp>
26  #include <rti/routing/processor/LoanedSamples.hpp>
27  #include <rti/routing/processor/PortIterator.hpp>
28
29  namespace rti { namespace routing { namespace processor {
```

```
30
31 namespace detail {
32
33 class ProcessorForwarder;
34
35 }
36
37 class Route;
38
39 template <typename PORT> struct port_list;
40
41 template <> struct port_list<Input>
42 {
43     static std::vector<Input>& get(Route& route);
44 };
45
46 template <> struct port_list<Output>
47 {
48     static std::vector<Output>& get(Route& route);
49 };
50
84 class Route {
85
86 public:
87
88     template <typename DataRep, typename InfoRep = dds::sub::SampleInfo>
89     struct Inputs {
90         typedef PortIterator<Input, DataRep, InfoRep> iterator;
91
92         Inputs(RTI_RoutingServiceRoute *native_route)
93                 : native_route_(native_route)
94         {
95         }
96
97         iterator begin()
98         {
99             return iterator(native_route_);
100         }
101
102         iterator end()
103         {
104             return iterator();
105         }
106
107     private:
108         RTI_RoutingServiceRoute *native_route_;
109     };
110
111     template <typename DataRep, typename InfoRep = dds::sub::SampleInfo>
112     struct Outputs {
113         typedef PortIterator<Output, DataRep, InfoRep> iterator;
114
115         Outputs(RTI_RoutingServiceRoute *native_route)
116                 : native_route_(native_route)
117         {
118         }
119
120         iterator begin()
121         {
122             return iterator(native_route_);
123         }
124
125         iterator end()
126         {
127             return iterator();
128         }
129
130     private:
131         RTI_RoutingServiceRoute *native_route_;
132     };
133
134
139     int32_t input_count() const
140     {
141         return RTI_RoutingServiceRoute_get_input_count(native_route_);
142     }
143
148     int32_t output_count() const
149     {
150         return RTI_RoutingServiceRoute_get_output_count(native_route_);
151     }
```

```
152
157     bool input_enabled(int32_t index) const
158     {
159         return RTI_RoutingServiceRoute_is_input_enabled(native_route_, index)
160                 ? true
161                 : false;
162     }
163
168     bool output_enabled(int32_t index) const
169     {
170         return RTI_RoutingServiceRoute_is_output_enabled(native_route_, index)
171                 ? true
172                 : false;
173     }
174
193     Input& input(int32_t index)
194     {
195         return *(get_input_at_index(index));
196     }
197
211     Input& input(const std::string& name)
212     {
213         return *(lookup_input(name));
214     }
215
237     template <typename DataRep, typename InfoRep>
238     TypedInput<DataRep, InfoRep> input(int32_t index)
239     {
240         return &input(index);
241     }
242
248     template <typename DataRep>
249     TypedInput<DataRep> input(int32_t index)
250     {
251         return input<DataRep, dds::sub::SampleInfo>(index);
252     }
253
259     template <typename DataRep, typename InfoRep>
260     TypedInput<DataRep, InfoRep> input(const std::string& name)
261     {
262         return &input(name);
263     }
264
269     template <typename DataRep>
270     TypedInput<DataRep> input(const std::string& name)
271     {
272         return input<DataRep, dds::sub::SampleInfo>(name);
273     }
274
293     Output& output(int32_t index)
294     {
295         return *(get_output_at_index(index));
296     }
297
311     Output& output(const std::string& name)
312     {
313         return *(lookup_output(name));
314     }
315
336     template <typename DataRep, typename InfoRep>
337     TypedOutput<DataRep, InfoRep> output(int32_t index)
338     {
339         return &output(index);
340     }
341
347     template <typename DataRep>
348     TypedOutput<DataRep> output(int32_t index)
349     {
350         return output<DataRep, dds::sub::SampleInfo>(index);
351     }
352
358     template <typename DataRep, typename InfoRep>
359     TypedOutput<DataRep, InfoRep> output(const std::string& name)
360     {
361         return &output(name);
362     }
363
368     template <typename DataRep>
369     TypedOutput<DataRep> output(const std::string& name)
370     {
371         return output<DataRep, dds::sub::SampleInfo>(name);
```

```
372     }
373
383     template <typename DataRep, typename InfoRep>
384     Inputs<DataRep, InfoRep> inputs()
385     {
386         return Inputs<DataRep, InfoRep>(native_route_);
387     }
388
393     template <typename DataRep>
394     Inputs<DataRep> inputs()
395     {
396         return inputs<DataRep, dds::sub::SampleInfo>();
397     }
398
408     template <typename DataRep, typename InfoRep>
409     Outputs<DataRep, InfoRep> outputs()
410     {
411         return Outputs<DataRep, InfoRep>(native_route_);
412     }
413
418     template <typename DataRep>
419     Outputs<DataRep> outputs()
420     {
421         return outputs<DataRep, dds::sub::SampleInfo>();
422     }
423
437     void period(const dds::core::Duration& period)
438     {
439         struct DDS_Duration_t native_period;
440         rti::core::native_conversions::to_native(native_period, period);
441         RTI_RoutingServiceRoute_set_period(
442                 native_route_,
443                 &native_period);
444     }
445
446     dds::core::Duration period() const
447     {
448         struct DDS_Duration_t native_period;
449         RTI_RoutingServiceRoute_get_period(
450                 native_route_,
451                 &native_period);
452
453         return rti::core::native_conversions::from_native(native_period);
454     }
455
461     const std::string& full_name() const
462     {
463         return full_name_;
464     }
465
466
467 private:
468     friend class detail::ProcessorForwarder;
469     friend class Output;
470     friend struct port_list<Input>;
471     friend struct port_list<Output>;
472
473     typedef std::vector<Input>::iterator private_input_it;
474     typedef std::vector<Output>::iterator private_output_it;
475
476     Route& operator= (const Route& other);
477
478     Route(RTI_RoutingServiceRoute *native_route,
479             RTI_RoutingServiceEnvironment *native_env)
480         : native_route_(native_route),
481           native_env_(native_env)
482     {
483         const char *native_full_name =
484                 RTI_RoutingServiceRoute_get_full_name(native_route);
485         if (native_full_name == nullptr) {
486             throw dds::core::InvalidArgumentError(
487                     "Native Route has invalid full name");
488         }
489         full_name_ = native_full_name;
490     }
491
492     Input* get_input_at_index(int32_t index)
493     {
494         RTI_RoutingServiceInput *native_input =
495                 RTI_RoutingServiceRoute_get_input_at(native_route_, index);
496         if (native_input == NULL) {
```

```
497             throw dds::core::InvalidArgumentError(
498                     RTI_RoutingServiceEnvironment_get_error_message(native_env_));
499         }
500
501         return static_cast<Input*>(RTI_RoutingServiceInput_get_user_data(native_input));
502     }
503
504     Output* get_output_at_index(int32_t index)
505     {
506         RTI_RoutingServiceOutput *native_output =
507                 RTI_RoutingServiceRoute_get_output_at(native_route_, index);
508         if (native_output == NULL) {
509             throw dds::core::InvalidArgumentError(
510                     RTI_RoutingServiceEnvironment_get_error_message(native_env_));
511         }
512
513         return static_cast<Output*>(RTI_RoutingServiceOutput_get_user_data(native_output));
514     }
515
516
517     Input* lookup_input(const std::string& name)
518     {
519         RTI_RoutingServiceInput *native_input =
520                 RTI_RoutingServiceRoute_lookup_input_by_name(
521                         native_route_,
522                         name.c_str());
523         if (native_input == NULL) {
524             throw dds::core::InvalidArgumentError(
525                     RTI_RoutingServiceEnvironment_get_error_message(native_env_));
526         }
527         return static_cast<Input*>(RTI_RoutingServiceInput_get_user_data(native_input));
528
529     }
530
531     Output* lookup_output(const std::string& name)
532     {
533        RTI_RoutingServiceOutput *native_output =
534                 RTI_RoutingServiceRoute_lookup_output_by_name(
535                         native_route_,
536                         name.c_str());
537         if (native_output == NULL) {
538             throw dds::core::InvalidArgumentError(
539                     RTI_RoutingServiceEnvironment_get_error_message(native_env_));
540         }
541         return  static_cast<Output*>(RTI_RoutingServiceOutput_get_user_data(native_output));
542
543     }
544
545 private:
546     RTI_RoutingServiceRoute *native_route_;
547     RTI_RoutingServiceEnvironment *native_env_;
548     std::string full_name_;
549
550 };
551
552
553 } } }
554
555 #endif // RTI_ROUTING_PROCESSOR_ROUTE_HPP_
```

9.36 SampleIterator.hpp

```
1 /* $Id$
2
3 (c) Copyright, Real-Time Innovations, 2014-2016.
4 All rights reserved.
5
6 No duplications, whole or partial, manual or electronic, may be made
7 without express written permission.  Any such copies, or
8 revisions thereof, must display this notice unaltered.
9 This code contains trade secrets of Real-Time Innovations, Inc.
10
11
12 modification history
13 --------------------
14 1.0a,05may14,acr Added ValidSampleIterator and valid_samples helper
15 1.0a,20mar13,acr Created
```

```
16 ============================================================================== */
17
18 #ifndef RTI_ROUTING_PROCESSOR_SAMPLE_ITERATOR_HPP_
19     #define RTI_ROUTING_PROCESSOR_SAMPLE_ITERATOR_HPP_
20
21
22     #include <rti/routing/processor/LoanedSample.hpp>
23     #include <exception>
24
25 namespace rti {
26 namespace routing {
27 namespace processor {
28
29 /*i
30  * @brief Tag type to emulate/indicate that sample infos are not available
31  */
32 struct no_sample_info_t {
33 };
34
43 template <typename T, typename U>
44 class SampleIterator {
45 public:
46     // iterator traits
47     using iterator_category = std::random_access_iterator_tag;
48     using value_type = const LoanedSample<T, U>;
49     using reference = value_type;
50     using pointer = value_type;
51     using difference_type = std::size_t;
52     using SampleSeqType = RTI_RoutingServiceSample*;
53     using InfoSeqType = RTI_RoutingServiceSampleInfo*;
54
55 private:
56     static no_sample_info_t *_null_info_scrachtpad[1];
57     static InfoSeqType _null_info_seq;
58     static difference_type _null_info_pos;
59
60 public:
61     SampleIterator()
62     : _sample_seq(0),
63     _info_seq(0),
64     _length(0),
65     _pos(0),
66     _info_pos(_pos)
67     {
68     }
69
70     SampleIterator(
71             const SampleSeqType sample_seq,
72             const InfoSeqType info_seq,
73             int length,
74             int position = 0)
75     : _sample_seq(sample_seq),
76     _info_seq(info_seq == NULL ? _null_info_seq : info_seq),
77     _length(length),
78     _info_pos(info_seq == NULL ? _null_info_pos : _pos)
79     {
80         if (position < 0) {
81             throw dds::core::IllegalOperationError("SampleIterator out of range");
82         }
83
84         _pos = static_cast<difference_type>(position);
85     }
86
87     SampleIterator(const SampleIterator<T, U> & other)
88     : _sample_seq(other._sample_seq),
89     _info_seq(other._info_seq),
90     _length(other._length),
91     _pos(other._pos),
92     _info_pos(other._info_pos)
93     {
94     }
95
96     SampleIterator & operator=(const SampleIterator<T, U> & other)
97     {
98         _sample_seq = other._sample_seq;
99         _info_seq = other._info_seq;
100         _length = other._length;
101         _pos = other._pos;
102         _info_pos = other._info_pos;
103
104         return *this;
```

```
105     }
106
107     value_type operator*() const
108     {
109         return value_type(_sample_seq[_pos], _info_seq[_info_pos]);
110     }
111
112     value_type operator->() const
113     {
114         return value_type(_sample_seq[_pos], _info_seq[_info_pos]);
115     }
116
117     value_type operator[](difference_type offset) const
118     {
119         return (_info_seq == _null_info_seq)
120                 ? value_type(_sample_seq[_pos], NULL)
121                 : value_type(_sample_seq[_pos + offset], _info_seq[_info_pos + offset]);
122     }
123
124 /*
125  * Windows defines min and max in one of the Windows headers, and they
126  * collide with the max() definition we use below
127  */
128 #ifdef RTI_WIN32
129 #undef max
130 #undef min
131 #endif
132     SampleIterator & operator++()
133     {
134         if (_pos == std::numeric_limits<difference_type>::max()) {
135             throw dds::core::IllegalOperationError("SampleIterator out of range");
136         }
137
138         ++_pos;
139         return *this;
140     }
141
142     SampleIterator operator++(int)
143     {
144         SampleIterator temp(*this);
145         ++(*this);
146         return temp;
147     }
148
149     SampleIterator & operator--()
150     {
151         if (_pos == std::numeric_limits<difference_type>::min()) {
152             throw dds::core::IllegalOperationError("SampleIterator out of range");
153         }
154
155         --_pos;
156         return *this;
157     }
158
159     SampleIterator operator--(int)
160     {
161         SampleIterator temp(*this);
162         --(*this);
163         return temp;
164     }
165
166     SampleIterator & operator-=(difference_type i)
167     {
168         if (i > _pos) {
169             throw dds::core::IllegalOperationError("SampleIterator out of range");
170         }
171
172         _pos -= i;
173         return *this;
174     }
175
176     SampleIterator & operator+=(difference_type i)
177     {
178         if ((std::numeric_limits<difference_type>::max() - _pos) < i) {
179             throw dds::core::IllegalOperationError("SampleIterator out of range");
180         }
181
182         _pos += i;
183         return *this;
184     }
185
```

```
186     friend SampleIterator operator-(
187             const SampleIterator& si,
188             difference_type i)
189     {
190         SampleIterator temp(si);
191         temp -= i;
192         return temp;
193     }
194 
195     friend SampleIterator operator+(
196             const SampleIterator& si,
197             difference_type i)
198     {
199         SampleIterator temp(si);
200         temp += i;
201         return temp;
202     }
203 
204     friend difference_type operator-(
205             const SampleIterator& s1,
206             const SampleIterator& s2)
207     {
208         return s1._pos - s2._pos;
209     }
210 
211     friend bool operator<(const SampleIterator & s1,
212             const SampleIterator & s2)
213     {
214         return s1._pos < s2._pos;
215     }
216 
217     friend bool operator>(const SampleIterator & s1,
218             const SampleIterator & s2)
219     {
220         return s1._pos > s2._pos;
221     }
222 
223     friend bool operator<=(const SampleIterator & s1,
224             const SampleIterator & s2)
225     {
226         return s1._pos <= s2._pos;
227     }
228 
229     friend bool operator>=(const SampleIterator & s1,
230             const SampleIterator & s2)
231     {
232         return s1._pos >= s2._pos;
233     }
234 
235     friend bool operator==(const SampleIterator & s1,
236             const SampleIterator & s2)
237     {
238         return (s1._sample_seq == s2._sample_seq) && (s1._pos == s2._pos);
239     }
240 
241     friend bool operator!=(const SampleIterator & s1,
242             const SampleIterator & s2)
243     {
244         return !(s1 == s2);
245     }
246 
247     bool is_end() const
248     {
249         return (_length == _pos);
250     }
251 
252     // Keep these members public otherwise some of the
253     // converting constructors will not compile.
254 public:
255     SampleSeqType _sample_seq;
256     InfoSeqType _info_seq;
257     int _length;
258     difference_type _pos;
259     difference_type& _info_pos;
260 };
261 
262 
263 template <typename T, typename U>
264 std::size_t SampleIterator<T, U>::_null_info_pos = 0;
265 
266 template <typename T, typename U>
```

```
267 no_sample_info_t * SampleIterator<T, U>::_null_info_scrachtpad[1] = {NULL};
268
269 template <typename T, typename U>
270 typename SampleIterator<T, U>::InfoSeqType SampleIterator<T, U>::_null_info_seq =
271 reinterpret_cast<InfoSeqType> (SampleIterator<T, U>::_null_info_scrachtpad);
272
273 } } }
274
275 #endif // RTI_ROUTING_PROCESSOR_SAMPLE_ITERATOR_HPP_
```

9.37 PropertySet.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_PROPERTY_SET_HPP_
12 #define RTI_ROUTING_PROPERTY_SET_HPP_
13
14 #include <map>
15
16 #include "routingservice/routingservice_infrastructure.h"
17 #include <rti/config/Version.hpp>
18
19 namespace rti { namespace routing {
20
21 inline
22 std::string app_name_property_name()
23 {
24     return RTI_ROUTING_SERVICE_APP_NAME_PROPERTY_NAME;
25 }
26
27 inline
28 std::string group_name_property_name()
29 {
30     return RTI_ROUTING_SERVICE_GROUP_PROPERTY_NAME;
31 }
32
33 inline
34 std::string version_property_name()
35 {
36     return RTI_ROUTING_SERVICE_VERSION_PROPERTY_NAME;
37 }
38
39 inline
40 std::string verbosity_property_name()
41 {
42     return RTI_ROUTING_SERVICE_VERBOSITY_PROPERTY_NAME;
43 }
44
45 inline
46 std::string entity_resource_name_property_name()
47 {
48     return RTI_ROUTING_SERVICE_ENTITY_RESOURCE_NAME_PROPERTY_NAME;
49 }
60 typedef std::map<std::string,std::string> PropertySet;
61
62 }}
63
64 #endif // RTI_ROUTING_PROPERTY_SET_HPP_
```

9.38 RoutingService.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2021.
3  * All rights reserved.
4  *
```

```
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ROUTING_SERVICE_HPP_
12 #define RTI_ROUTING_ROUTING_SERVICE_HPP_
13
14 #include <rti/routing/Service.hpp>
15
16
17 #endif // RTI_ROUTING_ROUTING_SERVICE_HPP_
```

9.39 Service.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_SERVICE_HPP_
12 #define RTI_ROUTING_SERVICE_HPP_
13
14 #include <dds/core/Reference.hpp>
15 #include <rti/config/Logger.hpp>
16
17 #include "routingservice/routingservice_adapter_new.h"
18
19 #include <rti/routing/Logger.hpp>
20 #include <rti/routing/ServiceProperty.hpp>
21 #include <rti/routing/detail/RoutingServiceImpl.hpp>
22 #include <rti/routing/adapter/AdapterPlugin.hpp>
23 #include <rti/routing/processor/ProcessorPlugin.hpp>
24
25 namespace rti { namespace routing {
26
27 /*e \defgroup RTI_RoutingServiceLibModule RTI Routing Library API
28  * \ingroup ROUTER
29  *
30  * @brief @product  can be deployed as a native library linked into your
31  * application in select architectures.
32  *
33  * This API allows you to create, configure and start @product instances from
34  * your application.
35  *
36  * The following code shows the typical use of the API:
37  *
38  * \code
39  *
40  * int main ()
41  * {
42  *     rti::routing::Service service(
43  *             rti::routing::ServiceProperty()
44  *             .cfg_file("MyRouter.xml")
45  *             .cfg_name("MyRouter"));
46  *     ...
47  *
48  *     service.start();
49  *
50  *     while(keep_running) {
51  *         sleep();
52  *         ...
53  *     }
54  *
55  *     return 0;
56  * }
57  *
58  * \endcode
59  *
60  * Instead of a file, you can use XML strings to configure @product.
61  * See ServiceProperty for more information.
```

```
62  * <p>
63  * To build your application you need to link with the @product native library
64  * in <b> &lt;@ndds home&gt;/bin/&lt;architecture&gt;/ </b>.
65  *
66  * ### Development Requirements
67  *
68  * |                 | Linux/macOS Systems                    | Windows Systems           |
69  * | --------------: | :--------------:                       | :-------------:           |
70  * | Shared Libraries| libnddscpp2.so                         | nddscpp2.dll              |
71  * | ^               | librtirsinfrastructure.so              | rtirsinfrastructure.dll   |
72  * | ^               | librtiserviceadmincpp.so               | rtiserviceadmincpp.dll    |
73  * | ^               | librtidlc.so                           | rtidlc.dll                |
74  * | ^               | librtiapputilsc.so                     | rtiapputilsc.dll          |
75  * | ^               | libnddsmetp.so                         | nddsmetp.dll              |
76  * | ^               | librticonnextmsgc.so                   | rticonnextmsgc.dll        |
77  * | ^               | libnddsc.so                            | nddsc.dll                 |
78  * | ^               | librtixml2.so                          | rtixml2.dll               |
79  * | ^               | libnddscore.so                         | nddscore.dll              |
80  * | Headers         | rti/routing/RoutingService.hpp         ||
81  *
82  */
83
89 class Service : public dds::core::Reference<RoutingServiceImpl> {
90 public:
91
92     typedef RTI::Service::Admin::CommandRequest CommandRequest;
93     typedef RTI::Service::Admin::CommandReply CommandReply;
94
95     typedef dds::core::Reference<RoutingServiceImpl> Base;
96     OMG_DDS_REF_TYPE_NOTYPENAME(
97             Service,
98             dds::core::Reference,
99             RoutingServiceImpl);
100
113     explicit Service(const ServiceProperty& property)
114             : Base(new RoutingServiceImpl(property))
115     {
116         this->delegate()->remember_reference(this->delegate());
117     }
118
170     template <typename HookFunc>
171     Service(
172             const ServiceProperty& property,
173             HookFunc& shutdown_hook)
174             : Base(new RoutingServiceImpl(property, shutdown_hook))
175     {
176         this->delegate()->remember_reference(this->delegate());
177     }
178
179     Service(const RTI_RoutingServiceProperty& property)
180             : Base(new RoutingServiceImpl(property))
181     {
182         this->delegate()->remember_reference(this->delegate());
183     }
184
185     explicit Service(Base::DELEGATE_REF_T reference) : Base(reference)
186     {
187         if (this->delegate()) {
188             this->delegate()->remember_reference(this->delegate());
189         }
190     }
191
198     void start()
199     {
200         this->delegate()->start();
201     }
202
208     void stop()
209     {
210         this->delegate()->stop();
211     }
212
275     void attach_adapter_plugin(
276             rti::routing::adapter::AdapterPlugin *adapter_plugin,
277             const std::string& plugin_name)
278     {
279         this->delegate()->attach_adapter_plugin(
280                 adapter_plugin,
281                 plugin_name);
282     }
283
```

```
290     void attach_processor_plugin(
291             rti::routing::processor::ProcessorPlugin *processor_plugin,
292             const std::string& plugin_name)
293     {
294         this->delegate()->attach_processor_plugin(
295                 processor_plugin,
296                 plugin_name);
297     }
298
305     void attach_transformation_plugin(
306             rti::routing::transf::TransformationPlugin *transformation_plugin,
307             const std::string& plugin_name)
308     {
309         this->delegate()->attach_transformation_plugin(
310                 transformation_plugin,
311                 plugin_name);
312     }
313
362     CommandReply execute_command(const CommandRequest& request)
363     {
364         return this->delegate()->execute_command(request);
365     }
366
379     CommandReply& execute_command(
380             CommandReply& reply,
381             const CommandRequest& request)
382     {
383         return this->delegate()->execute_command(reply, request);
384     }
385
390     static void finalize_globals()
391     {
392         rti::routing::Logger::instance().warn(
393                 "Calling 'rti::routing::Service::finalize_globals()' is no "
394                 "longer necessary and the function will be removed in future "
395                 "versions");
396     }
397
398 };
399
400 typedef Service RoutingService;
401
402 }}
403
404 #endif // RTI_ROUTING_SERVICE_HPP_
```

9.40 ServiceProperty.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_ROUTING_SERVICE_PROPERTY_HPP_
12 #define RTI_ROUTING_ROUTING_SERVICE_PROPERTY_HPP_
13
14 // IMPORTANT: macros.hpp must be the first RTI header included in every header
15 // file so that symbols are exported correctly on Windows
16 #include <dds/core/macros.hpp>
17 #include <rti/core/NativeValueType.hpp>
18
19 #include "routingservice/routingservice_service.h"
20 #include <rti/routing/detail/ForwarderUtils.hpp>
21
22 namespace rti { namespace routing {
23
24 class PropertyAdapter {
25 public:
26     typedef RTI_RoutingServiceProperty native_type;
27
28     static void initialize(native_type& native_value)
29     {
```

```
30          static const native_type DEFAULT
31                  = RTI_RoutingServiceProperty_INITIALIZER;
32          native_value = DEFAULT;
33          native_value.service_verbosity = -1;
34          native_value.dds_verbosity = -1;
35      }
36
37      static void finalize(native_type& native_value)
38      {
39          if (native_value.cfg_strings != NULL) {
40              RTIOsapiHeap_freeArray(native_value.cfg_strings);
41              native_value.cfg_strings = NULL;
42          }
43
44          RTI_RoutingServiceProperty_finalize(&native_value);
45      }
46
47      static void copy(native_type& destination, const native_type& source)
48      {
49          RTIOsapiUtility_unusedParameter(destination);
50          RTIOsapiUtility_unusedParameter(source);
51
52          throw dds::core::UnsupportedError("Unsupported operation: copy");
53      }
54
55      static bool equals(const native_type& first, const native_type& second)
56      {
57          RTIOsapiUtility_unusedParameter(first);
58          RTIOsapiUtility_unusedParameter(second);
59
60          throw dds::core::UnsupportedError("Unsupported operation: equals");
61      }
62
63      static void add_properties_from_native(
64              std::map<std::string, std::string>& properties,
65              const RTI_RoutingServiceProperties *native_properties)
66      {
67          for (int i = 0; i < native_properties->count; i++) {
68              properties[native_properties->properties[i].name] =
69                      (const char *) native_properties->properties[i].value;
70          }
71      }
72
73      static void add_properties_to_native(
74              RTI_RoutingServiceProperties *native_properties,
75              const std::map<std::string, std::string>& properties)
76      {
77          RTI_RoutingServiceProperties_finalize(native_properties);
78          RTI_RoutingServiceProperties_initialize(native_properties);
79
80          std::map<std::string, std::string>::const_iterator it = properties.begin();
81          for (; it != properties.end(); ++it) {
82              if (!RTI_RoutingServiceProperties_add(
83                      native_properties,
84                      it->first.c_str(),
85                      it->second.c_str())) {
86                  throw dds::core::Error("failed to add native property");
87              }
88          }
89      }
90
91 };
92
93 class ServiceProperty;
94
95 }
96
97 namespace core {
98
99 template<>
100 struct native_type_traits<rti::routing::ServiceProperty> {
101     typedef rti::routing::PropertyAdapter adapter_type;
102     typedef RTI_RoutingServiceProperty native_type;
103 };
104
105 }
106
107 namespace routing {
108
114 class ServiceProperty : public rti::core::NativeValueType<ServiceProperty> {
115 public:
```

```
116     RTI_NATIVE_VALUE_TYPE_DEFINE_DEFAULT_MOVE_OPERATIONS(ServiceProperty)
117
118     typedef rti::core::NativeValueType<ServiceProperty> Base;
119
123     ServiceProperty(): Base()
124     {
125     }
126
127     ServiceProperty(const struct RTI_RoutingServiceProperty &native_property)
128             : Base(native_property)
129     {
130     }
131
132
136     std::string cfg_file() const
137     {
138         return native().cfg_file == NULL ? "" : native().cfg_file;
139     }
140
146     ServiceProperty& cfg_file(const std::string& filename)
147     {
148         DDS_String_replace(
149                 &native().cfg_file,
150                 filename.c_str());
151         return *this;
152     }
153
157     const std::vector<std::string>& cfg_strings() const
158     {
159         return cfg_strings_;
160     }
161
176     ServiceProperty& cfg_strings(
177             const std::vector<std::string>& the_cfg_strings)
178     {
179         cfg_strings_ = the_cfg_strings;
180         if (cfg_strings_.size() > 0) {
181             if (!RTIOsapiHeap_reallocateArray(
182                     (char ***) &native().cfg_strings,
183                     cfg_strings_.size(),
184                     char *)) {
185                 throw dds::core::OutOfResourcesError("cfg_strings native array");
186             }
187             for (uint32_t i = 0; i < cfg_strings_.size(); i++) {
188             native().cfg_strings[i] = cfg_strings_[i].c_str();
189             }
190         }
191         native().cfg_strings_count = static_cast<int>(cfg_strings_.size());
192
193         return *this;
194     }
195
199     std::string service_name() const
200     {
201         return native().service_name == NULL ? "" : native().service_name;
202     }
203
214     ServiceProperty& service_name(const std::string& name)
215     {
216         DDS_String_replace(
217                 &native().service_name,
218                 name.c_str());
219         return *this;
220     }
221
225     std::string application_name() const
226     {
227         return native().application_name;
228     }
229
244     ServiceProperty& application_name(const std::string& name)
245     {
246         DDS_String_replace(
247                 &native().application_name,
248                 name.c_str());
249         return *this;
250     }
251
255     bool enforce_xsd_validation() const
256     {
257         return (native().enforce_xsd_validation == DDS_BOOLEAN_TRUE)
```

```
258                 ? true
259                 : false;
260     }
261
268     ServiceProperty& enforce_xsd_validation(const bool enforce_xsd_validation)
269     {
270         native().enforce_xsd_validation = enforce_xsd_validation;
271         return *this;
272     }
273
277     int domain_id_base() const
278     {
279         return native().domain_id_base;
280     }
281
282     /*e
283      * @brief Value that is added to the domain IDs of the
284      *        domain routes in the XML configuration.
285      *
286      * By using this, an XML file can use relative domain IDs.
287      *
288      * @default 0
289      */
290     ServiceProperty& domain_id_base(const int domain_id)
291     {
292         native().domain_id_base = domain_id;
293         return *this;
294     }
295
299     std::string plugin_search_path() const
300     {
301         return native().plugin_search_path == NULL
302                 ? ""
303                 : native().plugin_search_path;
304     }
305
315     ServiceProperty& plugin_search_path(const std::string& path)
316     {
317         DDS_String_replace(
318                 &native().plugin_search_path,
319                 path.c_str());
320         return *this;
321     }
322
326     bool dont_start_service() const
327     {
328         return (native().dont_start_service == DDS_BOOLEAN_TRUE)
329                 ? true
330                 : false;
331     }
332
341     ServiceProperty& dont_start_service(const bool not_start)
342     {
343         native().dont_start_service = not_start
344                 ? DDS_BOOLEAN_TRUE
345                 : DDS_BOOLEAN_FALSE;
346         return *this;
347     }
348
352     bool enable_administration() const
353     {
354         return (native().enable_administration == DDS_BOOLEAN_TRUE)
355                 ? true
356                 : false;
357     }
358
365     ServiceProperty& enable_administration(const bool enable)
366     {
367         native().enable_administration = enable
368                 ? DDS_BOOLEAN_TRUE
369                 : DDS_BOOLEAN_FALSE;
370         return *this;
371     }
372
376     int administration_domain_id() const
377     {
378         return native().administration_domain_id;
379     }
380
391     ServiceProperty& administration_domain_id(const int domain_id)
392     {
```

```
393         native().administration_domain_id = domain_id;
394         return *this;
395     }
396
400     bool enable_monitoring() const
401     {
402         return native().enable_monitoring == DDS_BOOLEAN_TRUE
403                 ? true
404                 : false;
405     }
406
413     ServiceProperty& enable_monitoring(const bool& enable)
414     {
415         native().enable_monitoring = enable
416                 ? DDS_BOOLEAN_TRUE
417                 : DDS_BOOLEAN_FALSE;
418         return *this;
419     }
420
424     int monitoring_domain_id() const
425     {
426         return native().monitoring_domain_id;
427     }
428
439     ServiceProperty& monitoring_domain_id(const int& domain_id)
440     {
441         native().monitoring_domain_id = domain_id;
442         return *this;
443     }
444
448     bool skip_default_files() const
449     {
450         return (native().skip_default_files == DDS_BOOLEAN_TRUE)
451                 ? true
452                 : false;
453     }
454
464     ServiceProperty& skip_default_files(const bool& skip)
465     {
466         native().skip_default_files = skip
467                 ? DDS_BOOLEAN_TRUE
468                 : DDS_BOOLEAN_FALSE;
469         return *this;
470     }
471
472
476     bool identify_execution() const
477     {
478         return (native().identify_execution == DDS_BOOLEAN_TRUE)
479                 ? true
480                 : false;
481     }
482
491     ServiceProperty& identify_execution(const bool& identify)
492     {
493         native().identify_execution = identify
494                 ? DDS_BOOLEAN_TRUE
495                 : DDS_BOOLEAN_FALSE;
496         return *this;
497     }
498
502     std::string license_file_name() const
503     {
504         return native().license_file_name == NULL? ""
505                 : native().license_file_name;
506     }
507
517     ServiceProperty& license_file_name(const std::string& filename)
518     {
519         DDS_String_replace(
520                 &native().license_file_name,
521                 filename.c_str());
522         return *this;
523     }
524
528     std::map<std::string, std::string> user_environment() const
529     {
530         std::map<std::string, std::string> properties;
531
532         PropertyAdapter::add_properties_from_native(
533                 properties,
```

```
534                 &native().user_environment);
535
536         return properties;
537     }
538
547     ServiceProperty& user_environment(
548             const std::map<std::string, std::string>& user_environment)
549     {
550         rti::routing::PropertyAdapter::add_properties_to_native(
551                 &native().user_environment,
552                 user_environment);
553
554         return *this;
555     }
556
557 private:
558     std::vector<std::string> cfg_strings_;
559 };
560
561 }}
562
563 #endif // RTI_ROUTING_ROUTING_SERVICE_PROPERTY_HPP_
```

9.41 StreamInfo.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_STREAM_INFO_HPP_
12 #define RTI_ROUTING_ADAPTER_STREAM_INFO_HPP_
13
14 // IMPORTANT: macros.hpp must be the first RTI header included in every header
15 // file so that symbols are exported correctly on Windows
16 #include <dds/core/macros.hpp>
17
18
19 #include <routingservice/routingservice_infrastructure.h>
20 #include <dds/core/Value.hpp>
21 #include <dds/core/Exception.hpp>
22 #include <rti/core/NativeValueType.hpp>
23 #include <rti/routing/TypeInfo.hpp>
24
25 namespace rti { namespace routing {
26
31 class StreamInfoAdapter {
32 public:
33     typedef RTI_RoutingServiceStreamInfo native_type;
34
35     static void initialize(native_type& native_value)
36     {
37         RTIOsapiMemory_zero(
38                 &native_value,
39                 sizeof (RTI_RoutingServiceStreamInfo));
40     }
41
42     static void finalize(native_type& native_value)
43     {
44         TypeInfoAdapter::finalize(native_value.type_info);
45         if (native_value.stream_name != NULL) {
46             DDS_String_free(native_value.stream_name);
47             native_value.stream_name = NULL;
48         }
49     }
50
51     static void copy(native_type& destination, const native_type& source)
52     {
53         char *result = DDS_String_replace(
54                 &destination.stream_name,
55                 source.stream_name);
56         if (source.stream_name != NULL && result == NULL) {
```

```
57             throw std::bad_alloc();
58         }
59         destination.disposed = source.disposed;
60         TypeInfoAdapter::copy(
61                 destination.type_info,
62                 source.type_info);
63     }
64
65     static bool equals(const native_type& first, const native_type& second)
66     {
67         if (strcmp(first.stream_name, second.stream_name) != 0) {
68             return false;
69         }
70         if (first.disposed != second.disposed) {
71             return false;
72         }
73         return TypeInfoAdapter::equals(first.type_info, second.type_info);
74     }
75
76 };
77
78 class StreamInfo;
79
80 }
81
82 // native_type_traits needs to be defined in rti::core
83 namespace core {
84
85 template <>
86 struct native_type_traits<rti::routing::StreamInfo> {
87     typedef rti::routing::StreamInfoAdapter adapter_type;
88     typedef RTI_RoutingServiceStreamInfo native_type;
89 };
90
91 }
92
93 namespace routing {
94
106 class StreamInfo : public rti::core::NativeValueType<StreamInfo> {
107 public:
108     RTI_NATIVE_VALUE_TYPE_DEFINE_DEFAULT_MOVE_OPERATIONS(StreamInfo)
109
110     typedef rti::core::NativeValueType<StreamInfo> Base;
111 public:
112
122     StreamInfo(
123             const std::string& the_stream_name,
124             const std::string& the_type_name)
125     {
126         stream_name(the_stream_name);
127         type_info().type_name(the_type_name);
128     }
129
130     StreamInfo(const RTI_RoutingServiceStreamInfo &native_stream_info)
131             : Base(native_stream_info)
132     {
133     }
134
138     bool disposed() const
139     {
140         return native().disposed ? true : false;
141     }
142
153     StreamInfo& disposed(bool the_disposed)
154     {
155         native().disposed = the_disposed ? 1 : 0;
156         return *this;
157     }
158
162     std::string stream_name() const
163     {
164         return native().stream_name;
165     }
166
173     StreamInfo& stream_name(const std::string& the_stream_name)
174     {
175         if (DDS_String_replace(
176                 &native().stream_name,
177                 the_stream_name.c_str()) == NULL) {
178             throw std::bad_alloc();
179         }
```

```
180
181         return *this;
182     }
183
187     TypeInfo& type_info()
188     {
189         using namespace rti::core::native_conversions;
190
191         return cast_from_native<TypeInfo>(native().type_info);
192     }
193
199     const TypeInfo& type_info() const
200     {
201         using namespace rti::core::native_conversions;
202
203         return cast_from_native<TypeInfo>(native().type_info);
204     }
205
206 };
207
208 }}
209
210 #endif // RTI_ROUTING_ADAPTER_STREAM_INFO_HPP_
```

9.42 TransformationForwarder.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_TRANSF_DETAIL_TRANSFORMATION_FORWARDER_HPP_
12 #define RTI_ROUTING_TRANSF_DETAIL_TRANSFORMATION_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <dds/core/xtypes/DynamicData.hpp>
17 #include <dds/sub/SampleInfo.hpp>
18 #include <rti/routing/transf/Transformation.hpp>
19 #include <rti/routing/TypeInfo.hpp>
20 #include <rti/routing/detail/UpdatableEntityForwarder.hpp>
21 #include <rti/routing/detail/ForwarderUtils.hpp>
22
23
24 namespace rti { namespace routing { namespace transf { namespace detail {
25
26
27 class TransformationForwarder{
28 public:
29     static RTI_RoutingServiceTransformation create_native(
30             TransformationPlugin *plugin,
31             const struct RTI_RoutingServiceTypeInfo *native_input_type_info,
32             const struct RTI_RoutingServiceTypeInfo *native_output_type_info,
33             const struct RTI_RoutingServiceProperties *native_properties,
34             RTI_RoutingServiceEnvironment *environment)
35     {
36         try {
37             using rti::routing::detail::ScopedForwarder;
38
39             TypeInfo input_type_info(*native_input_type_info);
40             TypeInfo output_type_info(*native_output_type_info);
41
42             std::map<std::string, std::string> properties;
43             rti::routing::PropertyAdapter::add_properties_from_native(
44                     properties,
45                     native_properties);
46
47             TransformationForwarder *forwarder = new TransformationForwarder();
48             ScopedForwarder<TransformationPlugin, TransformationForwarder> scoped(
49                     plugin,
50                     forwarder,
51                     environment);
```

```
52              forwarder->transformation_ = plugin->create_transformation(
53                      input_type_info,
54                      output_type_info,
55                      properties);
56              RTI_ROUTING_THROW_ON_NULL(forwarder->transformation_);
57
58              scoped.release();
59              return forwarder->native();
60          } catch(const std::exception& ex) {
61              RTI_RoutingServiceEnvironment_set_error(
62                      environment,
63                      "%s",
64                      ex.what());
65              return NULL;
66          } catch (...) {
67              RTI_RoutingServiceEnvironment_set_error(
68                      environment,
69                      "unexpected exception");
70              return NULL;
71          }
72
73      }
74
75      static void delete_native(
76              TransformationPlugin *plugin,
77              RTI_RoutingServiceTransformation native_transformation,
78              RTI_RoutingServiceEnvironment *environment)
79      {
80          TransformationForwarder *forwarder =
81                  static_cast<TransformationForwarder*>(native_transformation);
82          try {
83              if (forwarder->transformation_ != NULL) {
84                  plugin->delete_transformation(
85                          forwarder->transformation_);
86                  forwarder->transformation_ = NULL;
87              }
88          } catch(const std::exception& ex) {
89              RTI_RoutingServiceEnvironment_set_error(
90                      environment,
91                      "%s",
92                      ex.what());
93          } catch (...) {
94              RTI_RoutingServiceEnvironment_set_error(
95                      environment,
96                      "unexpected exception");
97          }
98
99          delete forwarder;
100     }
101
102     static void transform(
103             RTI_RoutingServiceTransformation native_transformation,
104             RTI_RoutingServiceSample **out_sample_lst,
105             RTI_RoutingServiceSampleInfo **out_info_lst,
106             int *out_count,
107             RTI_RoutingServiceSample *in_sample_lst,
108             RTI_RoutingServiceSampleInfo *in_info_lst,
109             int in_count,
110             RTI_RoutingServiceEnvironment *environment)
111     {
112         TransformationForwarder *forwarder =
113                 static_cast<TransformationForwarder*> (native_transformation);
114         if (forwarder->transformation_ == NULL) {
115             return;
116         }
117
118         *out_count = 0;
119         try {
120             // get native samples
121             forwarder->in_sample_seq_.resize(in_count);
122             forwarder->in_info_seq_.resize(in_count);
123             for (int i = 0; i < in_count; i++) {
124                 forwarder->in_sample_seq_[i] = in_sample_lst[i];
125                 if (in_info_lst != NULL) {
126                     forwarder->in_info_seq_[i] = in_info_lst[i];
127                 }
128             }
129
130             forwarder->transformation_->transform(
131                     forwarder->out_sample_seq_,
132                     forwarder->out_info_seq_,
```

```
133                     forwarder->in_sample_seq_,
134                     forwarder->in_info_seq_);
135             if (!forwarder->out_info_seq_.empty()
136                     && (forwarder->out_info_seq_.size()
137                             != forwarder->out_sample_seq_.size())) {
138                 throw dds::core::PreconditionNotMetError(
139                         "sample and info sequences length mismatch");
140             }
141         } catch (const std::exception& ex) {
142             RTI_RoutingServiceEnvironment_set_error(
143                     environment,
144                     "%s",
145                     ex.what());
146             return;
147         } catch (...) {
148             RTI_RoutingServiceEnvironment_set_error(
149                     environment,
150                     "unexpected exception");
151             return;
152         }
153 
154         // direct mapping
155         *out_count = static_cast<int>(forwarder->out_sample_seq_.size());
156         if (*out_count > 0) {
157             *out_sample_lst = &forwarder->out_sample_seq_[0];
158             *out_info_lst = &forwarder->out_info_seq_[0];
159         }
160     }
161 
162     static void return_loan(
163         RTI_RoutingServiceTransformation native_transformation,
164         RTI_RoutingServiceSample *,
165         RTI_RoutingServiceSampleInfo *,
166         int,
167         RTI_RoutingServiceEnvironment *environment)
168     {
169         TransformationForwarder *forwarder =
170                 static_cast<TransformationForwarder*>(native_transformation);
171         if (forwarder->transformation_ == NULL) {
172             return;
173         }
174 
175         try {
176             forwarder->transformation_->return_loan(
177                     forwarder->out_sample_seq_,
178                     forwarder->out_info_seq_);
179         } catch (const std::exception& ex) {
180             RTI_RoutingServiceEnvironment_set_error(
181                     environment,
182                     "%s",
183                     ex.what());
184         } catch (...) {
185             RTI_RoutingServiceEnvironment_set_error(
186                     environment,
187                     "unexpected exception");
188         }
189 
190         forwarder->out_sample_seq_.clear();
191         forwarder->out_info_seq_.clear();
192 
193     }
194 
195     static void update(
196             RTI_RoutingServiceTransformation native_stream_reader,
197             const struct RTI_RoutingServiceProperties *native_properties,
198             RTI_RoutingServiceEnvironment *environment)
199     {
200 
201         TransformationForwarder *stream_reader_forwarder =
202                 static_cast<TransformationForwarder*> (native_stream_reader);
203 
204         rti::routing::detail::UpdatableEntityForwarder::update(
205                 stream_reader_forwarder->transformation_,
206                 native_properties,
207                 environment);
208     }
209 
210     RTI_RoutingServiceTransformation native()
211     {
212         return this;
213     }
```

```
214
215
216 private:
217
218     TransformationForwarder()
219         : transformation_(NULL)
220     {
221
222     }
223
224     ~TransformationForwarder()
225     {
226     }
227
228
229 private:
230
231     Transformation *transformation_;
232     std::vector<Transformation::SamplePtr> out_sample_seq_;
233     std::vector<Transformation::InfoPtr> out_info_seq_;
234     std::vector<Transformation::SamplePtr> in_sample_seq_;
235     std::vector<Transformation::InfoPtr> in_info_seq_;
236 };
237
238 }}}}
239
240 #endif // RTI_ROUTING_TRANSF_DETAIL_TRANSFORMATION_FORWARDER_HPP_
```

9.43 TransformationPluginForwarder.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_TRANSF_DETAIL_TRANSFORMATION_PLUGIN_FORWARDER_HPP_
12 #define RTI_ROUTING_TRANSF_DETAIL_TRANSFORMATION_PLUGIN_FORWARDER_HPP_
13
14 #include <rti/core/Exception.hpp>
15
16 #include <routingservice/routingservice_transformation.h>
17
18 #include <rti/routing/transf/detail/TransformationForwarder.hpp>
19
20 namespace rti { namespace routing { namespace transf { namespace detail {
21
22
23 class TransformationPluginForwarder {
24 public:
25     static RTI_RoutingServiceTransformationPlugin * create_plugin(
26             TransformationPlugin *transformation_plugin)
27     {
28         RTI_RoutingServiceTransformationPlugin *native_plugin = NULL;
29         RTIOsapiHeap_allocateStructure(
30                 &native_plugin,
31                 struct RTI_RoutingServiceTransformationPlugin);
32         rti::core::check_create_entity(
33                 native_plugin,
34                 "RTI_RoutingServiceTransformationPlugin");
35         RTI_RoutingServiceTransformationPlugin_initialize(native_plugin);
36
37         // Set adapter version
38         rti::config::LibraryVersion version =  transformation_plugin->get_version();
39         native_plugin->plugin_version.major = version.major_version();
40         native_plugin->plugin_version.minor = version.minor_version();
41         native_plugin->plugin_version.release = version.release_version();
42
43         // Initiallize adapter methods
44         native_plugin->user_object = static_cast<void *>(transformation_plugin);
45         native_plugin->transformation_plugin_delete =
46                 TransformationPluginForwarder::delete_plugin;
47         native_plugin->transformation_plugin_create_transformation =
```

```
48                 TransformationPluginForwarder::create_transformation;
49         native_plugin->transformation_plugin_delete_transformation =
50                 TransformationPluginForwarder::delete_transformation;
51         native_plugin->transformation_transform =
52                 TransformationForwarder::transform;
53         native_plugin->transformation_return_loan =
54                 TransformationForwarder::return_loan;
55         native_plugin->transformation_update =
56                 TransformationForwarder::update;
57 
58         return native_plugin;
59     }
60 
61     static void delete_plugin(
62         RTI_RoutingServiceTransformationPlugin *native_plugin,
63         RTI_RoutingServiceEnvironment *)
64     {
65         TransformationPlugin *plugin = static_cast<TransformationPlugin *>(
66                 native_plugin->user_object);
67         // Plug-in is destructor not allowed to throw
68         delete plugin;
69         RTIOsapiHeap_freeStructure(native_plugin);
70     }
71 
72     static RTI_RoutingServiceTransformation create_transformation(
73             struct RTI_RoutingServiceTransformationPlugin  *native_plugin,
74             const struct RTI_RoutingServiceTypeInfo *input_type_info,
75             const struct RTI_RoutingServiceTypeInfo *output_type_info,
76             const struct RTI_RoutingServiceProperties *native_properties,
77             RTI_RoutingServiceEnvironment  *environment)
78     {
79         return TransformationForwarder::create_native(
80                 static_cast<TransformationPlugin *>(native_plugin->user_object),
81                 input_type_info,
82                 output_type_info,
83                 native_properties,
84                 environment);
85 
86     }
87 
88     static void delete_transformation(
89             struct RTI_RoutingServiceTransformationPlugin *native_plugin,
90             RTI_RoutingServiceTransformation native_transformation,
91             RTI_RoutingServiceEnvironment *environment)
92     {
93         TransformationForwarder::delete_native(
94                 static_cast<TransformationPlugin *>(native_plugin->user_object),
95                 native_transformation,
96                 environment);
97     }
98 
99 };
100 
101 
102 }}}}
103 
104 #endif // RTI_ROUTING_TRANSF_DETAIL_TRANSFORMATION_PLUGIN_FORWARDER_HPP_
```

9.44 Transformation.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10 
11 #ifndef RTI_ROUTING_TRANSF_TRANSFORMATION_HPP_
12 #define RTI_ROUTING_TRANSF_TRANSFORMATION_HPP_
13 
14 #include <vector>
15 
16 #include <dds/core/xtypes/DynamicData.hpp>
17 #include <dds/sub/SampleInfo.hpp>
```

```
18 #include <rti/routing/UpdatableEntity.hpp>
19 #include <rti/routing/detail/ForwarderUtils.hpp>
20
21 namespace rti { namespace routing { namespace transf {
22
35 class Transformation : public UpdatableEntity{
36 public:
37
38     typedef void* SamplePtr;
39     typedef void* InfoPtr;
40
80     virtual void transform(
81             std::vector<SamplePtr>& output_sample_seq,
82             std::vector<InfoPtr>& output_info_seq,
83             const std::vector<SamplePtr>& input_sample_seq,
84             const std::vector<InfoPtr>& input_info_seq) = 0;
85
101     virtual void return_loan(
102             std::vector<SamplePtr>& sample_seq,
103             std::vector<InfoPtr>& info_seq) = 0;
104
105
106     virtual ~Transformation() {}
107 };
108
109
121 template <typename Data, typename Info>
122 class TypedTransformation : public Transformation {
123
124 public:
125
129     typedef Data DataRep;
133     typedef Info InfoRep;
134
135     using Transformation::transform;
136     using Transformation::return_loan;
137
142     void transform(
143             std::vector<SamplePtr>& output_sample_seq,
144             std::vector<InfoPtr>& output_info_seq,
145             const std::vector<SamplePtr>& input_sample_seq,
146             const std::vector<InfoPtr>& input_info_seq) RTI_FINAL
147     {
148         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(in_sample_seq_, input_sample_seq);
149             RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(in_info_seq_, input_info_seq);
150             transform(
151                     out_sample_seq_,
152                     out_info_seq_,
153                     in_sample_seq_,
154                     in_info_seq_);
155         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(output_sample_seq, out_sample_seq_);
156         RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(output_info_seq, out_info_seq_);
157     }
158
163     void return_loan(
164             std::vector<SamplePtr>& sample_seq,
165             std::vector<InfoPtr>& info_seq) RTI_FINAL
166     {
167             RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(out_sample_seq_, sample_seq);
168             RTI_ROUTING_SAMPLE_VECTOR_COPY_PTRS(out_info_seq_, info_seq);
169             return_loan(out_sample_seq_, out_info_seq_);
170         out_sample_seq_.clear();
171         out_info_seq_.clear();
172     }
173
179     virtual void transform(
180             std::vector<Data*>& output_sample_seq,
181             std::vector<Info*>& output_info_seq,
182             const std::vector<Data*>& input_sample_seq,
183             const std::vector<Info*>& input_info_seq) = 0;
184
190     virtual void return_loan(
191             std::vector<Data*>& sample_seq,
192             std::vector<Info*>& info_seq) = 0;
193
197     virtual ~TypedTransformation()
198     {
199
200     }
201 private:
202     std::vector<Data*> in_sample_seq_;
```

```
203     std::vector<Info*> in_info_seq_;
204     std::vector<Data*> out_sample_seq_;
205     std::vector<Info*> out_info_seq_;
206
207 };
208
216 typedef TypedTransformation<dds::core::xtypes::DynamicData, dds::sub::SampleInfo>
217         DynamicDataTransformation;
218
219
220 }}}
221
222
223 #endif // RTI_ROUTING_TRANSF_TRANSFORMATION_HPP_
```

9.45 TransformationPlugin.hpp File Reference

RTI Routing Service Transformation API.

```
#include "routingservice/routingservice_log.h"
#include <rti/config/Version.hpp>
#include <rti/routing/TypeInfo.hpp>
#include <rti/routing/PropertySet.hpp>
#include <rti/routing/transf/Transformation.hpp>
#include <rti/routing/detail/ForwarderUtils.hpp>
#include <rti/routing/transf/detail/TransformationPluginForwarder.hpp>
```

Data Structures

- class **rti::routing::transf::TransformationPlugin**

  The top-level plug-in class.

Macros

- #define **RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DECL**(TRANSFORMATION_PLUGIN_↩ CLASS)

  Utility macro that declares a native extern function that can be used to load a TransformationPlugin through a shared library.
- #define **RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DEF**(TRANSFORMATION_PLUGIN_↩ CLASS)

  Utility macro that implements the TransformationPlugin entry point declared with RTI_TRANSFORMATION_PLUGIN_↩ CREATE_FUNCTION_DECL.

9.45.1 Detailed Description

RTI Routing Service Transformation API.

9.46 TransformationPlugin.hpp

Go to the documentation of this file.

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_TRANSF_TRANSFORMATION_PLUGIN_HPP_
12 #define RTI_ROUTING_TRANSF_TRANSFORMATION_PLUGIN_HPP_
13
14 #include "routingservice/routingservice_log.h"
15
16 #include <rti/config/Version.hpp>
17 #include <rti/routing/TypeInfo.hpp>
18 #include <rti/routing/PropertySet.hpp>
19 #include <rti/routing/transf/Transformation.hpp>
20 #include <rti/routing/detail/ForwarderUtils.hpp>
21
22 /*e \file
23   @brief RTI Routing Service Transformation API
24 */
34 namespace rti { namespace routing { namespace transf {
35
44 class TransformationPlugin {
45 public:
46
72     virtual Transformation * create_transformation(
73             const rti::routing::TypeInfo& input_type_info,
74             const rti::routing::TypeInfo& output_type_info,
75             const rti::routing::PropertySet& properties) = 0;
76
89     virtual void delete_transformation(Transformation *transformation) = 0;
90
100     virtual rti::config::LibraryVersion get_version() const
101     {
102         return rti::config::LibraryVersion();
103     }
104
108     virtual ~TransformationPlugin() {}
109 };
110
111 }}}
112
113 #include <rti/routing/transf/detail/TransformationPluginForwarder.hpp>
114
184 #define RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DECL(TRANSFORMATION_PLUGIN_CLASS) \
185 extern "C" RTI_USER_DLL_EXPORT struct RTI_RoutingServiceTransformationPlugin * \
186     TRANSFORMATION_PLUGIN_CLASS ## _create_transformation_plugin(\
187         const struct RTI_RoutingServiceProperties *, \
188         RTI_RoutingServiceEnvironment *); \
189
195 #define RTI_TRANSFORMATION_PLUGIN_CREATE_FUNCTION_DEF(TRANSFORMATION_PLUGIN_CLASS) \
196 struct RTI_RoutingServiceTransformationPlugin * \
197     TRANSFORMATION_PLUGIN_CLASS ## _create_transformation_plugin( \
198         const struct RTI_RoutingServiceProperties * native_properties, \
199         RTI_RoutingServiceEnvironment *environment) \
200 { \
201     rti::routing::PropertySet properties; \
202     rti::routing::PropertyAdapter::add_properties_from_native(\
203             properties,\
204             native_properties); \
205     try { \
206         return rti::routing::transf::detail::TransformationPluginForwarder::create_plugin(\
207             new TRANSFORMATION_PLUGIN_CLASS(properties)); \
208     } catch (const std::exception& ex) {\
209         RTI_RoutingServiceEnvironment_set_error(\
210                 environment,\
211                 "%s",\
212                 ex.what());\
213     } catch (...) {}\
214     \
215     return NULL; \
216 }
```

```
217
218
219
220
221 #endif // RTI_ROUTING_TRANSF_TRANSFORMATION_PLUGIN_HPP_
```

9.47 TypeInfo.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_ROUTING_ADAPTER_TYPE_INFOHPP_
12 #define RTI_ROUTING_ADAPTER_TYPE_INFOHPP_
13
14 // IMPORTANT: macros.hpp must be the first RTI header included in every header
15 // file so that symbols are exported correctly on Windows
16 #include <dds/core/macros.hpp>
17
18 #include <routingservice/routingservice_infrastructure.h>
19 #include <dds/core/Value.hpp>
20 #include <dds/core/SafeEnumeration.hpp>
21 #include <dds/core/xtypes/DynamicType.hpp>
22 #include <rti/core/NativeValueType.hpp>
23
24 namespace rti { namespace routing {
25
41 struct TypeRepresentationKind_def {
45     enum type {
54         DYNAMIC_TYPE = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_DYNAMIC_TYPE,
64         XML = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_XML,
70         JAVA_OBJECT = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_JAVA_OBJECT,
77         CUSTOM = RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_FIRST_CUSTOM_REPRESENTATION
78     };
79 };
80
89 typedef dds::core::safe_enum<TypeRepresentationKind_def> TypeRepresentationKind;
90
91
92 class TypeInfoAdapter {
93 public:
94     typedef RTI_RoutingServiceTypeInfo native_type;
95
96     static void initialize(native_type& native_value)
97     {
98         static const native_type default_value = {
99             NULL,
100             RTI_ROUTING_SERVICE_TYPE_REPRESENTATION_DYNAMIC_TYPE,
101             NULL};
102         native_value = default_value;
103     }
104
105     static void finalize(native_type& native_value)
106     {
107         if (native_value.type_name != NULL) {
108             RTIOsapiHeap_freeString(native_value.type_name);
109             native_value.type_name = NULL;
110         }
111     }
112
113     static void copy(native_type& destination, const native_type& source)
114     {
115         char *result = DDS_String_replace(
116                 &destination.type_name,
117                 source.type_name);
118         if (source.type_name != NULL && result == NULL) {
119             throw std::bad_alloc();
120         }
121         destination.type_representation_kind =
122                 source.type_representation_kind;
123         destination.type_representation =
```

```
124                 source.type_representation;
125     }
126
127     static bool equals(const native_type& first, const native_type& second)
128     {
129         if (strcmp(first.type_name, second.type_name) != 0) {
130             return false;
131         }
132         if (first.type_representation_kind
133                 != second.type_representation_kind) {
134             return false;
135         }
136
137         return first.type_representation == second.type_representation;
138     }
139
140 };
141
142 class TypeInfo;
143
144 }
145
146 // native_type_traits needs to be defined in rti::core
147 namespace core {
148
149 template <>
150 struct native_type_traits<rti::routing::TypeInfo> {
151     typedef rti::routing::TypeInfoAdapter adapter_type;
152     typedef RTI_RoutingServiceTypeInfo native_type;
153 };
154
155 }
156
157 namespace routing {
158
169 class TypeInfo : public rti::core::NativeValueType<TypeInfo> {
170 public:
171     RTI_NATIVE_VALUE_TYPE_DEFINE_DEFAULT_MOVE_OPERATIONS(TypeInfo)
172
173     typedef rti::core::NativeValueType<TypeInfo> Base;
174
180     typedef RTI_RoutingServiceTypeRepresentation TypeRepresentationType;
181 public:
182
192     TypeInfo(
193             const std::string& the_type_name,
194             TypeRepresentationKind the_type_representation_kind =
195                     TypeRepresentationKind::DYNAMIC_TYPE)
196     {
197         type_name(the_type_name);
198         type_representation_kind(the_type_representation_kind);
199     }
200
201     TypeInfo(const struct RTI_RoutingServiceTypeInfo &native_stream_info)
202             : Base(native_stream_info)
203     {
204     }
205
209     std::string type_name() const
210     {
211         return native().type_name;
212     }
213
223     TypeInfo& type_name(const std::string& the_type_name)
224     {
225
226         if (DDS_String_replace(
227                 &native().type_name,
228                 the_type_name.c_str()) == NULL) {
229             throw std::bad_alloc();
230         }
231
232         return *this;
233     }
234
238     TypeRepresentationKind type_representation_kind() const
239     {
240         return static_cast<TypeRepresentationKind::type>(
241                 native().type_representation_kind);
242     }
243
```

```
251     TypeInfo& type_representation_kind(
252             TypeRepresentationKind the_type_representation_kind)
253     {
254         native().type_representation_kind = static_cast<int>(
255                 the_type_representation_kind.underlying());
256         return *this;
257     }
258
262     TypeRepresentationType  type_representation() const
263     {
264         return native().type_representation;
265     }
266
275     TypeInfo&  type_representation(TypeRepresentationType the_type_representation)
276     {
277         native().type_representation = the_type_representation;
278         return *this;
279     }
280
287     const dds::core::xtypes::DynamicType& dynamic_type() const
288     {
289         if (type_representation_kind() != TypeRepresentationKind::DYNAMIC_TYPE) {
290             throw dds::core::IllegalOperationError(
291                     "type representation is not dynamic type");
292         }
293
294         const DDS_TypeCode *native_typecode = reinterpret_cast<DDS_TypeCode*>(
295                 type_representation());
296         return rti::core::native_conversions::cast_from_native<
297                 dds::core::xtypes::DynamicType>(*native_typecode);
298     }
299
308     void dynamic_type(const dds::core::xtypes::DynamicType *dynamic_type)
309     {
310         DDS_TypeCode *native_typecode = const_cast<DDS_TypeCode*>(
311                 reinterpret_cast<const DDS_TypeCode *>(dynamic_type));
312         type_representation(static_cast<TypeRepresentationType>(native_typecode));
313         type_representation_kind(TypeRepresentationKind::DYNAMIC_TYPE);
314     }
315 };
316
317 }}
318
319 #endif // RTI_ROUTING_ADAPTER_TYPE_INFOHPP_
```

9.48 UpdatableEntity.hpp

```
1  /*
2   * (c) Copyright, Real-Time Innovations, 2017.
3   * All rights reserved.
4   *
5   * No duplications, whole or partial, manual or electronic, may be made
6   * without express written permission.  Any such copies, or
7   * revisions thereof, must display this notice unaltered.
8   * This code contains trade secrets of Real-Time Innovations, Inc.
9   */
10
11 #ifndef RTI_ROUTING_UPDATABLE_ENTITY_HPP_
12 #define RTI_ROUTING_UPDATABLE_ENTITY_HPP_
13
14 #include <map>
15 #include <string>
16
17 namespace rti { namespace routing {
18
34 class UpdatableEntity {
35 public:
36
45     virtual void update(const std::map<std::string, std::string>& properties)
46     {
47         (void) properties;
48     }
49
53     virtual ~UpdatableEntity()
54     {
55     }
56 };
```

```
57
58 }}
59
60 #endif // RTI_ROUTING_UPDATABLE_ENTITY_HPP_
```

9.49 RequestInterceptor.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_SERVICE_REQUEST_INTERCEPTOR_HPP_
12 #define RTI_SERVICE_REQUEST_INTERCEPTOR_HPP_
13
14 #include "routingservice/routingservice_remote_config.h"
15
16 #include "rti/apputils/log/LogConfig.hpp"
17
18 namespace rti { namespace service { namespace admin {
19
20 /*
21  * @brief string representation of EntityStateKind
22  */
23 inline
24 const char* to_native_string(const RTI_Service_Admin_CommandActionKind command_kind)
25 {
26     switch (command_kind) {
27     case RTI_SERVICE_COMMAND_ACTION_UPDATE:
28         return "UPDATE";
29     case RTI_SERVICE_COMMAND_ACTION_GET:
30         return "GET";
31     case RTI_SERVICE_COMMAND_ACTION_CREATE:
32         return "CREATE";
33     case RTI_SERVICE_COMMAND_ACTION_DELETE:
34         return "DELETE";
35     default:
36         return "";
37     }
38 }
39
40 /*
41  * @brief string representation of EntityStateKind
42  */
43 inline
44 const std::string to_string(const RTI_Service_Admin_CommandActionKind command_kind)
45 {
46     switch (command_kind) {
47     case RTI_SERVICE_COMMAND_ACTION_UPDATE:
48         return "UPDATE";
49     case RTI_SERVICE_COMMAND_ACTION_GET:
50         return "GET";
51     case RTI_SERVICE_COMMAND_ACTION_CREATE:
52         return "CREATE";
53     case RTI_SERVICE_COMMAND_ACTION_DELETE:
54         return "DELETE";
55     default:
56         std::ostringstream string_stream;
57         string_stream « (int) command_kind;
58         return string_stream.str();
59     }
60 }
61
62 class RequestInterceptor {
63 public:
64
65     virtual void on_command(
66             const RTI_Service_Admin_CommandRequest& incoming_request,
67             const DDS_SampleInfo& request_info,
68             RTI_RoutingServiceAdminReplier& replier) = 0;
69
70     virtual ~RequestInterceptor() {}
```

```
71
72 };
73
74 class RequestInterceptorForwarder  {
75 public:
76
77     typedef RTI_RoutingServiceConfigInterceptor native;
78
79     static native create(RequestInterceptor *interceptor)
80     {
81         native native_interceptor;
82         native_interceptor.on_command_request = on_request_forwarder;
83         native_interceptor.interceptor_data = static_cast<void*>(interceptor);
84
85         return native_interceptor;
86     }
87
88 private:
89     static void on_request_forwarder(
90             void *interceptor_data,
91             const RTI_Service_Admin_CommandRequest *incoming_request,
92             const DDS_SampleInfo *request_info,
93             RTI_RoutingServiceAdminReplier *replier)
94     {
95         RequestInterceptor *interceptor =
96                 static_cast<RequestInterceptor *>(interceptor_data);
97         try {
98             interceptor->on_command(*incoming_request, *request_info, *replier);
99         } catch (rti::apputils::ServiceException& ex) {
100             SERVICELog_fromException(ex);
101         } catch (...) {
102             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
103         }
104     }
105
106 };
107
108 }}}
109
110
111 #endif // RTI_SERVICE_REQUEST_INTERCEPTOR_HPP_
```

9.50 Monitorable.hpp

```
1 /*
2  * (c) Copyright, Real-Time Innovations, 2017.
3  * All rights reserved.
4  *
5  * No duplications, whole or partial, manual or electronic, may be made
6  * without express written permission.  Any such copies, or
7  * revisions thereof, must display this notice unaltered.
8  * This code contains trade secrets of Real-Time Innovations, Inc.
9  */
10
11 #ifndef RTI_SERVICE_MONITORING_MONITORABLE_HPP_
12 #define RTI_SERVICE_MONITORING_MONITORABLE_HPP_
13
14 #include "routingservice/routingservice_service_impl.h"
15
16 #include <dds/core/Duration.hpp>
17 #include <rti/core/detail/NativeConversion.hpp>
18 #include <rti/apputils/ServiceException.hpp>
19 #include <rti/apputils/log/LogConfig.hpp>
20 #include <rti/routing/RoutingService.hpp>
21
22 namespace rti { namespace routing { namespace monitoring {
23
24 class Monitorable {
25 public:
26
27     virtual void publish_status(
28             const dds::core::Duration& now)
29     {
30         RTIOsapiUtility_unusedParameter(now);
31     }
32
33     virtual void sample_status(
```

```
34              const dds::core::Duration& now)
35      {
36          RTIOsapiUtility_unusedParameter(now);
37      }
38
39      virtual ~Monitorable()
40      {
41      }
42  };
43
44  class MonitorableWrapper : public Monitorable {
45  public:
46      MonitorableWrapper()
47      {
48      }
49
50      void native(RTI_RoutingServiceMonitorable *native_monitorable)
51      {
52          native_monitorable_ = native_monitorable;
53      }
54
55      RTI_RoutingServiceMonitorable * native()
56      {
57          return native_monitorable_;
58      }
59
60      void publish_status(
61              const dds::core::Duration& now)
62      {
63          DDS_Duration_t native_now;
64          rti::core::native_conversions::to_native(native_now, now);
65          RTINtpTime ntp_now;
66          DDS_Duration_to_ntp_time(&native_now, &ntp_now);
67          native_monitorable_->publish_status(
68                  native_monitorable_->monitorable_data,
69                  &ntp_now);
70
71      }
72
73      virtual void sample_status(
74              const dds::core::Duration& now)
75      {
76          DDS_Duration_t native_now;
77          rti::core::native_conversions::to_native(native_now, now);
78          RTINtpTime ntp_now;
79          DDS_Duration_to_ntp_time(&native_now, &ntp_now);
80          native_monitorable_->sample_status(
81                  native_monitorable_->monitorable_data,
82                  &ntp_now);
83      }
84
85
86  private:
87      RTI_RoutingServiceMonitorable *native_monitorable_;
88  };
89
90  class MonitorableForwarder  {
91  public:
92
93      typedef RTI_RoutingServiceMonitorable native;
94
95      static native create(
96              Monitorable *monitorable)
97      {
98          native native_monitorable;
99          native_monitorable.publish_status =
100                 publish_status;
101         native_monitorable.sample_status =
102                 sample_status;
103         native_monitorable.monitorable_data =
104                 static_cast<void*>(monitorable);
105
106         return native_monitorable;
107     }
108
109 private:
110     static void publish_status(
111             void *monitorable_data,
112             const struct RTINtpTime *now)
113     {
114         Monitorable *monitorable =
```

```
115                 static_cast<Monitorable *>(monitorable_data);
116         try {
117             DDS_Duration_t native_duration = DDS_DURATION_ZERO;
118             DDS_Duration_from_ntp_time(&native_duration, now);
119 
120             monitorable->publish_status(
121                     rti::core::native_conversions::from_native(native_duration));
122         } catch (rti::apputils::ServiceException& ex) {
123             SERVICELog_fromException(ex);
124         } catch (std::exception& ex) {
125             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
126         } catch (...) {
127             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
128         }
129     }
130 
131     static void sample_status(
132             void *monitorable_data,
133             const struct RTINtpTime *now)
134     {
135         Monitorable *monitorable =
136                 static_cast<Monitorable *>(monitorable_data);
137         try {
138             DDS_Duration_t native_duration = DDS_DURATION_ZERO;
139             DDS_Duration_from_ntp_time(&native_duration, now);
140 
141             monitorable->sample_status(
142                     rti::core::native_conversions::from_native(native_duration));
143         } catch (rti::apputils::ServiceException& ex) {
144             SERVICELog_fromException(ex);
145         } catch (std::exception& ex) {
146             SERVICELog_exception(&RTI_LOG_ANY_s, ex.what());
147         } catch (...) {
148             SERVICELog_exception(&RTI_LOG_UNEXPECTED_EXCEPTION);
149         }
150     }
151 
152 };
153 
154 inline
155 void register_monitorable(
156         rti::routing::RoutingService& router,
157         rti::routing::monitoring::Monitorable *monitorable,
158         const dds::core::Duration& sampling_period,
159         const dds::core::Duration& publication_period)
160 {
161     MonitorableForwarder::native forwarder =
162             MonitorableForwarder::create(monitorable);
163     DDS_Duration_t native_sampling;
164     rti::core::native_conversions::to_native(native_sampling, sampling_period);
165     DDS_Duration_t native_publication;
166     rti::core::native_conversions::to_native(native_publication, publication_period);
167     if (RTI_RoutingService_register_monitorable(
168             router->native(),
169             &forwarder,
170             &native_sampling,
171             &native_publication) == -1) {
172         RTI_THROW_SERVICE_EXCEPTION(
173                 &RTI_LOG_ANY_FAILURE_s,
174                 "register native monitorable");
175     }
176 }
177 
178 inline
179 void unregister_monitorable(
180         rti::routing::RoutingService& router,
181         rti::routing::monitoring::Monitorable *monitorable)
182 {
183     MonitorableForwarder::native forwarder =
184             MonitorableForwarder::create(monitorable);
185     RTI_RoutingService_unregister_monitorable(router->native(),  &forwarder);
186 }
187 
188 
189 class MonitorableAccessGuard {
190 public:
191     MonitorableAccessGuard(rti::routing::RoutingService& service)
192         : service_(service),
193          is_taken_(false)
194     {
195         RTI_RoutingService_begin_monitorable_access(service_->native());
```

```
196         is_taken_ = true;
197     }
198
199     ~MonitorableAccessGuard()
200     {
201         end();
202     }
203
204     void end()
205     {
206         if (is_taken_) {
207             RTI_RoutingService_end_monitorable_access(service_->native());
208             is_taken_ = false;
209         }
210     }
211
212 private:
213     rti::routing::RoutingService& service_;
214     bool is_taken_;
215 };
216
217
218 }}}
219
220
221 #endif // RTI_SERVICE_MONITORING_MONITORABLE_HPP_
```

Index