RTI Connector for JavaScript

Version 1.1.0

Contents

RTI® Connext® DDS is a connectivity software framework for integrating data sources of all types. At its core is the world's leading ultra-high performance, distributed networking databus.

RTI Connector provides a quick and easy way to write applications that publish and subscribe to the *RTI Connext DDS* databus in JavaScript (with Node.js) and other languages.

You can learn how to use *RTI Connector* by reading the following sections, which include examples and detailed API reference. You can also find a specific type or function in the genindex.

Chapter 1

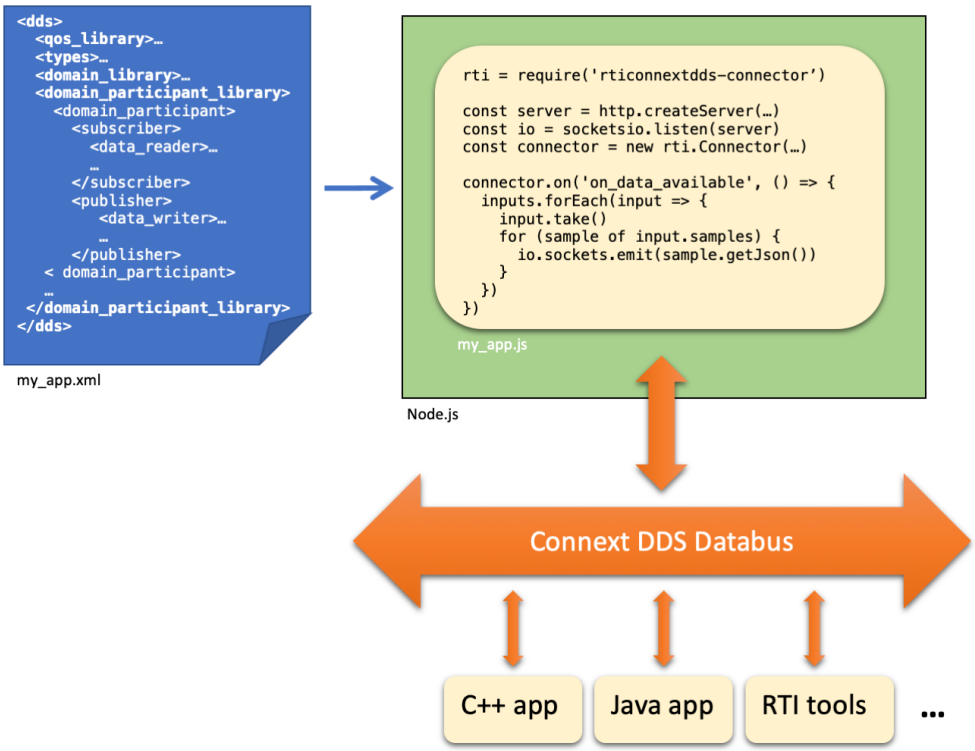
Introduction to RTI Connector

RTI Connext DDS is a software connectivity framework for real-time distributed applications. It uses the publish-subscribe communications model to make data distribution efficient and robust. At its core is the world's leading ultra-high performance, distributed networking databus.

RTI Connector is a family of simplified APIs for publishing and subscribing to the *Connext DDS* Databus in programming languages such as Python and JavaScript.

Note: This documentation assumes you are already familiar with basic DDS concepts. You can learn about DDS in the *RTI Connext DDS Getting Started Guide*, *RTI Connext DDS Core Libraries User's Manual*, and the *Connext DDS* API documentation for C, C++, Java and .NET. These documents are available from the RTI Community portal.

In *Connector*, the DDS system is defined in XML. This includes the DDS entities and their data types and quality of service. Applications instantiate a `Connector()` object that loads an XML configuration and creates the entities that allow publishing and subscribing to DDS Topics.



By publishing and subscribing to DDS Topics, *Connector* seamlessly works with any other *DDS* applications, including *Connext DDS* user applications, and RTI Tools and Infrastructure Services.

1.1 How to read this documentation

- First learn how to install *Connector* and run the examples in *Getting Started*.
- Learn how to define the DDS system in XML in *Defining a DDS System in XML*.
- Learn how to write a *Connector* application in *Loading a Connector*, *Writing Data (Output)*, and *Reading Data (Input)*. These sections include examples and detailed type and function documentation.
- See *Advanced Topics* for details on the different ways to access the data, the threading model, and error reporting. If you want to know whether a *Connext DDS* feature is supported in *Connector*, and how to use it, see *Connext DDS Features*.

If you're looking for a specific class or function, go directly to the genindex.

Chapter 2

Getting Started

2.1 Installing RTI Connector for JavaScript

RTI Connector for JavaScript can be installed with npm in two ways:

You can pass the package name:

```
$ npm install rticonnextdds-connector
```

Or the GitHub repository:

```
$ npm install https://www.github.com/rticommunity/rticonnextdds-connector-js.
↪git
```

In order to access the examples, run npm with the GitHub repository.

Connector works with Node.js versions 10.20.x[1] to 13.x.x[2]. It currently doesn't work with versions 14+ because one of its dependencies is not yet compatible with that version.

npm uses node-gyp to locally compile some of *Connector*'s dependencies. This requires Python 2.7 (it will not work with Python 3) and a relatively recent C++ compiler (such as gcc 4.8+).

On Windows systems, you can install the Windows Build Tools, which include both the Visual C++ compiler and Python 2.7.

For more information, see *Supported Platforms*.

2.2 Running the examples

The examples are in the examples/nodejs directory of the *RTI Connector for JavaScript* GitHub repository. The npm installation will copy the examples under `<installation directory>/node_modules/rticonnextdds-connector/`.

[1] Note that Connector for JavaScript is not compatible with versions of Node.js prior to v10.20.x.

[2] Note that Connector for JavaScript is not compatible with Node.js v12.19.0 due to a regression that was introduced in that version of Node.js. Connector for JavaScript works with Node.js versions 12.18.x and 12.20.x.

In the simple example, `writer.js` periodically publishes data for a *Square* topic, and `reader.js` subscribes to the topic and prints all the data samples it receives.

Run the reader as follows:

```
node examples/nodejs/simple/reader.js
```

And, in another shell, run the writer:

```
node examples/nodejs/simple/writer.js
```

This is what `reader.js` looks like:

```
const rti = require('rticonnextdds-connector')
const configFile = path.join(__dirname, '/../ShapeExample.xml')

const run = async () => {
  const connector = new rti.Connector('MyParticipantLibrary::MySubParticipant
↪', configFile)
  const input = connector.getInput('MySubscriber::MySquareReader')
  try {
    console.log('Waiting for publications...')
    await input.waitForPublications()

    console.log('Waiting for data...')
    for (let i = 0; i < 500; i++) {
      await input.wait()
      input.take()
      for (const sample of input.samples.validDataIter) {
        // You can obtain all the fields as a JSON object
        const data = sample.getJson()
        const x = data.x
        const y = data.y
        // Or you can access each field individually
        const size = sample.getNumber('shapesize')
        const color = sample.getString('color')

        console.log('Received x: ' + x + ', y: ' + y + ', shapesize: ' + size␣
↪+ ', color: ' + color)
      }
    }
  } catch (err) {
    console.log('Error encountered: ' + err)
  }
  connector.close()
}

run()
```

And this is `writer.js`:

```
const rti = require('rticonnextdds-connector')
const configFile = path.join(__dirname, '/../ShapeExample.xml')
```

(continues on next page)

(continued from previous page)

```
const run = async () => {
  const connector = new rti.Connector('MyParticipantLibrary::MyPubParticipant
↪', configFile)
  const output = connector.getOutput('MyPublisher::MySquareWriter')
  try {
    console.log('Waiting for subscriptions...')
    await output.waitForSubscriptions()

    console.log('Writing...')
    for (let i = 0; i < 500; i++) {
      output.instance.setNumber('x', i)
      output.instance.setNumber('y', i * 2)
      output.instance.setNumber('shapesize', 30)
      output.instance.setString('color', 'BLUE')
      output.write()

      sleep.msleep(500)
    }

    console.log('Exiting...')
    // Wait for all subscriptions to receive the data before exiting
    await output.wait()
  } catch (err) {
    console.log('Error encountered: ' + err)
  }
  connector.close()
}

run()
```

You can run the reader and the writer in any order, and you can run multiple instances of each at the same time. You can also run any other *DDS* application that publishes or subscribes to the *Square* topic. For example, you can use RTI Shapes Demo.

Chapter 3

Defining a DDS System in XML

Connector loads the definition of a DDS system from an XML configuration file that includes the definition of domains, *DomainParticipants*, *Topics*, *DataReaders* and *DataWriters*, data types and quality of service.

Connector uses the XML schema defined by RTI's XML-Based Application Creation feature.

Hint: The *Connext DDS* C, C++, Java and .NET APIs can also load the same XML files you write for *Connector*.

The following table summarizes the XML tags, the DDS concepts they define, and how they are exposed in the *Connector* API:

Table 1: XML Configuration Tags

XML Tag	DDS Concept	Connector API
`<types>`	*DDS data types* (the types associated with *Topics*)	Types used by inputs and outputs (`Input()` and `Output()`).
`<domain_library>`, `<domain>`, `<register_type>`, and `<topic>`	*DDS Domain*, *Topic*	Defines the domain joined by a `Connector()` and the *Topics* used by its inputs and outputs (`Input()` and `Output()`).
`<domain_participant_library>` and `<domain_participant>`	*DomainParticipant*	Each `Connector()` instance loads a `<domain_participant>`. See *Loading a Connector*.
`<publisher>` and `<data_writer>`	*Publisher* and *DataWriter*	Each `<data_writer>` defines an `Output()`. See *Writing Data (Output)*.
`<subscriber>` and `<data_reader>`	*Subscriber* and *DataReader*	Each `<data_reader>` defines an `Input()`. See *Reading Data (Input)*.
`<qos_library>` and `<qos_profile>`	Quality of service (QoS)	Quality of service used to configure `Connector()`, `Output()` and `Input()`.

Hint: For an example configuration file, see ShapeExample.xml.

3.1 Data types

The `<types>` tags defines the data types associated with the *Topics* to be published or subscribed to.

The following example defines a *ShapeType* with four members: `color`, `x`, `y` and `shapesize`:

```
  <types>
    <struct name="ShapeType">
        <member name="color" type="string" stringMaxLength="128" key="true"/>
        <member name="x" type="int32"/>
        <member name="y" type="int32"/>
        <member name="shapesize" type="int32"/>
    </struct>
</types>
```

Types are associated with *Topics*, as explained in the next section, *Domain library*.

Hint: You can define your types in IDL and convert them to XML with rtiddsgen. For example: `rtiddsgen -convertToXml MyTypes.idl`

For more information about defining types, see Creating User Data Types with XML in the *Connext DDS Core Libraries User's Manual*.

For more information about accessing the data samples, see *Accessing the data*.

3.2 Domain library

A domain library is a collection of domains. A domain specifies:

- A domain id.
- A set of registered types (from a subset of the types in `<types>`). A registered type can have a local name.
- A set of topics, which are used by *DataReaders* and *DataWriters*.

```
<domain_library name="MyDomainLibrary">
    <domain name="MyDomain" domain_id="0">
        <register_type name="ShapeType" type_ref="ShapeType"/>
        <topic name="Square" register_type_ref="ShapeType"/>
        <topic name="Circle" register_type_ref="ShapeType"/>
    </domain>
</domain_library>
```

For more information about the format of a domain library, see XML-Based Application Creation: Domain Library.

3.3 Participant library

A *DomainParticipant* joins a domain and contains *Publishers* and *Subscribers*, which contain *DataWriters* and *DataReaders*, respectively.

Each `Connector()` instance created by your application is associated with a <*domain_participant*>, as explained in *Loading a Connector*.

DataWriters and *DataReaders* are associated with a *DomainParticipant* and a *Topic*. In *Connector*, each `<data_writer>` tag defines an `Output()`, as described in *Writing Data (Output)*; and each `<data_reader>` tag defines an `Input()`, as described in *Reading Data (Input)*.

```
<domain_participant_library name="MyParticipantLibrary">
    <domain_participant name="MyPubParticipant" domain_ref=
↪"MyDomainLibrary::MyDomain">
        <publisher name="MyPublisher">
            <data_writer name="MySquareWriter" topic_ref="Square" />
        </publisher>
    </domain_participant>

    <domain_participant name="MySubParticipant" domain_ref=
↪"MyDomainLibrary::MyDomain">
        <subscriber name="MySubscriber">
            <data_reader name="MySquareReader" topic_ref="Square" />
        </subscriber>
```

(continues on next page)

(continued from previous page)

```
    </domain_participant>
</domain_participant_library>
```

For more information about the format of a participant library, see XML-Based Application Creation: Participant Library.

3.4 Quality of service

All DDS entities have an associated quality of service (QoS). There are several ways to configure it.

You can define a QoS profile and make it the default. The following example configures all *DataReaders* and *DataWriters* with reliable and transient-local QoS:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyQosProfile" is_default_qos="true">
        <datareader_qos>
            <reliability>
                <kind>RELIABLE_RELIABILITY_QOS</kind>
            </reliability>
            <durability>
                <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
            </durability>
        </datareader_qos>
        <datawriter_qos>
            <reliability>
                <kind>RELIABLE_RELIABILITY_QOS</kind>
            </reliability>
            <durability>
                <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
            </durability>
        </datawriter_qos>
    </qos_profile>
</qos_library>
```

You can define the QoS for each individual entity:

```
<domain_participant name="MyPubParticipant" domain_ref="MyDomainLibrary::MyDomain
↪">
    <domain_participant_qos> <!-- ... --> </domain_participant_qos>
    <publisher name="MyPublisher">
        <publisher_qos> <!-- ... --> </publisher_qos>
        <data_writer name="MySquareWriter" topic_ref="Square">
            <datawriter_qos>
                <reliability>
                    <kind>RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
                <durability>
                    <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
                </durability>
```

(continues on next page)

(continued from previous page)

```
            </datawriter_qos>
        </data_writer>
    </publisher>
</domain_participant>
```

Or you can use profiles and override or define additional QoS policies for each entity:

```
<domain_participant name="MyPubParticipant" domain_ref="MyDomainLibrary::MyDomain
↪">
    <domain_participant_qos base_name="MyQosLibrary::MyQosProfile">
        <!-- override or configure additional Qos policies -->
    </domain_participant_qos>
    <publisher name="MyPublisher">
        <publisher_qos base_name="MyQosLibrary::MyQosProfile">
            <!-- override or configure additional Qos policies -->
        </publisher_qos>
        <data_writer name="MySquareWriter" topic_ref="Square">
            <datawriter_qos base_name="MyQosLibrary::MyQosProfile">
                <!-- override or configure additional Qos policies -->
            </datawriter_qos>
        </data_writer>
    </publisher>
</domain_participant>
```

You can also use built-in profiles and QoS snippets. For example, the following profile is equivalent to *MyQosProfile* above:

```
<qos_library name="MyQosLibrary">
    <qos_profile name="MyQosProfile" is_default_qos="true">
        <base_name>
            <element>BuiltinQosSnippetLib::QosPolicy.Durability.TransientLocal
↪</element>
            <element>BuiltinQosSnippetLib::QosPolicy.Reliability.Reliable</
↪element>
        </base_name>
    </qos_profile>
</qos_library>
```

You can read more in the *RTI Connext DDS Core Libraries User's Manual*, Configuring QoS with XML.

3.4.1 Logging

Logging can be configured as explained in Configuring Logging via XML.

For example, to increase the logging verbosity from the default (ERROR) to WARNING, define a `qos_profile` with the attribute `is_default_participant_factory_profile="true"`:

```
<qos_profile name="Logging" is_default_participant_factory_profile="true">
    <participant_factory_qos>
        <logging>
```

(continues on next page)

(continued from previous page)

```
            <verbosity>WARNING</verbosity>
        </logging>
    </participant_factory_qos>
</qos_profile>
```

Chapter 4

Loading a Connector

4.1 Import the Connector package

To use the `rticonnextdds_connector` package, require it. For example:

```
const rti = require('rticonnextdds-connector')
```

4.2 Creating a new Connector

To create a new *`Connector()`*, pass an XML file and a configuration name to the constructor:

```
const connector = new rti.Connector('MyParticipantLibrary::MyParticipant',
↪'ShapeExample.xml')
```

The XML file defines your types, QoS profiles, and DDS Entities. *Connector* uses the XML schema of RTI's XML-Based Application Creation.

The previous code loads the `<domain_participant>` named *MyParticipant* in the `<domain_participant_library>` named *MyParticipantLibrary*, which is defined in the file `ShapeExample.xml`:

```
<domain_participant_library name="MyParticipantLibrary">
  <domain_participant name="MyParticipant" domain_ref="MyDomainLibrary::MyDomain
↪">
    ...
  </domain_participant>
</domain_participant_library>
```

See the full file here: ShapeExample.xml.

When you create a *`Connector()`*, the DDS *DomainParticipant* that you selected and all of its contained entities (*Topics*, *Subscribers*, *DataReaders*, *Publishers*, *DataWriters*) are created.

For more information about the DDS entities, see Core Concepts in the *RTI Connext DDS Core Libraries User's Manual*.

Note: Operations on the same *Connector()* instance or its contained entities are not protected for multi-threaded access. See *Threading model* for more information.

4.3 Closing a Connector

To destroy all the DDS entities that belong to a previously created *Connector()*, call *Connector.close()*:

```
const connector = new rti.Connector('MyParticipantLibrary::MyParticipant',
↪'ShapeExample.xml')
// ...
connector.close()
```

Connector.close() returns a Promise that will resolve once the Connector object has been destroyed. The *Connector.close()* operation is only asynchronous (and therefore, it is only necessary to wait for the Promise to resolve) if you have installed a listener for the 'on_data_available' event.

For more information, see *Additional considerations when using event-based functionality*.

4.4 Getting the Inputs and Outputs

Once you have created a *Connector()* instance, *Connector.getOutput()* returns the *Output()* that allows writing data, and *Connector.getInput()* returns the *Input()* that allows reading data.

Note: If the <domain_participant> you load contains both <data_writer> tags (Outputs) and <data_reader> tags (Inputs) for the same *Topic* and they have matching QoS, when you write data, the Inputs will receive the data even before you call *Connector.getInput()*. To avoid that, you can configure the <subscriber> that contains the <data_reader> with <subscriber_qos>/<entity_factory>/<autoenable_created_entities> set to false. Then the Inputs will only receive data after you call *Connector.getInput()*.

For more information see:

- *Writing Data (Output)*
- *Reading Data (Input)*

4.5 Class reference: Connector

class Connector(*configName*, *url*)
Loads a configuration and creates its Inputs and Outputs.

Note: The *Connector()* class inherits from EventEmitter. This allows us to support event-based notification for data, using the following syntax:

```
connector.on('on_data_available', () => { } )
```

Please refer to *Reading Data (Input)* for more information.

A *Connector()* instance loads a configuration file from an XML document. For example:

```
const connector = new rti.Connector('MyParticipantLibrary::MyParticipant
↪', 'MyExample.xml')
```

After creating it, the *Connector()* object's Inputs can be used to read data, and the Outputs to write data. The methods *Connector.getOutput()* and *Connector.getInput()* return an *Input()* and *Output()*, respectively.

An application can create multiple *Connector()* instances for the same or different configurations.

Arguments

- **configName** (*string*) – The configuration to load. The configName format is *LibraryName::ParticipantName*, where LibraryName is the name attribute of a <domain_participant_library> tag, and ParticipantName is the name attribute of a <domain_participant> tag within that library.
- **url** (*string*) – A URL locating the XML document. It can be a file path (e.g., /tmp/my_dds_config.xml), a string containing the full XML document with the following format: str://"<dds>...</dds>", or a combination of multiple files or strings, as explained in the URL Groups section of the *Connext DDS Core Libraries User's Manual*.

Connector.**close**(*timeout*)

Frees all the resources created by this *Connector()* instance.

Arguments

- **timeout** (*number*) – Optional parameter to indicate the timeout of the operation, in seconds. By default, 10s. If this operation does not complete within the specified timeout, the returned Promise will be rejected.

Returns **Promise** – Which resolves once the *Connector()* object has been freed. It is only necessary to wait for this promise to resolve if you have attached a listener for the on_data_available event.

Connector.**getInput**(*inputName*)

Returns the *Input()* named inputName.

inputName identifies a <data_reader> tag in the configuration loaded by the *Connector()*. For example:

```
const connector = new rti.Connector('MyParticipantLibrary::MyParticipant
↪', 'MyExample.xml')
connector.getInput('MySubscriber::MyReader')
```

Loads the Input in this XML:

```
<domain_participant_library name="MyParticipantLibrary">
  <domain_participant name="MyParticipant" domain_ref=
↪"MyDomainLibrary::MyDomain">
    <subscriber name="MySubscriber">
      <data_reader name="MyReader" topic_ref="MyTopic"/>
      ...
    </subscriber>
    ...
  </domain_participant>
  ...
<domain_participant_library>
```

Arguments

- **inputName** (*string*) – The name of the `data_reader` to load, with the format *SubscriberName::DataReaderName*.

Returns **Input** – The Input, if it exists.

`Connector.`**getOutput**(*outputName*)

Returns the *`Output()`* named outputName.

`outputName` identifies a `<data_writer>` tag in the configuration loaded by the *`Connector()`*. For example:

```
const connector = new rti.Connector('MyParticipantLibrary::MyParticipant
↪', 'MyExample.xml')
connector.getOutput('MyPublisher::MyWriter')
```

Loads the Input in this XML:

```
<domain_participant_library name="MyParticipantLibrary">
  <domain_participant name="MyParticipant" domain_ref=
↪"MyDomainLibrary::MyDomain">
    <publisher name="MyPublisher">
      <data_writer name="MyWriter" topic_ref="MyTopic"/>
      ...
    </publisher>
    ...
  </domain_participant>
  ...
<domain_participant_library>
```

Arguments

- **outputName** (*string*) – The name of the `data_writer` to load, with the format `PublisherName::DataWriterName`.

Returns **Output** – The Output, if it exists.

`Connector.`**wait**(*timeout*)

Waits for data to be received on any Input.

Note: This operation is asynchronous.

Arguments

- **timeout** (*number*) – The maximum time to wait, in milliseconds. By default, infinite.

Throws ***TimeoutError*** – *`TimeoutError()`* will be thrown if the timeout expires before data is received.

Returns **Promise** – A `Promise` which will be resolved once data is available, or rejected once the timeout expires.

`Connector.`**waitForCallbackFinalization**(*timeout*)

Returns a `Promise` that will resolve once the resources used internally by the *`Connector()`* are no longer in use.

Note: This returned promise will be rejected if there are any listeners registered for the `on_data_available` event. Ensure that they have all been removed before calling this method using `connector.removeAllListeners(on_data_available)`.

It is currently only necessary to call this method if you remove all of the listeners for the `on_data_available` event and at some point in the future wish to use the same *`Connector()`* object to get notifications of new data (via the *`Connector.wait()`* method, or by re-adding a listener for the `on_data_available` event).

This operation does **not** free any resources. It is still necessary to call *`Connector.close()`* when the *`Connector()`* is no longer required.

Arguments

- **timeout** (*number*) – Optional parameter to indicate the timeout of the operation, in seconds. By default, 10s. If this operation does not complete within the specified timeout, the returned Promise will be rejected.

Returns **Promise** – A Promise that will be resolved once the resources being used internally by the *`Connector()`* object are no longer in use.

`Connector.`**setMaxObjectsPerThread**(*value*)

Allows you to increase the number of *`Connector()`* instances that can be created.

The default value is 2048 (which allows for approximately 15 instances of *Connector()* to be created in a single application). If you need to create more than 8 instances of *Connector()*, you can increase the value from the default.

Note: This is a static method. It can only be called before creating a *Connector()* instance.

See SYSTEM_RESOURCE_LIMITS QoS Policy in the *RTI Connext DDS Core Libraries User's Manual* for more information.

Arguments

- **value** (*number*) – The value for max_objects_per_thread

Chapter 5

Writing Data (Output)

5.1 Getting the Output

To write a data sample, first look up an output:

```
output = connector.getOutput('MyPublisher::MySquareWriter')
```

Connector.getOutput() returns an *Output()* object. This example obtains the output defined by the `data_writer` named *MySquareWriter* within the `publisher` named *MyPublisher*:

```
<publisher name="MyPublisher">
  <data_writer name="MySquareWriter" topic_ref="Square" />
</publisher>
```

This `publisher` is defined inside the `domain_participant` selected to create this *Connector()* (see *Creating a new Connector*).

5.2 Populating the data sample

The next step is to set the *Instance()* fields. You can set them member-by-member:

```
output.instance.setNumber('x', 1)
output.instance.setNumber('y', 2)
output.instance.setNumber('shapesize', 30)
output.instance.setString('color', 'BLUE')
```

Or using a JSON object:

```
output.instance.setFromJson({ x: 1, y: 2, shapesize: 30, color: 'BLUE' })
```

The name of each member corresponds to the type assigned to this output in XML. For example, the XML configuration corresponding to the above code snippets is:

```
<struct name="ShapeType">
  <member name="color" type="string" stringMaxLength="128" key="true" default=
↪"RED"/>
  <member name="x" type="long" />
  <member name="y" type="long" />
  <member name="shapesize" type="long" default="30"/>
 </struct>
```

See *Instance()* and *Accessing the data* for more information.

5.3 Writing the data sample

To write the values that have been set in `Output.instance`, call *Output.write()*:

```
output.write()
```

If the `datawriter_qos` is reliable, you can use *Output.wait()* to block until all matching reliable subscribers acknowledge the reception of the data sample:

```
try {
  await output.wait()
} catch (err) {
  console.log('Error caught: ' + err)
}
```

The write method can also receive a JSON object specifing several options. For example, to write with a specific timestamp:

```
output.write({ source_timestamp: 100000 })
```

It is also possible to dispose or unregister an instance:

```
output.write({ action: 'dispose' })
output.write({ action: 'unregister' })
```

In these two cases, only the *key* fields in the `Output.instance` are relevant.

See *Output.write()* for more information on the supported parameters.

5.4 Matching with a subscription

Before writing, you can use the method *Output.waitForSubscriptions()* to detect when a compatible DDS subscription is matched or stops matching. It returns a `Promise` that will resolve to the change in the number of matched subscriptions since the last time it was called.

You can wait for the `Promise` using `await` in an async function:

```
let changeInMatches = await output.waitForSubscriptions()
```

Or using the `then` and `catch` methods:

```
output.waitForSubscriptions().then((res) => {
  // The Promise resolved successfully and the number of matches is stored in␣
↪res
}).catch((err) => {
  // Handle the error (which is possibly a timeout)
}
```

For example, if a new compatible subscription is discovered within the specified `timeout`, the Promise will resolve to 1; if a previously matching subscription no longer matches (for example, due to the application being closed), it resolves to -1.

You can obtain information about the currently matched subscriptions with the *`Output.matchedSubscriptions`* property:

```
output.matchedSubscriptions.forEach((match) => {
 subName = match.name
}
```

5.5 Class reference: Output, Instance

5.5.1 Output class

`class Output`(*connector*, *name*)

Allows writing data for a DDS Topic.

This class is used to publish data for a DDS Topic. To get an Output object, use *`Connector.getOutput()`*.

Attributes:

- `instance` (*`Instance()`*) - The data that is written when *`Output.write()`* is called.
- `connector` (*`Connector()`*) - The *`Connector()`* object that created this object.
- `name` (str) - The name of this Output (the name used in *`Connector.getOutput()`*).
- `native` (pointer) - The native handle that allows accessing additional *Connext DDS* APIs in C.
- `matchedSubscriptions` (JSON) - Information about matched subscriptions (see below).

`Output.`**`clearMembers`**`()`

Resets the values of the members of this *`Instance()`*.

If the member is defined with a *default* attribute in the configuration file, it gets that value. Otherwise, numbers are set to 0 and strings are set to empty. Sequences are cleared and optional members are set to 'null'. For example, if this Output's type is *ShapeType*, then `clearMembers()` sets:

```
color = 'RED'
shapesize = 30
x = 0
y = 0
```

Output.**matchedSubscriptions**
type: JSON

Provides information about matched subscriptions.

This property returns a JSON array, with each element of the array containing information about a matched subscription.

Currently the only information contained in this JSON object is the subscription name of the matched subscription. If the matched subscription doesn't have a name, the name for that specific subscription will be null.

Note that *Connector()* Inputs are automatically assigned a name from the data_reader name element in the XML configuration.

Output.**wait**(*timeout*)
Waits until all matching reliable subscriptions have acknowledged all the samples that have currently been written.

This method only waits if this Output is configured with a reliable QoS.

Note: This operation is asynchronous

Arguments

- **timeout** (*timeout*) – The maximum time to wait, in milliseconds. By default, infinite.

Throws ***TimeoutError*** – *TimeoutError()* will be thrown if the timeout expires before all matching reliable subscriptions acknowledge all the samples.

Returns **Promise** – Promise object which will be rejected if not all matching reliable subscriptions acknowledge all of the samples within the specified timeout.

Output.**waitForSubscriptions**(*timeout*)
Waits for this Output to match or unmatch a compatible DDS Publication.

This method waits for the specified timeout (or if no timeout is specified, it waits forever), for a match (or unmatch) to occur.

Note: This operation is asynchronous

Arguments

- **timeout** (*number*) – The maximum time to wait, in milliseconds. By default, infinite.

Throws ***TimeoutError*** – *TimeoutError()* will be thrown if the timeout expires before a subscription is matched.

Returns **Promise** – Promise object resolving with the change in the current number of matched inputs. If this is a positive number, the output has matched with new subscribers. If it is negative, the output has unmatched from a subscription. It is possible for multiple matches and/or unmatches to be returned (e.g., 0 could be returned, indicating that the output matched the same number of inputs as it unmatched).

Output.**write**(*params*)

Publishes the values of the current Instance.

Note that after writing it, the Instance's values remain unchanged. If, for the next write, you need to start from scratch, you must first call *Output.clearMembers()*.

This method accepts an optional JSON object as a parameter, which may specify the parameters to use in the *write* call. The supported parameters are a subset of those documented in the Writing Data section of the *RTI Connext DDS Core Libraries User's Manual*. These are:

- `action` – One of `write` (default), `dispose` or `unregister`
- `source_timestamp` – An integer representing the total number of nanoseconds
- `identity` – A JSON object containing the fields `writer_guid` and `sequence_number`
- `related_sample_identity` – Used for request-reply communications. It has the same format as identity

Arguments

- **params** (*JSON*) – [Optional] The optional parameters described above

Throws ***TimeoutError*** – The write method can block under multiple circumstances (see 'Blocking During a write()' in the Writing Data section of the *RTI Connext DDS Core Libraries User's Manual*.) If the blocking time exceeds the `max_blocking_time`, this method throws *TimeoutError()*.

Examples:

```
output.write()
```

```
output.write({
    action: 'dispose',
    identity: { writer_guid: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,␣
↪13, 14, 15, 16], sequence_number: 1 }
})
```

5.5.2 Instance class

class Instance(*output*)

A data sample.

Instance() is the type obtained through `Output.instance` and is the object that is published by *Output.write()*.

An Instance has an associated DDS Type, specified in the XML configuration, and it allows setting the values for the fields of the DDS Type.

Attributes:

- `output` (*Output()*) - The *Output()* that owns this Instance.
- `native` (pointer) - Native handle to this Instance that allows for additional *Connext DDS Pro* C APIs to be called.

`Instance.`**clearMember**(*fieldName*)

Resets a member to its default value.

The effect is the same as that of *Output.clearMembers()*, except that only one member is cleared.

Arguments

- **fieldName** (*string*) – The name of the field. It can be a complex member or a primitive member.

`Instance.`**getJson**()

Retrives the value of this instance as a JSON object.

Returns JSON – The value of this instance as a JSON object.

`Instance.`**native**

type: pointer

The native C object.

This property allows accessing additional *Connext DDS* APIs in C.

`Instance.`**set**(*fieldName*, *value*)

Sets the value of fieldName.

The type of the argument `value` must correspond with the type of the field with name `fieldName` (as defined in the configuration XML file).

This method is an alternative to *Instance.setNumber()*, *Instance.setString()* and *Instance.setBoolean()*. The main difference is that it is type-independent (in that the same method can be used for all fields).

Arguments

- **fieldName** (*string*) – The name of the field.
- **value** (*number|boolean|string|null*) – The value to set. Note that `null` is used to unset an optional member.

`Instance.`**`setBoolean`**(*fieldName*, *value*)
Sets a boolean field.

Arguments

- **`fieldName`** (*`string`*) – The name of the field.
- **`value`** (*`boolean`*) – A boolean value, or null, to unset an optional member.

`Instance.`**`setFromJson`**(*jsonObj*)
Sets the member values specified in a JSON object.

The keys in the JSON object are the member names of the DDS Type associated with the Output, and the values are the values to set for those members.

This method sets the values of those members that are explicitly specified in the JSON object. Any member that is not specified in the JSON object will retain its previous value.

To clear members that are not in the JSON object, call *`Output.clearMembers()`* before this method. You can also explicitly set any value in the JSON object to *null* to reset that field to its default value.

Arguments

- **`jsonObj`** (*`JSON`*) – The JSON object containing the keys (field names) and values (values for the fields).

`Instance.`**`setNumber`**(*fieldName*, *value*)
Sets a numeric field.

Arguments

- **`fieldName`** (*`string`*) – The name of the field.
- **`value`** (*`number`*) – A numeric value, or null, to unset an optional member.

`Instance.`**`setString`**(*fieldName*, *value*)
Sets a string field.

Arguments

- **`fieldName`** (*`string`*) – The name of the field.
- **`value`** (*`number`*) – A string value, or null, to unset an optional member.

Chapter 6

Reading Data (Input)

6.1 Getting the input

To read/take samples, first get a reference to the *Input()*:

```
input = connector.getInput('MySubscriber::MySquareReader')
```

Connector.getInput() returns an *Input()* object. This example, obtains the Input defined by the data_reader named *MySquareReader* within the <subscriber> named *MySubscriber*:

```
<subscriber name="MySubscriber">
  <data_reader name="MySquareReader" topic_ref="Square" />
</subscriber>
```

This <subscriber> is defined inside the <domain_participant> selected to create this connector (see *Creating a new Connector*).

6.2 Reading or taking the data

Call *Input.take()* to access and remove the samples:

```
input.take()
```

or *Input.read()* to access the samples but leave them available for a future read() or take():

```
input.read()
```

The method *Input.wait()* can be used to identify when there is new data available on a specific *Input()*. It returns a Promise that will be resolved when new data is available, or rejected if the supplied timeout expires.

You can wait for the Promise using await in an async function:

```
await input.wait()
input.take()
```

Or using the `then` method:

```
input.wait()
  .then(() => {
    input.take()
  })
```

The method *`Connector.wait()`* has the same behavior as *`Input.wait()`*, but the returned promise will be resolved when data is available on *any* of the *`Input()`* objects within the *`Connector()`*:

```
await connector.wait()
```

You can also install a listener in a *`Connector()`*. *`Connector()`* inherits from EventEmitter. If a listener for the `on_data_available` event is attached to a *`Connector()`*, this event will be emitted whenever new data is available on any `Input defined within the :class:`Connector()`.`

```
// Store all the inputs in an array
const inputs = [
  connector.getInput('MySubscriber::MySquareReader'),
  connector.getInput('MySubscriber::MyCircleReader'),
  connector.getInput('MySubscriber::MyTriangleReader')
]

// Install the listener
connector.on('on_data_available', () => {
  // One or more inputs have data
  inputs.forEach(input => {
    input.take()
    for (const sample of input.samples.validDataIter) {
      // Access the data
    }
  }
})
```

For more information on how to use the event-based notification, refer to the documentation of the events module.

The web_socket example shows how to use this event.

Warning: There are additional threading concerns to take into account when using the `on_data_available` event. Refer to *Additional considerations when using event-based functionality* for more information.

Note: When using the event-based methods to be notified of available data, errors are propagated using the `error` event. See *Error Handling* for more information.

6.3 Accessing the data samples

After calling *Input.read()* or *Input.take()*, *Input.samples* contains the data samples:

```
for (const sample of input.samples) {
   if (sample.validData) {
      console.log(JSON.stringify(sample.getJson()))
   }
}
```

SampleIterator.getJson() retrieves all the fields of a sample.

If you don't need to access the meta-data (see *Accessing sample meta-data*), the simplest way to access the data is to use *Samples.validDataIter* to skip samples with invalid data:

```
for (const sample of input.samples.validDataIter) {
   // It is not necessary to check the sample.validData field
   console.log(JSON.stringify(sample.getJson()))
}
```

It is also possible to access an individual sample:

```
// Obtain the first sample in the Input's queue
const theSample = input.samples.get(0)
if (theSample.validData) {
   console.log(JSON.stringify(theSample.getJson()))
}
```

Both of the iterables shown above also provide iterator implementations, allowing them to be incremented outside of a `for` loop:

```
const iterator = input.samples.validDataIter.iterator()
let sample = iterator.next()
// sample.value contains contains the current sample and sample.done is a
// boolean value which will become true when we have iterated over all of
// the available samples
console.log(JSON.stringify(sample.value.getJson()))
```

Warning: All the methods described in this section return generators. Calling read/take again invalidates all generators currently in use.

Samples.getJson() can receive a `fieldName` to only return the fields of a complex member. In addition to `getJson`, you can get the values of specific primitive fields using *SampleIterator.getNumber()*, *SampleIterator.getBoolean()* and *SampleIterator.getString()*, for example:

```
for (const sample of input.samples.validDataIter) {
   const x = sample.getNumber('x')
   const y = sample.getNumber('y')
```

(continues on next page)

(continued from previous page)

```
    const size = sample.getNumber('shapesize')
    const color = sample.getString('color')
}
```

See more information and examples in *Accessing the data*.

6.4 Accessing sample meta-data

Every sample contains an associated *SampleInfo* with meta-information about the sample:

```
for (const sample of input.samples) {
    const sourceTimestamp = sample.info.get('source_timestamp')
}
```

See *SampleIterator.info* for the list of available meta-data fields.

Connext DDS can produce samples with invalid data, which contain meta-data only. For more information about this, see Valid Data Flag in the *RTI Connext DDS Core Libraries User's Manual*. These samples indicate a change in the instance state. Samples with invalid data still provide the following information:

- The *SampleInfo()*
- When an instance is disposed (`sample.info.get('instance_state')` is `'NOT_ALIVE_DISPOSED'`), the sample data contains the value of the key that has been disposed. You can access the key fields only. See *Accessing key values of disposed samples*.

6.5 Matching with a publication

The method *Input.waitForPublications()* can be used to detect when a compatible DDS publication is matched or unmatched. It returns a promise that resolves to the change in the number of matched publications since the last time it was called:

```
// From within an async function. Otherwise, use the .then() syntax
let changeInMatches = await input.waitForPublications()
```

For example, if 1 new compatible publication is discovered within the specified `timeout`, the promise will resolve to 1; if a previously matching publication no longer matches, it resolves to -1.

You can obtain information about the existing matched publications through the *Input.matchedPublications* property:

```
input.matchedPublications.forEach((match) => {
    pubName = match.name
}
```

6.6 Class reference: Input, Samples, SampleIterator

6.6.1 Input class

class Input(*connector*, *name*)

Allows reading data for a DDS Topic.

This class is used to subscribe to a specific DDS Topic.

To get an Input object, use *Connector.getInput()*.

Attributes:

- connector (*Connector()*) - The Connector creates this Input.
- name (string) - The name of the Input (the name used in *Connector.getInput()*).
- native (pointer) - A native handle that allows accessing additional *Connext DDS* APIs in C.
- matchedPublications (JSON) - A JSON object containing information about all the publications currently matched with this Input.

Input.**matchedPublications**

type: JSON

Returns information about matched publications.

This property returns a JSON array, with each element of the array containing information about a matched publication.

Currently the only information contained in this JSON object is the publication name of the matched publication. If the matched publication doesn't have a name, the name for that specific publication will be null.

Note that *Connector()* Outputs are automatically assigned a name from the `data_writer name` element in the XML configuration.

Input.**read**()

Accesses the samples received by this Input.

This operation performs the same operation as *Input.take()* but the samples remain accessible (in the internal queue) after the operation has been called.

Input.**samples**

type: Samples

Allows iterating over the samples returned by this input.

This container provides iterators to access the data samples retrieved by the most-recent call to *Input.take()* and *Input.read()*.

Input.**take**()

Accesses the samples receieved by this Input.

After calling this method, the samples are accessible using *Input.samples()*.

`Input.`**`wait`**(*timeout*)
Wait for this Input to receive data.

Note: This operation is asynchronous.

Arguments

- **`timeout`** (*`number`*) – The maximum time to wait, in milliseconds. By default, infinite.

Throws ***`TimeoutError`*** – *`TimeoutError()`* will be thrown if the timeout expires before data is received.

Returns **Promise** – A `Promise` which will be resolved once data is available, or rejected if the timeout expires.

`Input.`**`waitForPublications`**(*timeout*)
Waits for this Input to match or unmatch a compatible DDS Subscription.

Note: This operation is asynchronous.

This method waits for the specified timeout (or if no timeout is specified, it waits forever), for a match (or unmatch) to occur.

Arguments

- **`timeout`** (*`number`*) – The maximum time to wait, in milliseconds. By default, infinite.

Throws ***`TimeoutError`*** – *`TimeoutError()`* will be thrown if the timeout expires before any publications are matched.

Returns **Promise** – Promise object resolving with the change in the current number of matched outputs. If this is a positive number, the input has matched with new publishers. If it is negative, the input has unmatched from an output. It is possible for multiple matches and/or unmatches to be returned (e.g., 0 could be returned, indicating that the input matched the same number of outputs as it unmatched).

6.6.2 Samples class

`class Samples`(*input*)
Provides access to the data samples read by an *`Input()`*.

This class provides access to data samples read by an *`Input()`* (using either the *`Input.read()`* or *`Input.take()`* methods).

This class implements a `[Symbol.iterator]()` method, making it an iterable. This allows it to be used in `for... of` loops, to iterate through available samples:

```
for (const sample of input.samples) {
   console.log(JSON.stringify(sample.getJson()))
}
```

The method *Samples.get()* returns a *SampleIterator()* which can also be used to access available samples:

```
const sample = input.samples.get(0)
console.log(JSON.stringify(sample.getJson()))
```

The samples returned by these methods may only contain meta-data (see *SampleIterator.info*). The *Samples.validDataIter* iterable only iterates over samples that contain valid data (a *ValidSampleIterator()*).

Samples() and *ValidSampleIterator()* both also provide generators to the samples, allowing applications to define their own iterables (see *Samples.iterator()* and *ValidSampleIterator.iterator()*).

Samples is the type of the property *Input.samples()*.

For more information and examples, see *Accessing the data samples*.

Attributes:

- length (number) - The number of samples available since the last time *Input.read()* or *Input.take()* was called.
- validDataIter (*ValidSampleIterator()*) - The class used to iterate through the available samples that have valid data.

Samples.**get**(*index*)

Returns an iterator to the data samples, starting at the index specified.

The iterator provides access to all the data samples retrieved by the most recent call to *Input.read()* or *Input.take()*.

This iterator may return samples with invalid data (samples that only contain meta-data). Use *Samples.validDataIter* to avoid having to check *SampleIterator.validData*.

Arguments

- **index** (*number*) – The index of the sample from which the iteration should begin. By default, the iterator begins with the first sample.

Returns *SampleIterator()* - An iterator to the samples (which implements both iterable and iterator logic).

Samples.**getBoolean**(*index*, *fieldName*)

Obtains the value of a boolean field within this sample.

See *Accessing the data samples*.

Arguments

- **index** (*number*) – The index of the sample.

- **fieldName** (*string*) – The name of the field.

Returns number – The obtained value.

Samples.**getJson**(*index*, *memberName*)

Gets a JSON object with the values of all the fields of this sample.

Arguments

- **index** (*number*) – The index of the sample.
- **memberName** (*string*) – The name of the complex member. The type of the member with name memberName must be an array, sequence, struct, value or union.

Returns JSON – The obtained JSON object. See *Accessing the data samples*.

Samples.**getNative**(*index*)

Obtains a native handle to the sample, which can be used to access additional *Connext DDS* APIs in C.

Arguments

- **index** (*number*) – The index of the sample for which to obtain the native pointer.

Returns pointer – A native pointer to the sample.

Samples.**getNumber**(*index*, *fieldName*)

Obtains the value of a numeric field within this sample.

See *Accessing the data samples*.

Arguments

- **index** (*number*) – The index of the sample.
- **fieldName** (*string*) – The name of the field.

Returns number – The obtained value.

Samples.**getString**(*index*, *fieldName*)

Obtains the value of a string field within this sample.

See *Accessing the data samples*.

Arguments

- **index** (*number*) – The index of the sample.
- **fieldName** (*string*) – The name of the field.

Returns string – The obtained value.

Samples.**getValue**(*fieldName*)

Gets the value of a field within this sample.

See *Accessing the data samples*.

This API can be used to obtain strings, numbers, booleans and the JSON representation of complex members.

Arguments

- **fieldName** (*string*) – The name of the field.

Returns **number|string|boolean|JSON** – The value of the field.

`Samples.`**`iterator`**`()`

The iterator generator (used by the iterable).

This method returns a generator, which must be incremented manually by the application (using the iterator.next() method).

Once incremented, the data can be accessed via the `.value` attribute. Once no more samples are available, the `.done` attribute will be true.

Using this method, it is possible to create your own iterable:

```
const iterator = input.samples.iterator()
const singleSample = iterator.next().value
```

`Samples.`**`length`**

The number of samples available.

`Samples.`**`validDataIter`**

Returns an iterator to the data samples that contain valid data.

The iterator provides access to all the data samples retrieved by the most recent call to *`Input.read()`* or *`Input.take()`*, and skips samples with invalid data (meta-data only).

By using this iterator, it is not necessary to check if each sample contains valid data.

6.6.3 SampleIterator class

`class SampleIterator`(*input*, *index*)

Iterates and provides access to a data sample.

A SampleIterator provides access to the data receieved by an *`Input()`*.

The *`Input.samples`* attribute implements a *`SampleIterator()`*, meaning it can be iterated over. An individual sample can be accessed using `Input.samples.get()`.

See *`ValidSampleIterator()`*.

This class provides both an iterator and iterable, and is used internally by the *`Samples()`* class. The following options to iterate over the samples exist:

```
// option 1 - The iterable can be used in for...of loops
for (const sample of input.samples)
// option 2 - Returns an individual sample at the given index
const individualSample = input.samples.get(0)
```

(continues on next page)

(continued from previous page)

```
// option 3 - Returns a generator which must be incremented by the␣
↪application
const iterator = input.samples.iterator()
```

Arguments

- **validData** (*boolean*) – Whether or not the current sample contains valid data.
- **infos** (SampleInfo) – The meta-data associated with the current sample.
- **native** (*pointer*) – A native handle that allows accessing additional *Connext DDS* APIs in C.

SampleIterator.**get**(*fieldName*)

Gets the value of a field within this sample.

This API can be used to obtain strings, numbers, booleans and the JSON representation of complex members.

Arguments

- **fieldName** (*string*) – The name of the field.

Returns **number|string|boolean|JSON** – The value of the field.

SampleIterator.**getBoolean**(*fieldName*)

Gets the value of a boolean field in this sample.

Arguments

- **fieldName** (*string*) – The name of the field.

Returns **boolean** – The boolean value of the field.

SampleIterator.**getJson**(*memberName*)

Returns a JSON object with the values of all the fields of this sample.

See *Accessing the data samples*.

Arguments

- **memberName** (*string*) – The name of the complex member or field to obtain.

Returns **JSON** – The obtained JSON object.

SampleIterator.**getNumber**(*fieldName*)

Gets the value of a numeric field in this sample.

Arguments

- **fieldName** (*string*) – The name of the field.

Returns **number** – The numeric value of the field.

SampleIterator.**getString**(*fieldName*)

Gets the value of a string field in this sample.

Arguments

- **fieldName** (*string*) – The name of the field.

Returns **string** – The string value of the field.

SampleIterator.**info**

Provides access to this sample's meta-data.

The `info` property expects one of the *SampleInfo()* field names:

```
const value = sample.info.get('field')
```

The supported field names are:

- `'source_timestamp'` returns an integer representing nanoseconds
- `'reception_timestamp'` returns an integer representing nanoseconds
- `'sample_identity'` or `'identity'` returns a JSON object (see *Output.write()*)
- `'related_sample_identity'` returns a JSON object (see *Output.write()*)
- `'valid_data'` returns a boolean (equivalent to *SampleIterator.validData*)
- `'view_state'`, returns a string (either "NEW" or "NOT_NEW")
- `'instance_state'`, returns a string (one of "ALIVE", "NOT_ALIVE_DISPOSED" or "NOT_ALIVE_NO_WRITERS")
- `'sample_state'`, returns a string (either "READ" or "NOT_READ")

These fields are documented in The SampleInfo Structure section in the *RTI Connext DDS Core Libraries User's Manual*.

See *SampleInfo()*.

SampleIterator.**validData**

type: boolean

Whether or not this sample contains valid data.

If `false`, the methods to obtain values of the samples (e.g., *SampleIterator.getNumber()*, *SampleIterator.getBoolean()*, *SampleIterator.getJson()*, *SampleIterator.getString()*) should not be called. To avoid this restraint, use a *ValidSampleIterator()*.

6.6.4 ValidSampleIterator class

class ValidSampleIterator()

Iterates and provides access to data samples with valid data.

This iterator provides the same methods as *SampleIterator()*. It can be obtained using `Input.samples.validDataIter`.

ValidSampleIterator.**iterator**()
The iterator generator (used by the iterable).

Using this method, it is possible to create your own iterable:

```
const iterator = input.samples.validDataIter.iterator()
const singleSample = iterator.next().value
```

6.6.5 SampleInfo class

class SampleInfo(*input*, *index*)
The type returned by the property *SampleIterator.info()*.

This class provides a way to access the SampleInfo of a received data sample.

SampleInfo.**get**(*fieldName*)
Type-independent function to obtain any value from the SampleInfo structure.

The supported fieldNames are:

- 'source_timestamp' returns an integer representing nanoseconds
- 'reception_timestamp' returns an integer representing nanoseconds
- 'sample_identity' or 'identity' returns a JSON object (see *Output.write()*)
- 'related_sample_identity' returns a JSON object (see *Output.write()*)
- 'valid_data' returns a boolean (equivalent to *SampleIterator.validData*)

These fields are documented in The SampleInfo Structure section in the *RTI Connext DDS Core Libraries User's Manual*.

Arguments

- **fieldName** (*string*) – The name of the SampleInfo field to obtain

Returns The obtained value from the SampleInfo structure

Examples:

```
const source_timestamp = input.samples.get(0).info.get('source_
↪timestamp')
```

Chapter 7

Advanced Topics

7.1 Accessing the data

The types you use to write or read data may include nested structs, sequences and arrays of primitive types or structs, etc.

These types are defined in XML following the format of RTI's XML-Based Application Creation feature.

To access the data, *`Instance()`* and *`SampleIterator()`* provide setters and getters that expect a `fieldName` string, used to identify specific fields within the type. This section describes the format of this string.

We will use the following XML type definition of MyType:

```
<types>
    <enum name="Color">
        <enumerator name="RED"/>
        <enumerator name="GREEN"/>
        <enumerator name="BLUE"/>
    </enum>
    <struct name= "Point">
        <member name="x" type="int32"/>
        <member name="y" type="int32"/>
    </struct>
    <union name="MyUnion">
        <discriminator type="nonBasic" nonBasicTypeName="Color"/>
        <case>
          <caseDiscriminator value="RED"/>
          <member name="point" type="nonBasic"  nonBasicTypeName= "Point"/>
        </case>
        <case>
          <caseDiscriminator value="GREEN"/>
          <member name="my_long" type="int32"/>
        </case>
    </union>
    <struct name= "MyType">
        <member name="my_long" type="int32"/>
```

(continues on next page)

(continued from previous page)

```
        <member name="my_double" type="float64"/>
        <member name="my_enum" type="nonBasic"  nonBasicTypeName= "Color"␣
↪default="GREEN"/>
        <member name="my_boolean" type="boolean" />
        <member name="my_point" type="nonBasic"  nonBasicTypeName= "Point"/>
        <member name="my_union" type="nonBasic"  nonBasicTypeName= "MyUnion"/>
        <member name="my_int_sequence" sequenceMaxLength="10" type="int32"/>
        <member name="my_point_sequence" sequenceMaxLength="10" type="nonBasic
↪"  nonBasicTypeName= "Point"/>
        <member name="my_point_array" type="nonBasic"  nonBasicTypeName=
↪"Point" arrayDimensions="3"/>
        <member name="my_optional_point" type="nonBasic"  nonBasicTypeName=
↪"Point" optional="true"/>
        <member name="my_optional_long" type="int32" optional="true"/>
    </struct>
</types>
```

Which corresponds to the following IDL definition:

```
enum Color {
    RED,
    GREEN,
    BLUE
};

struct Point {
    long x;
    long y;
};

union MyUnion switch(Color) {
    case RED: Point point;
    case GREEN: string<512> my_string;
};

struct MyType {
    long my_long;
    double my_double;
    Color my_enum;
    boolean my_boolean;
    string<512> my_string;
    Point my_point;
    MyUnion my_union;
    sequence<long, 10> my_int_sequence;
    sequence<Point, 10> my_point_sequence;
    Point my_point_array[3];
    @optional Point my_optional_point;
    @optional long my_optional_long;
};
```

Hint: You can get the XML definition of an IDL file with `rtiddsgen -convertToXml MyType.`

`idl`.

We will refer to an `Output` named `output` and `Input` named `input` such that `input.samples.length > 0`.

7.1.1 Using JSON objects vs accessing individual members

On an Input or an Output, you can access the data all at once by using a JSON object, or member-by-member. Using a JSON object is usually more efficient if you intend to access most or all of the data members of a large type.

On an Output, *`Instance.setFromJson()`* receives a JSON object with all, or some, of the Output type members, and in an Input, *`SampleIterator.getJson()`* retrieves all of the members.

It is also possible to provide a `memberName` to *`SampleIterator.getJson()`* to obtain a JSON object containing the fields of that nested member only.

On the other hand, the methods described in the following section receive a `fieldName` argument to get or set a specific member.

7.1.2 Accessing basic members (numbers, strings and booleans)

To set a field in an *`Output()`*, use the appropriate setter.

To set any numeric type, including enumerations:

```
output.instance.setNumber('my_long', 2)
output.instance.setNumber('my_double', 2.14)
output.instance.setNumber('my_enum', 2)
```

Warning: The range of values for a numeric field is determined by the type used to define that field in the configuration file. However, `setNumber` and `getNumber` can't handle 64-bit integers (*int64* and *uint64*) whose absolute values are larger than 2^53. This is a *Connector* limitation due to the use of *double* as an intermediate representation.

When `setNumber` or `getNumber` detect this situation, they will raise an *`DDSError()`*. `getJson` and `setJson` do not have this limitation and can handle any 64-bit integer.

To set booleans:

```
output.instance.setBoolean('my_boolean', True)
```

To set strings:

```
output.instance.setString('my_string', 'Hello, World!')
```

As an alternative to the previous setters, the type-independent method `set` can be used as follows:

```
// The set method works on all basic types
output.instance.set('my_double', 2.14)
output.instance.set('my_boolean', true)
output.instance.set('my_string', 'Hello, World!')
```

In all cases, the type of the assigned value must be consistent with the type of the field, as defined in the configuration file.

Similarly, to get a field in a *Input()* sample, use the appropriate getter: *SampleIterator.getNumber()*, *SampleIterator.getBoolean()*, *SampleIterator.getString()*, or the type-independent *SampleIterator.get()*. `getString` also works with numeric fields, returning the number as a string:

```
for (const sample of input.samples.validDataIter) {
    // Use the basic type specific getters
    let value = sample.getNumber('my_double')
    value = sample.getBoolean('my_boolean')
    value = sample.getString('my_string')

    // or alternatively, use the type-independent get method
    value = sample.get('my_double')
    value = sample.get('my_boolean')
    value = sample.get('my_string')

    // get a number as string:
    value = sample.getString('my_double')
}
```

Note: The typed getters and setters perform better than `set` and `get` in applications that write or read at high rates. Also prefer `getJson` and `setFromJson` over `set` and `get` when accessing all or most of the fields of a sample (see previous section).

Note: If a field `my_string`, defined as a string in the configuration file, contains a value that can be interpreted as a number, `sample.get('my_string')` returns a number, not a string.

7.1.3 Accessing structs

To access a nested member, use `.` to identify the fully-qualified `fieldName` and pass it to the corresponding setter or getter.

```
output.instance.setNumber('my_point.x', 10)
output.instance.setNumber('my_point.y', 20)

// alternatively:
output.instance.set('my_point.x', 10)
output.instance.set('my_point.y', 20)
```

It is possible to reset the value of a complex member back to its default:

```
output.instance.clearMember('my_point') // x and y are now 0
```

It is also possible to reset members using the `set` method:

```
output.instance.set('my_point', null)
```

Structs are set via JSON objects as follows:

```
output.instance.setFromJson({ 'my_point': { 'x':10, 'y':20 } })
```

When an member of a struct is not set, it retains its previous value. If we run the following code after the previous call to `setFromJson`:

```
output.instance.setFromJson({ 'my_point': { 'y': 200 } })
```

The value of `my_point` is now `{ 'x': 10, 'y':200 }`. If you do not want the values to be retained you must clear the value first (as described above).

It is possible to obtain the JSON object of a nested struct:

```
for (const sample of input.samples.validDataIter) {
   let point = sample.getJson('my_point')
}
```

`memberName` must be one of the following types: array, sequence, struct, value or union. If not, the call to `getJson` will fail:

```
 for (let sample of input.samples.validDataIter) {
    try {
       let long = sample.getJson('my_long')
    } catch (err) {
       // Error was thrown since my_long is a basic type
    }
}
```

It is also possible to obtain the JSON of a struct using the *`SampleIterator.get()`* method:

```
 for (const sample of input.samples.validDataIter) {
     let point = sample.get('my_point')
     // point is a JSON object
}
```

The same limitations described in *Accessing basic members (numbers, strings and booleans)* of using *`SampleIterator.get()`* apply here.

7.1.4 Accessing arrays and sequences

Use `fieldName[index]` to access an element of a sequence or array, where `0 <= index < length`:

```
let value = input.samples.get(0).getNumber('my_int_sequence[1]')
value = input.samples.get(0).getNumber('my_point_sequence[2].y')
```

To obtain the length of a sequence in an *Input()* sample, append # to the `fieldName`:

```
let length = input.samples[0].getNumber('my_int_sequence#')
```

Another option is to use `SampleIterator.getJson('fieldName')` to obtain a JSON object containing all of the elements of the array or sequence with name `fieldName`:

```
for (let sample of input.samples.validDataIter) {
    let thePointSequence = sample.getJson('my_point_sequence')
}
```

You can also get a specific element as a dictionary (if the element type is complex):

```
for (let sample of input.samples.validDataIter) {
   let pointElement = sample.getJson('my_point_sequence[1]')
}
```

In an *Output()*, sequences are automatically resized:

```
output.instance.setNumber('my_int_sequence[5]', 10) // length is now 6
output.instance.setNumber('my_int_sequence[4]', 9) // length still 6
```

You can clear a sequence:

```
output.instance.clearMember('my_int_sequence') // my_int_sequence is now empty
```

In JSON objects, sequences and arrays are represented as lists. For example:

```
output.instance.setFromJson({
    my_int_sequence: [1, 2],
    my_point_sequence: [{ x: 1, y: 1 }, { x: 2, y: 2 }]
})
```

Arrays have a constant length that can't be changed. When you don't set all the elements of an array, the remaining elements retain their previous values. However, sequences are always overwritten. See the following example:

```
output.instance.setFromJson({
    my_point_sequence: [{ x: 1, y: 1 }, { x: 2, y: 2 }],
    my_point_array: [{ x: 1, y: 1 }, { x: 2, y: 2 }, { x: 3, y: 3 }] })

output.instance.setFromJson({
    my_point_sequence: [{ x: 100 }],
    my_point_array: [{ x: 100}, { y: 200}] })
```

After the second call to `setFromJson`, the contents of `my_point_sequence` are `[{ x: 100, y: 0 }]`, but the contents of `my_point_array` are: `[{ x: 100, y: 1 }, { x: 2, y: 200 }, {x: 3, y: 3 }]`.

7.1.5 Accessing optional members

A optional member is a member that applications can decide to send or not as part of every published sample. Therefore, optional members may have a value or not. They are accessed the same way as non-optional members, except that `null` is a possible value.

On an Input, any of the getters may return `null` if the field is optional:

```
if (input.samples.get(0).getNumber('my_optional_long') == null) {
    console.log('my_optional_long not set')
}

if (input.samples.get(0).getNumber('my_optional_point.x') == null) {
    console.log('my_optional_point not set')
}
```

SampleIterator.getJson() returns a JSON object that doesn't include unset optional members.

To set an optional member on an Output:

```
output.instance.setNumber('my_optional_long', 10)
```

If the type of the optional member is not primitive, when any of its members is first set, the rest are initialized to their default values:

```
output.instance.setNumber('my_optional_point.x', 10)
```

If `my_optional_point` was not previously set, the previous code also sets `y` to 0.

There are several ways to reset an optional member. If the type is primitive:

```
output.instance.setNumber('my_optional_long', null) // Option 1
output.instance.clearMember('my_optional_long') // Option 2
output.instance.set('my_optional_long', null) // Option 3
```

If the member type is complex, all the above options except option 1 are available:

```
output.instance.clearMember('my_optional_point')
output.instance.set('my_optional_point', null)
```

Note that *Instance.setFromJson()* doesn't clear those members that are not specified; their values remain. For example:

```
output.instance.setNumber('my_optional_long', 5)
output.instance.setFromJson({ my_double: 3.3, my_long: 4 })
// my_optional_long is still 5
```

To clear a member, set it to `null` explicitly:

```
output.instance.setFromJson({ my_double: 3.3, my_long: 4, my_optional_long:␣
↪null })
```

For more information about optional members in DDS, see the *Getting Started Guide Addendum for Extensible Types*, Optional Members.

7.1.6 Accessing unions

In an *Output()*, the union member is automatically selected when you set it:

```
output.instance.setNumber('my_union.point.x', 10)
```

You can change it later:

```
output.instance.setNumber('my_union.my_long', 10)
```

In an *Input()*, you can obtain the selected member as a string:

```
if (input.samples.get(0).getString('my_union#') == 'point') {
    value = input.samples.get(0).getNumber('my_union.point.x')
}
```

7.1.7 Accessing key values of disposed samples

Using *Output.write()*, an *Output()* can write data, or dispose or unregister an instance. Depending on which of these operations is performed, the `instance_state` of the received sample will be `'ALIVE'`, `'NOT_ALIVE_NO_WRITERS'` or `'NOT_ALIVE_DISPOSED'`. If the instance was disposed, this `instance_state` will be `'NOT_ALIVE_DISPOSED'`. In this state, it is possible to access the key fields of the instance that was disposed.

Note: `SampleInfo.valid_data` will be false when the `SampleInfo.instance_state` is `'NOT_ALIVE_DISPOSED'`. In this situation it's possible to access the key fields in the received sample.

The key fields can be accessed as follows:

```
// The output and input are using the following type:
// struct ShapeType {
//     @key string<128> color;
//     long x;
//     long y;
//     long shapesize;
// }

output.instance.set('x', 4)
output.instance.set('color', 'Green')
// Assume that some data associated with this instance has already been sent
output.write({ action: 'dispose' })
await input.wait()
input.take()
let sample = input.samples.get(0)
```

(continues on next page)

(continued from previous page)

```
if (sample.info.get('instance_state') === 'NOT_ALIVE_DISPOSED') {
    // sample.info.get('valid_data') will be false in this situation
    // Only the key-fields should be accessed
    let color = sample.get('color') // 'Green'
    // The fields 'x','y' and 'shapesize' cannot be retrieved because they're
    // not part of the key
    // You can also call getJson() to get all of the key fields in a JSON␣
↪object.
    // Again, only the key fields returned within the JSON object should
    // be used.
    let keyValues = sample.getJson() // { color: 'Green', x: 0, y: 0,␣
↪shapesize: 0 }
}
```

Warning: When the sample has an instance state of `'NOT_ALIVE_DISPOSED'` only the key fields should be accessed.

7.2 Threading model

All operations on *different* *`Connector()`* instances are thread-safe.

Operations on the *same* *`Connector()`* instance or any contained *`Input()`* or *`Output()`* are, in general, not protected for multi-threaded access. The only exceptions are the following *wait* operations with the caveats mentioned below.

Note: If you are using the event-based functionality (e.g., `connector.on(on_data_available, () => {})`), refer to the additional restraints described in the *Additional considerations when using event-based functionality* section below.

Thread-safe operations:

- *`Connector.wait()`* (wait for data on any `Input`)
- *`Output.wait()`* (wait for acknowledgments)
- *`Output.waitForSubscriptions()`* (wait to (un)match a subscription)
- *`Input.wait()`* (wait for data on this `Input`)
- *`Input.waitForPublications()`* (wait to (un)match a publication)

Note: All of the operations listed above are asynchronous (and return a `Promise` which will eventually be resolved or rejected).

Note: *Output.write()* can block the current thread under certain circumstances, but *Output.write()* is not thread-safe.

Warning: *Input.wait()* and *Input.waitForPublications()* are not reentrant (it is currently not possible to have more than one Promise pending on *Input.wait()* or *Input.waitForPublications()*). Since internally the same resource is used for both of these operations, it is not possible to wait on both *Input.wait()* or *Input.waitForPublications()* simultaneously.

For example, the following code will throw an *DDSError()*:

```
const waitForDiscovery = async () => {
   try {
      await input.waitForSubscriptions()
   } catch (err) {
      console.log('Caught error: ' + err)
   }
}

const wait = async () => {
   try {
      await input.wait()
   } catch (err) {
      console.log('Caught error: ' + err)
   }
}

waitForDiscovery()
wait()
```

The `input.wait` call within the asynchronous function `wait` will fail since there is a simultaneous request to `input.waitForSubscriptions`. This can be avoided by ensuring you only have a single `wait` operation pending at a time:

```
const waitForDiscovery = async () => {
   try {
      await input.waitForSubscriptions()
   } catch (err) {
      console.log('Caught error: ' + err)
   }
}

const wait = async () => {
   try {
      await input.wait()
   } catch (err) {
      console.log('Caught error: ' + err)
   }
}

const myApplication = async () => {
   await waitForDiscovery()
   await wait()
}

myApplication()
```

The same limitation exists between *Output.wait()* and *Output.waitForSubscriptions()*.

7.2.1 Additional considerations when using event-based functionality

If using event-based notifications (that is, if you have installed a listener for the on_data_available event on a *Connector()*, as explained in *Reading or taking the data*), there are additional restrictions to be aware of.

It is possible to install multiple listeners for the on_data_available event:

```
connector.on('on_data_available', () => {
  // Read the samples so that they remain available within the Input
  input.read()
  doSomething(input.samples)
})
connector.on('on_data_available', () => {
  // Take the samples to remove them from the Input. Since event callbacks
  // are, by default, run in the order they are registered, there is no risk
  // that this is run before the above listener
  input.take()
  doSomethingElse(input.samples)
})
```

In the above example, when data is received, both the installed callbacks will be run. These callbacks are run sequentially, so it is not necessary to protect them manually at an application level.

It is not possible to call *Connector.wait()* if there is an installed listener for the on_data_available event. This is due to the fact that while the on_data_available listener is installed, the resource required internally for *Connector.wait()* is busy.

In your application, if you remove all the registered listeners for on_data_available and later need to re-add them (or wait for data using *Connector.wait()*), it is necessary to call *Connector.waitForCallbackFinalization()*. This method returns a Promise that will resolve once the resources used internally by the *Connector()* are no longer in use:

```
connector.on('on_data_available', handleDataCallback)
connector.off('on_data_available', handleDataCallback)
// Since we removed the only callback for the on_data_available event, we must
// now wait for the Promise to resolve before re-adding a new callback
await connector.waitForCallbackFinalization()
connector.on('on_data_available', newHandleDataCallback)
```

Warning: It is important to note that *Connector.waitForCallbackFinalization()* does **not** free any resources. It should only be used for notification of when a *Connector()* can be re-used for other wait operations. It is still necessary to call *Connector.close()* to free the resources.

If you install a on_data_available listener, you need to wait for the Promise returned by *Connector.close()* to resolve before continuing with the application shutdown procedure. This allows

the *Connector()* to synchronize its shutdown with the listener:

```
connector.close()
  .then(() => {
    // continue with application shutdown
  })
```

7.3 Error Handling

Connector reports internal errors in *RTI Connext DDS* by raising an `rticonnextdds_connector.DDSError()`. This exception may contain a description of the error.

A subclass, `rticonnextdds_connector.TimeoutError()`, indicates that an operation that can block has timed out.

Other errors may be raised as `Error`, `TypeError`, or other built-in Node.js exceptions.

If the `on_data_available` event is used to be notified of new data, errors will be propagated through the `error` event. If the `error` event is emitted and the object that emits it has no attached listeners for the `error` event, the program will be terminated with a non-zero error code and the stack trace will be printed. For more information please refer to the documentation for the EventEmitter class.

7.3.1 Class reference: DDSError, TimeoutError

Error class

class DDSError(*message*, *extra*)

An error originating from the *RTI Connext DDS* Core

This error is thrown when an error is encountered from within one of the APIs within the *RTI Connext DDS* Core.

TimeoutError class

class TimeoutError(*message*, *extra*)

A timeout error thrown by operations that can block

This error is thrown when blocking errors timeout.

7.4 Connext DDS Features

Because *RTI Connector* is a simplified API, it provides access to a subset of the features in *RTI Connext DDS*.

In addition to the functionality described in the rest of this documentation, this section summarizes the support that *RTI Connector* provides for some notable *RTI Connext DDS* features.

7.4.1 General features

Table 1: General Features

Feature	Level of support	Notes
Quality of Service (QoS)	Partial	Most QoS policies are supported because they can be configured in XML, but those that are designed to be mutable can't be changed in *Connector*. QoS policies that require a supporting API may have limited or no support. A few examples of QoS policies that are fully supported in *Connector*: • Reliability • Durability • History • Ownership A few examples of QoS policies that are supported but can't be changed in *Connector* even though they are mutable by design and changeable in other APIs: • Partition • Lifespan • Ownership Strength • Property and User Data • Time-Based Filter A few examples of QoS policies that have limited support because they require a supporting API that is not available in *Connector*: • Batch - fully supported except that manual flushing is not available. • Entity Factory - *autoenable_created_entities* can be set to *false* only for a *subscriber*, in order to enable an *Input* only when `Connector.getInput()` is called. • Property - Properties can be set in XML, but they can't be looked up in *Connector*. Topic Qos is not supported in *Connector*. Use DataReader QoS and DataWriter QoS directly.
Entity Statuses	Partial	Only `Input.wait()` (data available), `Input.waitForPublications()`, `Output.waitForSubscriptions()` are supported
Managing Data Instances	Partial	On an `Output`, it is possible to dispose or unregister an instance (see `Output.write()`). Instances are automatically registered when first written. On an `Input` the instance state can be obtained, alongside the key fields of a disposed instance (see *Accessing key values of disposed samples*). Instance handles are not exposed.
Application Acknowledgment	Partial	*DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE* is supported. If enabled, when a call to `Input.take()` or `Input.read()` is followed by another call, the second one automatically acknowledges the samples read in the first one. *DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE* is not supported.
Request-Reply	Partial	The correlation between two samples can be established at the application level: • The *Requester* application writes by calling `Output.write()` with the parameter `identity=A` (the *request* sample). • The *Replier* application receives the *request* sample, obtains the

7.4.2 Features related to sending data

Table 2: Features Related to Sending Data

Feature	Level of support	Notes
Waiting for Acknowledgments	Supported	See `Output.wait()`.
Coherent Sets	Not supported	API not available.
Flow Controllers	Partial	Most functionality is available via XML QoS configuration.
Asserting Liveliness Manually	Not supported	API not available.
Collaborative DataWriters	Limited	The virtual GUID can be set per writer in XML, but not per sample.

7.4.3 Features related to receiving data

Table 3: Features Related to Receiving Data

Feature	Level of support	Notes
Content-Filtered Topics	Partial	Configurable in XML but it can't be modified after creation
Sample Info	Partial	See `SampleIterator.info`
Query Conditions	Not supported	API not available
Group-Ordered Access	Not supported	API not available
Waiting for Historical Data	Not supported	API not available

7.4.4 Features related to the type system

Table 4: Features Related to the Type System

Feature	Level of support	Notes
DDS type system	Supported	*Connector* can use any DDS type. Types are defined in XML.
Type extensibility	Supported	*Connector* supports type extensibility, including mutable types in the XML definition of types. It also supports type-consistency enforcement, sample-assignability enforcement; these checks are performed by the *RTI Connext DDS* Core.
Optional members	Supported	See *Accessing optional members*

Default values	Supported	For example, to declare a default value for a member:
		`<struct name= "MyType" extensibility="mutable">`
		`  <!-- ... -->`
		`  <member name="my_int" type="int32" default="20" />`
		`</struct>`
		Now the value for `my_int` when you call *`Output.write()`* without setting it explicitly is 20. And when you receive a data sample in an `Input` from a *Publisher* whose type is compatible but doesn't have the field `my_int`, the value you receive is 20.

Unbounded data	Supported	To declare an unbounded sequence or string, set its max length to *-1*:
		`<struct name= "MyType">`
		`<member name="my_unbounded_int_sequence" sequenceMaxLength="-1" type="int32"/>`
		`<member name="my_bounded_int_sequence" sequenceMaxLength="10" type="int32"/>`
		`</struct>`
		For any `Output` using a topic for a type with unbounded members, set the following Property QoS policy:
		`<datawriter_qos>`
		`<!-- ... -->`
		`<property>`
		`<value>`
		`<element>`
		`<name>`
		`dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size`
		`</name>`
		`<value>4096</value>`
		`</element>`
		`</value>`
		`</property>`
		`</datawriter_qos>`
		The value *4096* is a threshold that indicates *RTI Connext DDS* to allocate memory dynamically for data samples that exceed that size. For samples below that threshold, memory comes from pre-allocated buffers.
		If the unbounded member is a *key*, then in any `Input` that uses the type, set the following:
		`<datareader_qos>`
		`<!-- ... -->`
		`<property>`

7.4.5 Loading Connext DDS Add-On Libraries

Connector supports features that require the loading of additional *Connext DDS* libraries, such as Monitoring and Security Plugins.

The Monitoring and Security plugins are configured in XML, as described in the previous links.

To use RTI Connext DDS add-ons you need an RTI Connext DDS installation. To configure your environment so that Connector can load these additional libraries:

- Set your environment using:

```
$ source <Connext DDS installation directory>/resource/scripts/rtisetenv_
↪<architecture>.bash
```

  or:

```
> <Connext DDS installation directory>\resource\scripts\rtisetenv_
↪<architecture>.bat
```

- Or set your system's library path to:

```
<Connext DDS installation directory>\lib\<architecture>\
```

Note: Each version of Connector can only load add-on libraries from its corresponding Connext DDS release. You can see this correspondence in the *Release Notes*. For example, Connector 1.1.0 can only load Connext DDS 6.1.0 add-on libraries.

Chapter 8

Release Notes

8.1 Supported Platforms

RTI Connector for JavaScript has been tested with Node.js versions 10.22.0, 11.15.0 and 12.13.0.

Connector uses a native C library that works on most Windows®, Linux® and macOS® platforms. It has been tested on the following systems:

Linux

- CentOS™ 6.0, 6.2-6.4, 7.0 (x64)
- Red Hat® Enterprise Linux 6.0-6.5, 6.7, 6.8, 7, 7.3, 7.5, 7.6, 8 (x64)
- SUSE® Linux Enterprise Server 12 SP2 (x64)
- Ubuntu® 14.04, 16.04, 18.04, 20.04 LTS (x64)
- Ubuntu 16.04, 18.04 LTS (64-bit Arm® v8)
- Ubuntu 18.04 LTS (32-bit Arm v7)
- Wind River® Linux 8 (Arm v7) (Custom-supported platform)

macOS

- macOS 10.13-10.15 (x64)

Windows

- Windows 8 (x64)
- Windows 10 (x64)
- Windows Server 2012 R2 (x64)
- Windows Server 2016 (x64)

Connector is supported in other languages in addition to JavaScript, see the main Connector repository.

8.2 Version 1.1.0

RTI Connector 1.1.0 is built on RTI Connext DDS 6.1.0.

8.2.1 What's New in 1.1.0

Support added for ARMv8 architectures

Connector for JavaScript now runs on ARMv8 architectures. Native libraries built for ARMv8 Ubuntu 16.04 are now shipped alongside Connector. These libraries have been tested on ARMv8 Ubuntu 16.04 and ARMv8 Ubuntu 18.04.

Support added for Node.js version 12

Previously, Node.js version 12 was not supported in *Connector* for JavaScript. Support has been added for Node.js version 12 (the current LTS), and support has been dropped for Node.js version 8 (which has been deprecated). Note that Node.js version 12.19.0 is incompatible with Connector for JavaScript due to a regression in that release of Node.js. Versions 12.18.x and 12.20.x are compatible with Connector for JavaScript.

Sample state, instance state and view state can now be obtained in Connector

The `SampleInfo()` class in *Connector* has been extended to provide access to the sample state, view state, and instance state fields. These new fields work the same as the existing fields in the structure (in *Connector* for Python they are the keys to the dictionary, in *Connector* for JavaScript they are the keys to the JSON Object). See *Accessing sample meta-data* for more information on this new feature.

Support for accessing the key values of disposed instances

Support for disposing instances was added in *Connector* 1.0.0. However, it was not possible to access the key values of the disposed instance. This functionality is now available in the Python and JavaScript bindings. When a disposed sample is received, the key values can be accessed. The syntax for accessing these key values is the same as when the sample contains valid data (i.e., using type-specific getters, or obtaining the entire sample as an object). When the instance state is NOT_ALIVE_DISPOSED, only the key values in the sample should be accessed. See *Accessing key values of disposed samples* for more information on this new feature.

Connector for Javascript dependencies now locked to specific versions

`package-lock.json` has been committed, fixing the versions of *Connector for Javascript's* dependencies.

Support for Security, Monitoring and other Connext DDS add-on libraries

It is now possible to load additional Connext DDS libraries at runtime. This means that Connext DDS features such as Monitoring and Security Plugins are now supported. Refer to *Loading Connext DDS Add-On Libraries* for more information.

8.2.2 What's Fixed in 1.1.0

Creating two instances of Connector resulted in a license error

Under some circumstances, it was not possible to create two *Connector* objects. The creation of the second *Connector* object failed due to a license error. This issue affected all of the *Connector* APIs (Python, JavaScript). This issue has been fixed.

[RTI Issue ID CON-163]

Some larger integer values may have been corrupted by Connector's internal JSON parser

The internal JSON parser used in *Connector* failed to identify integer numbers from double-precision floating-point numbers for certain values. For example, if a number could not be represented as a 64-bit integer, the parser may have incorrectly identified it as an integer, causing the value to become corrupted. This problem has been resolved.

[RTI Issue ID CON-170]

Support for loading multiple configuration files

A *Connector* object now supports loading multiple files. This allows separating the definition of types, QoS profiles, and *DomainParticipants* into different files:

```
const connector = new rti.Connector("my_profiles.xml;my_types.xml;my_
↪participants.xml", configName)
```

[RTI Issue ID CON-209]

Creating a Connector instance with a participant_qos tag in the XML may have resulted in a license error

In some cases, if the XML configuration file of *Connector* contained a <*participant_qos*> tag within the definition of the *DomainParticipant*, the creation of the *Connector* would fail with a "license not found" error. This problem has been resolved.

[RTI Issue ID CON-214]

Websocket example may have failed to run

The websocket example (available only in *Connector for Javascript*) may have failed to run due to one of its dependencies, socket.io, removing a public API. This problem has been resolved.

[RTI Issue ID CON-217]

8.3 Version 1.0.0

1.0.0 is the first official release of *RTI Connector for JavaScript* as well as RTI Connector for Python.

If you had access to previous experimental releases, this release makes the product more robust, modifies many APIs and adds new functionality. However the old APIs have been preserved for backward compatibility as much as possible.

RTI Connector 1.0.0 is built on RTI Connext DDS 6.0.1.

Chapter 9

Copyrights and License

License

RTI Connector for JavaScript and RTI Connector for Python are part of the Connext DDS Professional Package. If you have a valid license for the RTI Connext DDS Professional Package, such license shall govern your use of RTI Connector for Python and RTI Connector for JavaScript. All other use of this software shall be governed solely by the terms of RTI's Software License for Non-Commercial Use #4040, included at the top level of the Connector for JavaScript repository.

Trademarks

RTI, Real-Time Innovations, Connext, NDDS, the RTI logo, 1RTI and the phrase, "Your Systems. Working as one." are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished under and subject to the RTI software license agreement. The software may be used or copied only under the terms of the license agreement.

This is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.

The security features of this product include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). his product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Technical Support Real-Time Innovations, Inc. 232 E. Java Drive Sunnyvale, CA 94089 Phone: (408) 990-7444 Email: support@rti.com Website: https://support.rti.com/

External Third-Party Software Used in Connector

Lua

- The source code of this software is used to build the native libraries provided by *RTI Connector*.
- License (https://www.lua.org/license.html): Copyright © 1994–2021 Lua.org, PUC-Rio.

  Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

  The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

json-parser

- The source code of this software (from https://github.com/udp/json-parser) is used to build the native libraries provided by *RTI Connector*.
- License:

  Copyright (C) 2012, 2013, 2014 James McLaughlin et al. All rights reserved. https://github.com/udp/json-parser

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PRO-

CUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- genindex
- search

To learn more about *RTI Connext DDS*, see the RTI Connext DDS Getting Started Guide. More documentation is available in the RTI Community Portal.

Index