RTI Connext Getting Started

C#

Version 7.3.1

Contents

1 Before You Get Started **1**
- 1.1 What is Connext? . . . 1
- 1.2 Downloading Connext . . . 2
- 1.3 Installing Connext . . . 2
  - 1.3.1 Installing a Host . . . 3
  - 1.3.2 Installing additional packages with a GUI . . . 3
  - 1.3.3 Installing additional packages from a Command Line . . . 4
  - 1.3.4 Paths Mentioned in Documentation . . . 4
- 1.4 Setting Up a License . . . 5
  - 1.4.1 Setting up a License for the C# API . . . 5
  - 1.4.2 Setting up a License for the tools and other features . . . 6
- 1.5 Checking What is Installed . . . 7
- 1.6 Where Do I Get More Help? . . . 7

2 Publish/Subscribe **8**
- 2.1 Introduction to DataWriters, DataReaders, and Topics . . . 9
- 2.2 Hands-On 1: Your First DataWriter and DataReader . . . 10
  - 2.2.1 Clone Repository . . . 10
  - 2.2.2 Set Up Environment Variables . . . 11
  - 2.2.3 Run Code Generator . . . 11
    - Overview of Generated and Example Code . . . 13
  - 2.2.4 Open/Modify Publishing Application . . . 13
  - 2.2.5 Open/Modify Subscribing Application . . . 15
    - Details of Receiving Data . . . 15
  - 2.2.6 Compile Your Changes . . . 17
  - 2.2.7 Run the Applications . . . 17
  - 2.2.8 Taking It Further . . . 18
    - Start up Multiple Publishing or Subscribing Applications . . . 18
    - Publish/Subscribe across Multiple Machines . . . 20
    - Create Multiple DataWriters, DataReaders, and Topics in a Single Application . . . 20
- 2.3 Troubleshooting . . . 21
  - 2.3.1 Why aren't my applications communicating? . . . 21
  - 2.3.2 How do I set my discovery peers? . . . 21
  - 2.3.3 Why does the DataReader miss the first samples? . . . 22
  - 2.3.4 Why do I get a "No source for License information" error? . . . 22
- 2.4 Hands-On 2: Viewing Your Data . . . 23
  - 2.4.1 Open Admin Console . . . 23

2.4.2 Choose Automatically Join . . . 24
2.4.3 Switch to Data Visualization Perspective . . . 24
2.4.4 Open Topic Data Tab . . . 25
2.4.5 Subscribe to “HelloWorld Topic” . . . 25
2.4.6 Use Topic Data Tab . . . 26
2.4.7 Use Admin Console across Machines . . . 26
2.5 Next Steps . . . 27

3 Data Types **28**
3.1 Common IDL Types . . . 30
3.2 Introduction to Data Flows . . . 30
3.3 Hands-On 1: Streaming Data . . . 31
3.3.1 Run Code Generator . . . 31
3.3.2 Modify for Streaming Data . . . 32
3.3.3 Run the Applications . . . 34
3.4 Publishers, Subscribers, and DomainParticipants . . . 35
3.5 Hands-On 2: Add a Second DataWriter . . . 37
3.5.1 Add the New DataWriter . . . 37
3.5.2 Visualize the Data in rtiddsspy . . . 38
3.6 Next Steps . . . 41

4 Keys and Instances **42**
4.1 Why and How Do We Use Instances? . . . 44
4.1.1 Writing an Instance . . . 44
4.1.2 Reading an Instance . . . 46
4.1.3 Instance Lifecycle . . . 47
4.2 Example: Chocolate Factory . . . 48
4.2.1 Chocolate Factory: System Overview . . . 48
4.2.2 Chocolate Factory: Data Overview . . . 49
4.3 Hands-On 1: Build the Applications and View in Admin Console . . . 50
4.3.1 Build the Applications . . . 52
4.3.2 Run Multiple Copies of the Tempering Application . . . 54
4.3.3 View the Data in Admin Console . . . 55
4.4 Hands-On 2: Run Both Applications . . . 57
4.4.1 Run Monitoring and Tempering Applications . . . 57
4.4.2 Review the Tempering Application Code . . . 59
4.5 Hands-On 3: Dispose the ChocolateLotState . . . 60
4.5.1 Add Code to Tempering Application to Dispose ChocolateLotState . . . 61
4.5.2 Detect the Dispose in the Monitoring Application . . . 61
4.5.3 Run the Applications . . . 63
4.6 Hands-On 4: Debugging the System and Completing the Application . . . 64
4.6.1 Debug in Admin Console . . . 64
4.6.2 Add the ChocolateTemperature DataReader . . . 66
4.6.3 Run the Applications . . . 68
4.7 Next Steps . . . 69

5 Basic QoS **70**
5.1 Request-Offered QoS Policies . . . 71

5.2 Some Basic QoS Policies . . . 73
5.2.1 Reliability and History QoS Policies . . . 73
"Best Effort" Reliability . . . 73
"Reliable" Reliability + "Keep All" History . . . 74
"Reliable" Reliability + "Keep Last" History . . . 74
Summary . . . 76
5.2.2 Resource Limits QoS Policy . . . 77
5.2.3 Durability QoS Policy . . . 78
5.2.4 Deadline QoS Policy . . . 82
5.2.5 QoS Patterns Review . . . 83
5.3 QoS Profiles . . . 84
5.4 Hands-On 1: Update One QoS Profile in the Monitoring/Control Application . . . 84
5.5 Hands-On 2: Incompatible QoS in Admin Console . . . 89
5.6 Hands-On 3: Incompatible QoS Notification . . . 92
5.7 Hands-On 4: Using Correct QoS Profile . . . 93
5.8 Next Steps . . . 95

6 ContentFilteredTopics 96
6.1 The Complete Chocolate Factory . . . 97
6.2 Hands-On 1: Update the ChocolateLotState DataReader with a ContentFilteredTopic . . . 99
6.3 Hands-On 2: Review the Temperature DataReader's ContentFilteredTopic . . . 101
6.4 Hands-On 3: Review the Larger System in Admin Console . . . 104

7 Discovery 106
7.1 Domains . . . 107
7.2 Initial Peers . . . 108
7.3 Hands-On 1: Troubleshooting Discovery . . . 109
7.4 Hands-On 2: Start Applications in Different Domains . . . 111
7.5 Next Steps . . . 112

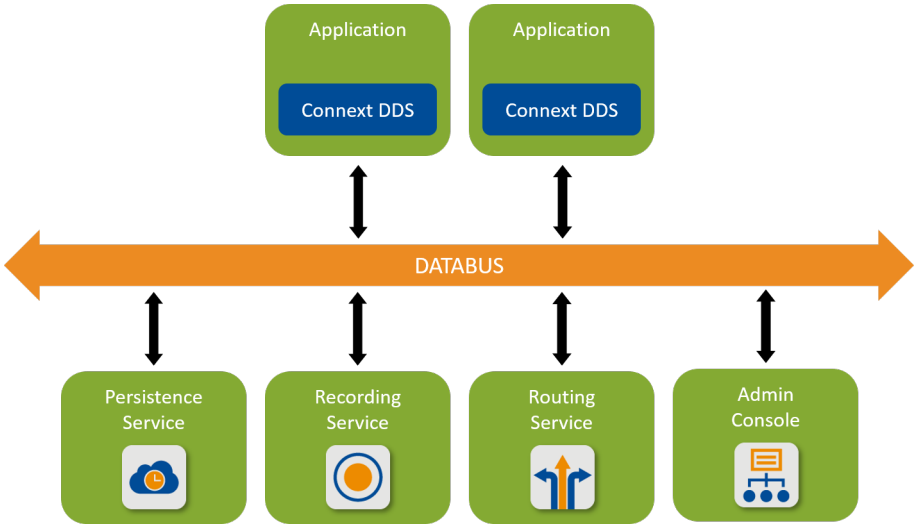
8 Next Steps 113
8.1 Documentation . . . 113
8.2 Examples . . . 114
8.3 Updates . . . 114

9 Copyrights and Notices 115

Chapter 1

Before You Get Started

1.1 What is Connext?



RTI® Connext® is a connectivity framework for building distributed applications with requirements for high performance and scalability. It includes these components:

- An **SDK** that provides you with APIs to help you send and receive data using the communication patterns described in this documentation (see *Next Steps*). These APIs allow you to connect your own applications to other applications on the databus.
- **Tools** that help you visualize your data and debug your distributed system.
- **Infrastructure Services** that can perform dedicated functions in your system, such as recording, bridging, and persisting data.

1.2 Downloading Connext

If you haven't already purchased a *Connext* bundle, you can follow this Getting Started guide with an evaluation bundle. To obtain an evaluation bundle, click the Free Trial button at https://www.rti.com/. Fill out the brief form, and you will receive an evaluation package shortly.

1.3 Installing Connext

Note: These instructions apply to *Connext* 6.1.1 and newer.

To develop C# applications, you will need the following *Connext* bundles:

- A **host** bundle that contains files such as documentation, the code generator, other tools, and infrastructure services executables. The host bundle is provided in a **.run** or **.exe** file that will run an installer.

  Host bundles are named:

  Linux

  `rti_connext_dds-<version>-<package_type>-host-<host-platform>.run`

  macOS

  `rti_connext_dds-<version>-<package_type>-host-<host-platform>.dmg`

  Windows

  `rti_connext_dds-<version>-<package_type>-host-<host-platform>.exe`

  The `<package_type>` is usually `pro`.

  The `<host_platform>` depends on your development platform, such as `x64Win64` for a 64-bit Windows® platform.

- The **NuGet packages** required to develop *Connext* C# applications. There are two ways to obtain these NuGet packages:

  – Install `rti_dotnet_support-<version>.rtipkg` to add the NuGet packages to your installation directory (see *Installing additional packages with a GUI* or *Installing additional packages from a Command Line*).

  or

  – Let .NET automatically download the NuGet packages from nuget.org when you build your C# applications. The nuget.org libraries require a license file when you run your application (see *Setting up a License for the C# API*).

  Note that the nuget.org packages allow writing C# applications without a host installation, but this guide uses the code generator and some tools that are included in the host package.

- If you plan to develop C, C++, or Java applications or use add-on libraries you will also need a **target** bundle. See the C++ version of this Getting Started Guide for more information.

1.3.1 Installing a Host

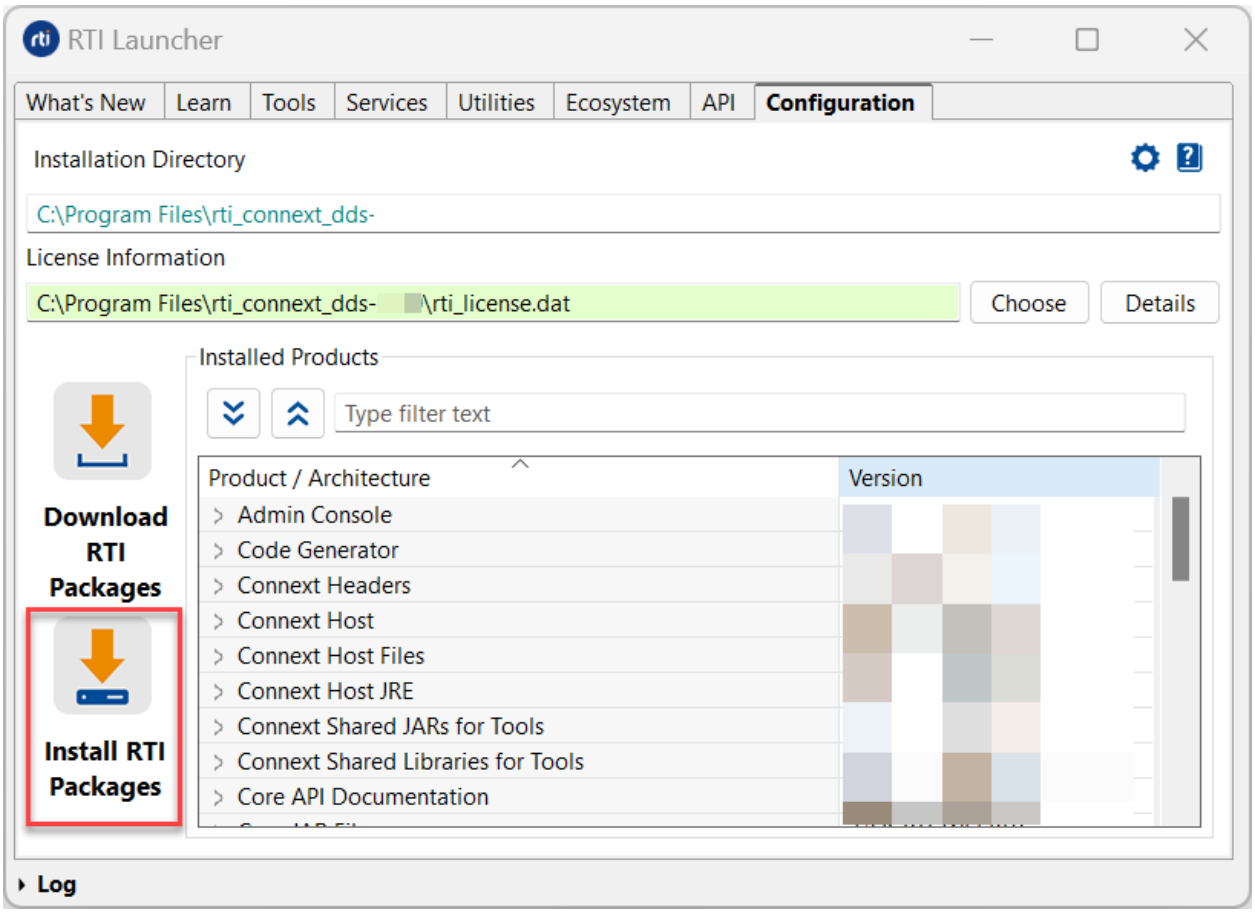
The host bundle is an application; thus, it can be started from a GUI or command line. To install the host bundle, do either of the following:

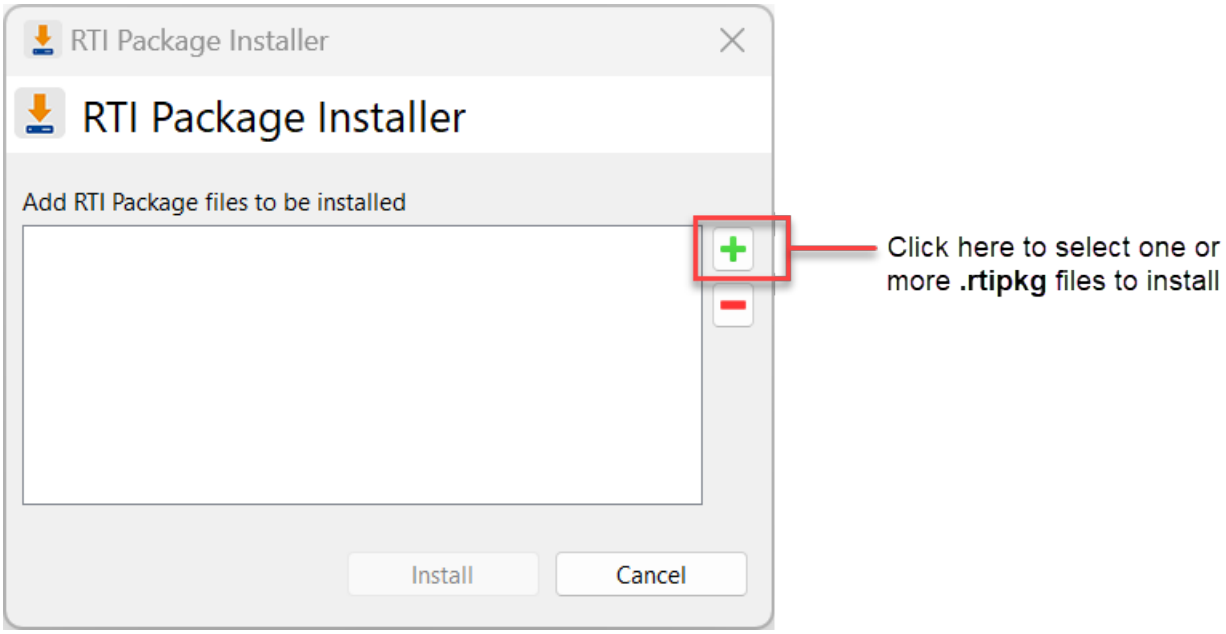
- Double-click the installer.
- Run the installation script from a command prompt. See Installing RTI Connext, in the RTI Connext Installation Guide.

1.3.2 Installing additional packages with a GUI

After you install the host bundle, you'll have a tool called *RTI Launcher*. (See Starting Launcher, in the RTI Launcher User's Manual.)

To install additional packages from the *Launcher* tool, open the Configuration tab, and select "Install RTI Packages." This will open a dialog that allows you to select one or more **.rtipkg** files that you would like to install.





1.3.3 Installing additional packages from a Command Line

To install additional packages from the command line, type:

Linux

```
$ <installdir>/bin/rtipkginstall <path to rtipkg>
```

macOS

```
$ <installdir>/bin/rtipkginstall <path to rtipkg>
```

Windows

```
> <installdir>\bin\rtipkginstall <path to rtipkg>
```

1.3.4 Paths Mentioned in Documentation

This documentation refers to the following directories, depending on your operating system:

Linux

`$NDDSHOME` This refers to the installation directory for *Connext*.

The default installation paths are:

- Non-root user:

  `/home/<your user name>/rti_connext_dds-<version>`

- Root user:

  `/opt/rti_connext_dds-<version>`

$NDDSHOME is an environment variable set to the installation path.

macOS

$NDDSHOME This refers to the installation directory for *Connext*.

The default installation path is:

/Applications/rti_connext_dds-<version>

$NDDSHOME is an environment variable set to the installation path.

Windows

%NDDSHOME% This refers to the installation directory for *Connext*.

The default installation paths are:

- User without Administrator privileges:

  <your home directory>\rti_connext_dds-<version>

- User with Administrator privileges:

  "C:\Program Files\rti_connext_dds-<version>"

%NDDSHOME% is an environment variable set to the installation path.

Note: When using a command prompt to enter a command that includes the path C:\Program Files (or any directory name that has a space), enclose the path in quotation marks. For example: "C:\Program Files\rti_connext_dds-version\bin\rtilauncher.bat". Or if you have defined the NDDSHOME environment variable: "%NDDSHOME%\bin\rtilauncher.bat".

Sometimes this documentation uses <NDDSHOME> to refer to the installation path. Whenever you see <NDDSHOME> used in a path, replace it with $NDDSHOME for Linux or macOS systems, with %NDDSHOME% for Windows systems, or with your installation path.

1.4 Setting Up a License

1.4.1 Setting up a License for the C# API

If you're using the NuGet packages from your **host installation**, you don't need a license file to run your applications.

If you're using the NuGet packages from **nuget.org**, your applications need to load a license file every time they run. To obtain an evaluation license, click the Free Trial button at https://www.rti.com/.

There are several ways to specify your license file:

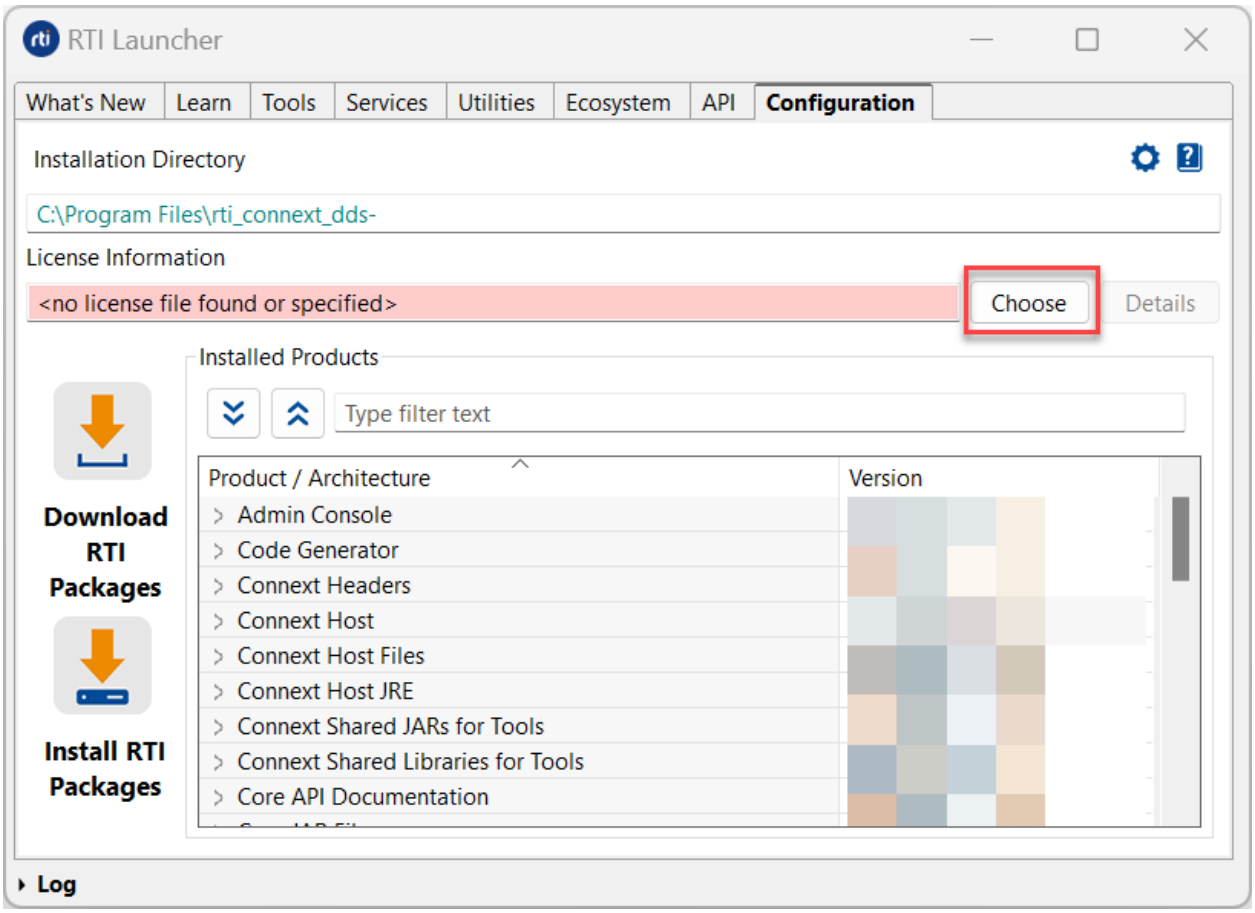
- You can copy the *rti_license.dat* file to the location where you run your application.
- You can set the RTI_LICENSE_FILE environment variable to the full path of your license file, including the file name.

- You can set the `DomainParticipantQos` to point to your license file (see the License Management section in the RTI Connext Installation Guide).

1.4.2 Setting up a License for the tools and other features

Most installations require a license file to run the tools or features included in the *Connext* platform. If your distribution requires a license file, you will receive one from RTI via e-mail.

The easiest way to permanently configure your license file is using *Launcher*, as shown below:



If you do not want to use *Launcher*, you can also install a license by placing it in one of these two locations:

- `<installation directory>/rti_license.dat`
- `<workspace directory>/rti_license.dat`

The workspace directory is in this location by default, depending on your operating system:

Linux

`/home/<your username>/rti_workspace/`

macOS

```
/Users/<your username>/rti_workspace/
```

Windows

```
<your Windows documents folder>\rti_workspace\
```

A third way to install a license is to configure the environment variable `RTI_LICENSE_FILE` to point to your license file.

For more details on how to install a license file, see the License Management section in the RTI Connext Installation Guide.

1.5 Checking What is Installed

To find out what target libraries or add-ons you have installed, you can use the *Launcher* tool. See Starting Launcher, in the RTI Launcher User's Manual. Once in *Launcher*, open the Configuration tab.

1.6 Where Do I Get More Help?

The full RTI Connext Installation Guide contains more information:

- Installer command-line options
- Controlling the location of the RTI Workspace directory
- Additional license management options
- Special backup of RTI libraries
- How to uninstall *Connext*

Additional documentation and user forums can be found on community.rti.com.

Continue to *Publish/Subscribe* to start learning about the capabilities and features of *Connext*.

Chapter 2

Publish/Subscribe

Prerequisites	• Install git • Install the .NET 5 SDK (the C# API supports more .NET versions, but the examples in this guide are set up to build for .NET 5) • Install release 7.3.1 (see *Before You Get Started*) • Clone repository from GitHub here
Time to complete	30 minutes
Concepts covered in this module	• Introduction to publish/subscribe • Introduction to *DataWriters*, and *DataReaders*, and *Topics* • Using the code generator • Using Waitsets • Viewing your data in *RTI Admin Console*

The most basic communication pattern supported by *RTI® Connext®* is the publish/subscribe model. Publish/Subscribe is a communications model where data producers "publish" data and data consumers "subscribe" to data. These publishers and subscribers don't need to know about each other ahead of time; they discover each other dynamically at runtime. The data they share is described by a "topic," and publishers and subscribers send and receive data only for the topics they are interested in. In this pattern, many publishers may publish the same topic, and many subscribers may subscribe to the same topic. Subscribers receive data from all of the publishers that they share a topic with. Publishers send data directly to subscribers, with no need for a broker or centralized application to mediate communications.

2.1 Introduction to DataWriters, DataReaders, and Topics

In DDS, the objects that actually publish data are called *DataWriters*, and the objects that subscribe to data are *DataReaders*. *DataWriters* and *DataReaders* are associated with a single *Topic* object that describes that data. (DDS also has *Publisher* and *Subscriber* objects, but we will talk about them later.) An application typically has a combination of *DataWriters* and *DataReaders*.

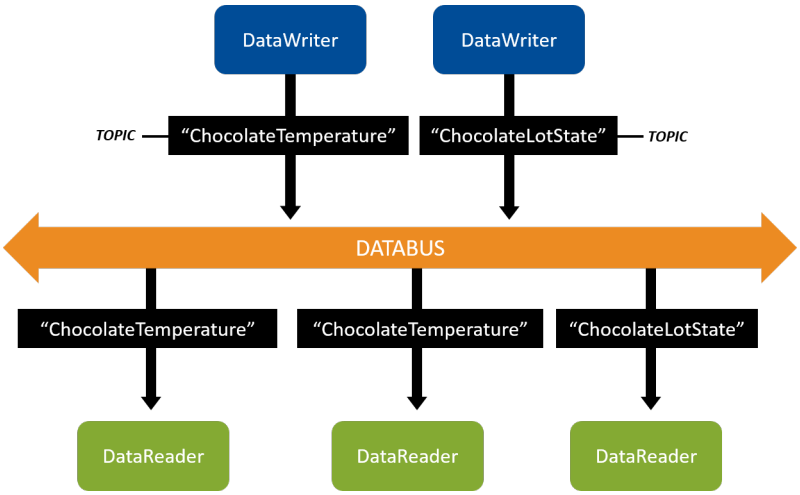


Figure 2.1: *DataWriters* write data and *DataReaders* read data of a *Topic*. *DataWriters* of the "ChocolateTemperature" *Topic* communicate with *DataReaders* of the "ChocolateTemperature" *Topic*. *DataWriters* of the "ChocolateLotState" *Topic* communicate with *DataReaders* of the "ChocolateLotState" *Topic*.

In a chocolate factory, for example, there might be a sensor that measures and publishes the current temperature of the tempering machine. Other applications monitor the temperature by subscribing to it. In this example, your *Topic* might be "ChocolateTemperature." The sensor's *DataWriter* will be associated with the "ChocolateTemperature" *Topic*. In a similar way, other *DataWriters* and *DataReaders* share different types of data using additional *Topics*.

Connext is responsible for discovering *DataWriters* and *DataReaders* in a system, checking if they have a matching *Topic* (and compatible quality of service, which we will discuss later) and then providing the communication between those *DataWriters* and *DataReaders*. Logically, this means you can visualize your applications as having *DataWriters* and *DataReaders* that connect to a "databus," because your applications are not specifying exactly which other applications they communicate with – they only specify which *Topics* they read from and write to the databus, and *Connext* sets up the communication. Note that there is no "databus" object in your system – it is a logical way to visualize systems in which you don't have to configure each communication path.

2.2 Hands-On 1: Your First DataWriter and DataReader

We are going to start with a simple "Hello World" application to show how to use the code generator, and how to create a *DataWriter* and a *DataReader*.

Tip: By the end of this exercise, a publishing application will send data, and a subscribing application will receive and print it to the console using `Console.WriteLine()`.

2.2.1 Clone Repository

Get the files you need to perform the hands-on exercises.

Note: Always use the version of the examples that matches your *Connext* release. By default, the GitHub repository branch is set to `master`, which is always the latest release. If your version of *Connext* is not the latest release, select the release version you want, such as `release/<version>`.

Do one of the following:

- Download the repository as a Zip file from the repository webpage here, for your desired release (e.g., `release/<version>`).
- Clone the repository from GitHub.

  Use the following command:

  Linux

  ```
  $ git clone https://github.com/rticommunity/rticonnextdds-
  ↪getting-started.git
  ```

  macOS

  ```
  $ git clone https://github.com/rticommunity/rticonnextdds-
  ↪getting-started.git
  ```

  Windows

  ```
  > git clone https://github.com/rticommunity/rticonnextdds-
  ↪getting-started.git
  ```

  Check out the desired release. For example:

  Linux

  ```
  $ git checkout release/<version>
  ```

  macOS

```
$ git checkout release/<version>
```

Windows

```
> git checkout release/<version>
```

2.2.2 Set Up Environment Variables

The *RTI Code Generator* tool is located in `<installdir>/bin/rtiddsgen` (`<installdir>` refers to the installation directory for *Connext*).

To follow the instructions in the coming sections, add `<installdir>/bin` to your path.

1. Open a command prompt window, if you haven't already.
2. Add `<installdir>/bin/` to your path.

Linux

```
$ export PATH=${PATH}:<installdir>/bin
```

macOS

```
$ export PATH=${PATH}:<installdir>/bin
```

Windows

```
> set PATH=%PATH%;<installdir>\bin
```

When a directory name has a space, enclose the path in quotation marks. For example: `"C:\Program Files\rti_connext_dds-<version>\bin"`.

2.2.3 Run Code Generator

Inside the repository you have cloned, there is a directory named `2_hello_world`, which contains the **HelloWorld** type definition in a file named `hello_world.idl`.

1. Open `hello_world.idl` to see the definition for our **HelloWorld** type:

```
// Hello world!
struct HelloWorld {

    // String with maximum length of 256 characters
    string<256> msg;

};
```

This language-independent interface is written in IDL, the Interface Definition Language. IDL allows you to declare data types used for communication (we'll cover this more in *Data Types*). *Connext* includes a code generator that translates from this language-independent data type into code specific

for your language. The generated code serializes and deserializes your data into and out of a network format.

2. From a terminal or command prompt, run *rtiddsgen*, which runs the code generator on `hello_world.idl`:

   Linux

   ```
   $ cd 2_hello_world
   $ rtiddsgen -language c# -example net5 -d csharp hello_world.idl
   ```

   macOS

   ```
   $ cd 2_hello_world
   $ rtiddsgen -language c# -example net5 -d csharp hello_world.idl
   ```

   Windows

   ```
   > cd 2_hello_world
   > rtiddsgen -language c# -example net5 -d csharp -ppDisable hello_world.
   ↪idl
   ```

   `-ppDisable` disables the preprocessor. It is necessary for running *rtiddsgen* on a Windows system if the preprocessor is not in your path. You can only use `-ppDisable` if your IDL is simple, as it is here—otherwise you must add the preprocessor to your path. See Command-Line Arguments for rtiddsgen, in the RTI Code Generator User's Manual if you want more information.

   If you haven't configured your path as described in *Set Up Environment Variables*, run `<install dir>/bin/rtiddsgen`.

   `-example net5` generates an example .NET 5 publisher and subscriber application.

   `-d csharp` specifies the directory where the code will be generated.

3. Open the `csharp` directory to review the code.

Overview of Generated and Example Code

The code you just generated includes the files in Table 2.1, in the `csharp` directory. Some of the files are generated by *rtiddsgen*, some came from the GitHub repository you cloned.

Table 2.1: Generated and Example Code Files

Files	Description
• `hello_world.cs` • `hello_worldPlugin.cs`	The C# definition of your data type, and the code used to serialize and deserialize it (convert it to a format for the network). This is the type-specific code that will be used in your real application.
• `hello_worldPublisher.cs` • `hello_worldSubscriber.cs` • `hello_worldProgram.cs`	Example code you can read and modify. These files will compile as a .NET application that can run as a publisher or as a subscriber. These files were generated because you specified `-example net5`.
• `hello_world.csproj` • `NuGet.Config`	Used to build and run your application. `NuGet.Config` tells the .NET compiler where to find the `Rti.ConnextDds` NuGet package. These files were generated because you specified `-example net5`.
• `USER_QOS_PROFILES.xml`	Configuration file for Quality of Service (to be discussed more later). This file came from the repository you cloned.

2.2.4 Open/Modify Publishing Application

To browse the code with IntelliSense (auto-completion) and inline API documentation you can use **Visual Studio Code** (with the C# extension by Microsoft), **Visual Studio 2019**, or **Visual Studio for Mac**.

To browse the code in **Visual Studio Code**, open the `2_hello_world/csharp` folder:

```
$ code 2_hello_world/csharp
```

To use **Visual Studio** or **Visual Studio for Mac**, double-click on `hello_world.csproj` or open it from the IDE.

Once you've opened the source code in your IDE of choice:

1. Open `hello_worldPublisher.cs`

   This snippet shows how to create a *Topic* (with a name and data type) and create a *DataWriter* for that *Topic*:

   ```
   // A Topic has a name and a datatype.
   Topic<HelloWorld> topic = participant.CreateTopic<HelloWorld>(
   ```

```
    "Example HelloWorld");

// A Publisher allows an application to create one or more DataWriters
// Publisher QoS is configured in USER_QOS_PROFILES.xml
Publisher publisher = participant.CreatePublisher();

// This DataWriter will write data on Topic "Example HelloWorld"
// DataWriter QoS is configured in USER_QOS_PROFILES.xml
DataWriter<HelloWorld> writer = publisher.CreateDataWriter(topic);
```

2. Change the *Topic* name from "Example HelloWorld" to "HelloWorld Topic":

```
Topic<HelloWorld> topic = participant.CreateTopic<HelloWorld>(
    "HelloWorld Topic");
```

3. Modify the code to send the message "Hello world!" with a count.

   The following snippet shows how to write a HelloWorld update using the *DataWriter*'s write method.

   In the **for** loop in the snippet, add the highlighted line, just after the comment `// Modify the data to be written here`. This will set **sample.msg** to "Hello world!" with a count:

```
var sample = new HelloWorld();
for (int count = 0; count < sampleCount; count++)
{
    // Modify the data to be written here
    sample.msg = $"Hello World! {count}";

    Console.WriteLine($"Writing HelloWorld, count {count}");

    writer.Write(sample);

    Thread.Sleep(1000);
}
```

   Recall that your "HelloWorld Topic" describes your data. This Topic is associated with the data type **HelloWorld**, which is defined in the IDL file (see *Run Code Generator*). The data type **HelloWorld** contains a string field named **msg**. In this step, you have just added code to set a value for the **msg** field. Now, when the *DataWriter* writes data, the **msg** field in the data will contain the string "Hello world! 1", "Hello world! 2", etc.

Definition

A **sample** is a single update to a *Topic*, such as "Hello world! 1". Every time an application calls `Write()`, it is "writing a sample." Every time an application receives data, it is "receiving a sample."

Note that samples don't necessarily overwrite each other. For example, if you set up a Reliable Quality of Service (QoS) with a History **kind** of KeepAll, all samples will be saved and accumulate. You can find more details in *Basic QoS*.

2.2.5 Open/Modify Subscribing Application

The subscriber application also creates a *Topic*.

1. Open `hello_worldSubscriber.cs`:

   This snippet shows how to create a Topic (with a name and data type) and create a *DataReader* for that Topic:

```
// A Topic has a name and a datatype. Create a Topic named
// "HelloWorld Topic" with type HelloWorld
Topic<HelloWorld> topic = participant.CreateTopic<HelloWorld>(
    "Example HelloWorld");

// A Subscriber allows an application to create one or more DataReaders
// Subscriber QoS is configured in USER_QOS_PROFILES.xml
Subscriber subscriber = participant.CreateSubscriber();

// This DataReader reads data on Topic "Example HelloWorld".
// DataReader QoS is configured in USER_QOS_PROFILES.xml
DataReader<HelloWorld> reader = subscriber.CreateDataReader(topic);
```

2. Change the *Topic* name from "Example HelloWorld" to "HelloWorld Topic", just as you did in the publishing application.

Note: The *Topic* names must match between the publishing and subscribing applications for the *DataWriter* and *DataReader* to communicate.

```
Topic<HelloWorld> topic = participant.CreateTopic<HelloWorld>(
    "HelloWorld Topic");
```

Details of Receiving Data

You don't need to make any changes here, but look at the `HelloWorldSubscriber.cs` code to see how it receives data. The *DataReader* is being notified of new data using an object called a WaitSet:

```
// Create a WaitSet and attach the StatusCondition
var waitset = new WaitSet();
waitset.AttachCondition(statusCondition);
while (samplesRead < sampleCount)
{
    // Dispatch will call the handlers associated to the WaitSet
    // conditions when they activate
    Console.WriteLine("HelloWorld subscriber sleeping for 4 sec...");
    waitset.Dispatch(Duration.FromSeconds(4));
}
```

This WaitSet object is a way for the application to sleep until some "condition" becomes true. When the application calls `waitset.Dispatch(Duration.FromSeconds(4))`, it will sleep for up to the duration time (4 seconds in this example), unless it is woken up due to a condition becoming true.

There are multiple types of conditions that you can attach to a WaitSet, but this example shows a StatusCondition. Here is the code for the StatusCondition:

```
// Obtain the DataReader's Status Condition
StatusCondition statusCondition = reader.StatusCondition;

// Enable the 'data available' status.
statusCondition.EnabledStatuses = StatusMask.DataAvailable;

// Associate an event handler with the status condition.
// This will run when the condition is triggered, in the context of
// the dispatch call (see below)
int samplesRead = 0;
statusCondition.Triggered += _ => samplesRead += ProcessData(reader);
```

In this example, we are saying that we are interested in waking up when the "data available" status becomes true. This means that when data arrives:

- The reader's `DataAvailable` status becomes true
- The application is woken up from the `Dispatch` call
- The `Triggered` event handler is called

A closer look at `ProcessData`, called by the event handler:

```
private static int ProcessData(DataReader<HelloWorld> reader)
{
    // Take all samples. Samples are loaned to application, loan is
    // returned when the samples variable is Disposed.
    int samplesRead = 0;
    using (var samples = reader.Take())
    {
        foreach (var sample in samples)
        {
            if (sample.Info.ValidData)
            {
                Console.WriteLine(sample.Data);
                samplesRead++;
            }
        }
    }

    return samplesRead;
}
```

This code calls `reader.Take()`, which removes any available samples out of the *DataReader* queue, and returns them in a collection. If data is arriving quickly, it is likely this collection will contain multiple samples. In this example, the sample's data is being printed to the screen with `Console.WriteLine`.

2.2.6 Compile Your Changes

Now that you have made changes to both the publisher and subscriber code, compile the code with your modifications. From `2_hello_world/csharp` run:

```
$ dotnet build
```

You can also build the code from **Visual Studio Code**, **Visual Studio**, and **Visual Studio for Mac**. This guide will use the `dotnet` command-line interface to build and run the applications.

2.2.7 Run the Applications

1. From within the `2_hello_world/csharp` directory, enter the following:

   ```
   $ dotnet run -- --pub
   ```

   Note: You must be *in* the `2_hello_world/csharp` directory when you run the previous command. Or if you opened the project in Visual Studio, make sure the current working directory is set to the full path to `2_hello_world/csharp`. The examples use Quality of Service (QoS) information from the file `USER_QOS_PROFILES.xml` in the `2_hello_world/csharp` directory. We'll talk more about QoS in a later module.

   You should see this in the window for the publisher:

   ```
   Writing HelloWorld, count 1
   Writing HelloWorld, count 2
   Writing HelloWorld, count 3
   Writing HelloWorld, count 4
   ...
   ```

2. Open another command prompt window, and from within the `2_hello_world/csharp` directory, enter the following:

   ```
   $ dotnet run -- --sub
   ```

   You should see this in the window for the subscriber:

   ```
   msg: "Hello world! 1"
   Wait timed out after 4 seconds.

   msg: "Hello world! 2"
   Wait timed out after 4 seconds.

   msg: "Hello world! 3"
   Wait timed out after 4 seconds.

   msg: "Hello world! 4"
   ...
   ```

Note: Since the `waitTimeout` for both the publisher and subscriber is up to 4 seconds, you may or may not get the `Wait timed out after 4 seconds` line between the samples, depending on the timing between the two applications.

The `msg: "Hello world! <count>"` line is the data being sent by the *DataWriter*. If the *DataWriter* weren't communicating with the *DataReader*, you would just see the `Wait timed out after 4 seconds` lines and not the "msg" lines. (The subscribing application prints the "timed out" lines after the WaitSet times out while waiting for data, then it prints the "msg:" lines when it receives data from the *DataWriter*.)

Note: If you don't get the results described here, see *Troubleshooting*.

2.2.8 Taking It Further

Under the hood, the publishing and subscribing applications are doing a lot of work:

Before communication starts, the *DataWriter* and *DataReader* discover each other and check that they have the same *Topic* name, compatible data types, and compatible QoS. (We will talk more about discovery in a later module). After discovery, the *DataWriter* sends data directly to the *DataReader*, with no message broker required.

When you run the applications on the same machine, by default they communicate over shared memory. If you run one on another machine, they communicate over the network using UDP.

Start up Multiple Publishing or Subscribing Applications

Try starting up multiple publishing or subscribing applications, and you will see that they will also send or receive data. (Remember to run from the `2_hello_world/csharp` directory.)

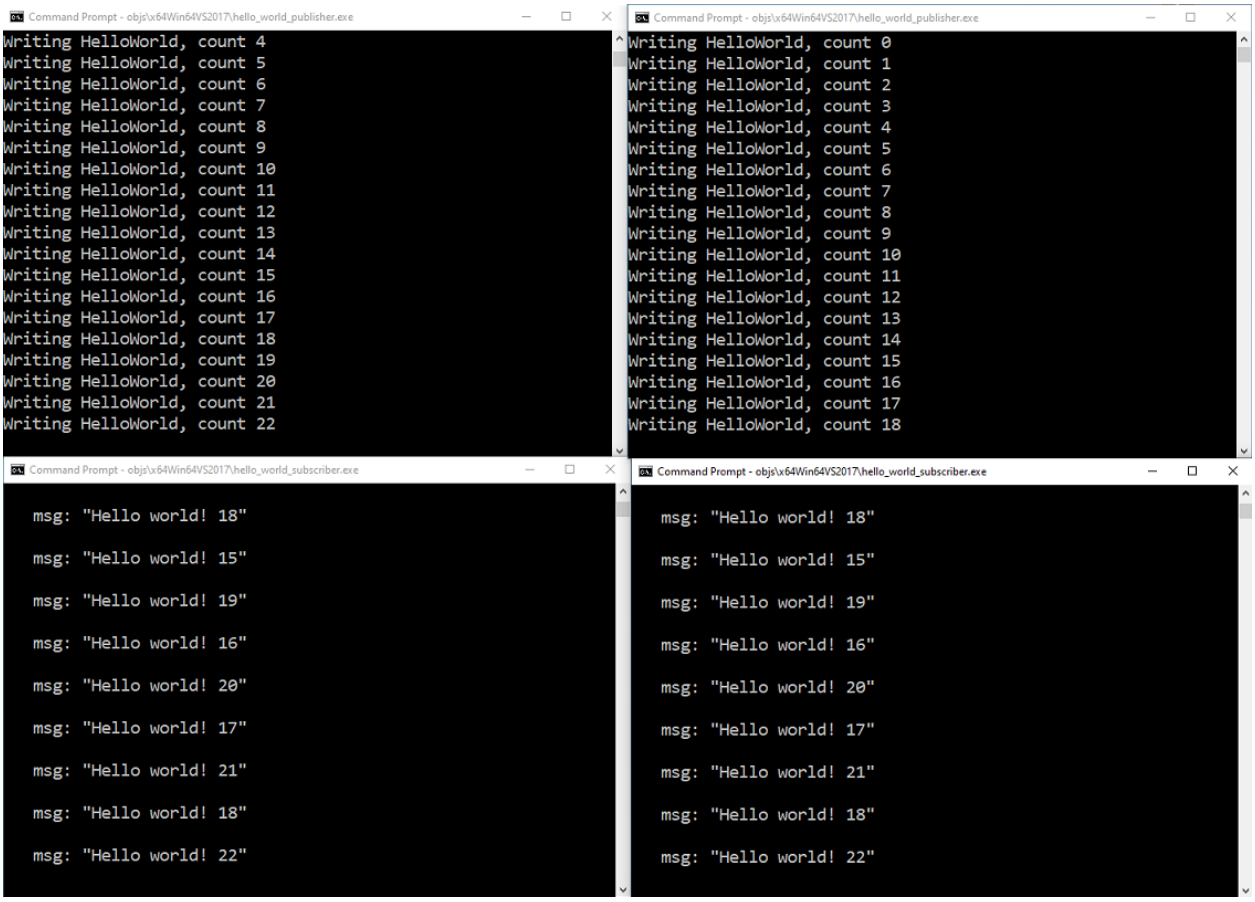


Figure 2.2: Two applications publishing, and two subscribing, to the same *Topic*. Notice that each subscriber is receiving data from both publishers.

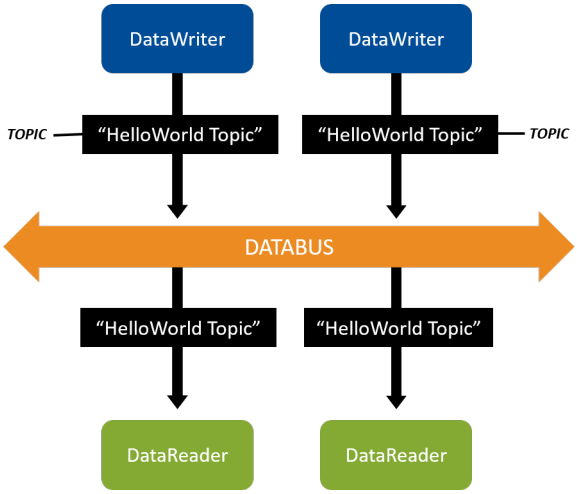


Figure 2.3: Two applications publishing, and two subscribing, to the same *Topic*.

Publish/Subscribe across Multiple Machines

To publish/subscribe between two machines:

1. Clone the repository on both machines. See *Clone Repository*.
2. In the `2_hello_world` directory on one machine, modify and run the publishing application as described in *Open/Modify Publishing Application*. See also *Compile Your Changes* and *Run the Applications*.
3. In the `2_hello_world` directory on the other machine, modify and run the subscribing application as described in *Open/Modify Subscribing Application*. See also *Compile Your Changes* and *Run the Applications*.

Note: If you are running both applications and they aren't communicating (you don't see the "msg:" lines shown at the end of *Run the Applications*), see *Troubleshooting*.

Create Multiple DataWriters, DataReaders, and Topics in a Single Application

So far, you have created two applications: one that uses a *DataReader* to subscribe to "HelloWorld Topic" and one that uses a *DataWriter* to publish "HelloWorld Topic." You have seen that these applications automatically discover each other and communicate, and that you can run multiple copies of them.

In *Data Types*, you'll add a second *DataWriter* to an application, and in *Keys and Instances*, you'll create an application that contains multiple *DataWriters*, *DataReaders*, and *Topics*.

Tip: Even though these initial example applications each do only one thing, typical real-world applications contain a combination of *DataWriters* and *DataReaders* for multiple *Topics* in a single application. We will illustrate this in later modules.

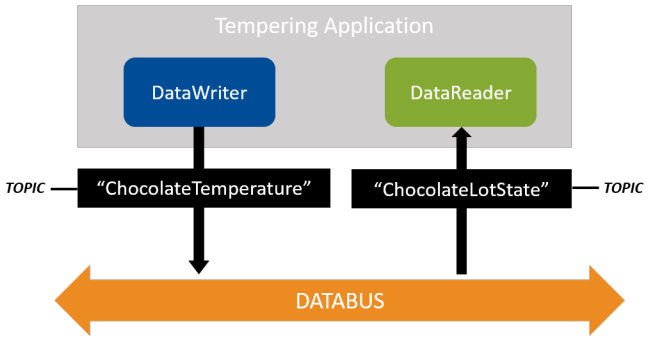


Figure 2.4: Sneak Preview: In a later module, we will create a tempering application that will write a "ChocolateTemperature" *Topic* and read a "ChocolateLotState" *Topic*. An application can write and read multiple *Topics* by creating more *DataWriters* and *DataReaders*.

2.3 Troubleshooting

2.3.1 Why aren't my applications communicating?

If you are running both applications and they aren't communicating (you don't see the `msg: Hello world` lines shown at the end of *Run the Applications*), here are some things to check:

- Did you change the *Topic* name in both applications before compiling? They should match.
- Are you running both applications from the `2_hello_world/csharp` directory, so they load the same `USER_QOS_PROFILES.xml` file?

  You must be *in* the `2_hello_world/csharp` directory when you type `dotnet run`.
- If you are running on multiple machines, does your network support multicast? If it does not, see *How do I set my discovery peers?* below, for how to specify the addresses of the remote machines your application plans to communicate with.
- Check to see if one or both machines use a firewall. You may need to disable your firewall or configure it to allow multicast traffic for communication to be established.

See also *Discovery*.

2.3.2 How do I set my discovery peers?

If you are running *Connext* on multiple machines and your network does not support multicast, specify the address(es) of the remote machine(s) you want to communicate with:

1. In the `2_hello_world\csharp` directory, find USER_QOS_PROFILES.xml.
2. Identify the following lines in USER_QOS_PROFILES.xml:

```
<domain_participant_qos>
    <!--
    The participant name, if it is set, will be displayed in the
    RTI tools, making it easier for you to tell one
    application from another when you're debugging.
    -->
    <participant_name>
        <name>HelloWorldParticipant</name>
    </participant_name>

</domain_participant_qos>
```

3. Add the `<discovery>` section as follows, and replace the IP address with the IP address or hostname of the other machine you want to communicate with:

```
<domain_participant_qos>
    <!--
    The participant name, if it is set, will be displayed in the
    RTI tools, making it easier for you to tell one
    application from another when you're debugging.
```

```
    -->
    <participant_name>
        <name>HelloWorldParticipant</name>
    </participant_name>
    <discovery>
        <initial_peers>
            <!-- Add an element for each machine you want to communicate
↪with -->
            <element>192.168.1.14</element>
        </initial_peers>
    </discovery>

</domain_participant_qos>
```

If you want more information about discovery peers, see Configuring the Peers List Used in Discovery, in the RTI Connext Core Libraries User's Manual.

2.3.3 Why does the DataReader miss the first samples?

Discovery is not an instantaneous event. It takes some time for the discovery process between applications to complete. The *DataWriter* and *DataReader* must discover each other before they can start communicating. Therefore, if you send data immediately after creating the *Connext* entities, *DataReaders* will not receive the first few samples because they are still in-process of discovering the *DataWriters* and vice versa. This is true even when the *DataWriters* and *DataReaders* are reliable, because the Reliability QoS on its own does not guarantee delivery to *DataReaders* that have not been discovered yet.

You can overcome this behavior with the Durability QoS Policy, which can be set to deliver historical samples (that *DataWriters* already sent) to late-joining *DataReaders*. The *DataReaders* will then receive the first samples they originally missed. We'll talk about this more in *Basic QoS*.

2.3.4 Why do I get a "No source for License information" error?

If you get the following error, it may be because you're importing the `Rti.ConnextDds` NuGet package from nuget.org, which requires a license.

```
RTI Connext DDS No source for License information
Please contact support@rti.com with any questions or comments
```

If you intend to use the nuget.org packages, make sure a license file is available when you run your application (see *Setting Up a License*).

If you have installed the C# support package (*Installing Connext*), which doesn't require a license file to run applications, make sure there is a `NuGet.Config` file in the same directory where you run `dotnet build` or `dotnet run`.

It should look like this:

```
<configuration>
    <packageSources>
        <add key="local" value="<installdir>/lib/dotnet" />
```

```
    </packageSources>
</configuration>
```

This file is generated by **rtiddsgen** when you specify the option `-example` or `-create makefiles`. `<installdir>` refers to the installation directory for *Connext*.

Alternatively, you can add this package source globally with the following command:

```
$ dotnet nuget add source <installdir>/lib/dotnet --name RTI
```

After this, the `Rti.ConnextDds` package will be located locally without the need of `NuGet.Config` files.

Note: You can remove it at any time: `dotnet nuget remove source RTI`.

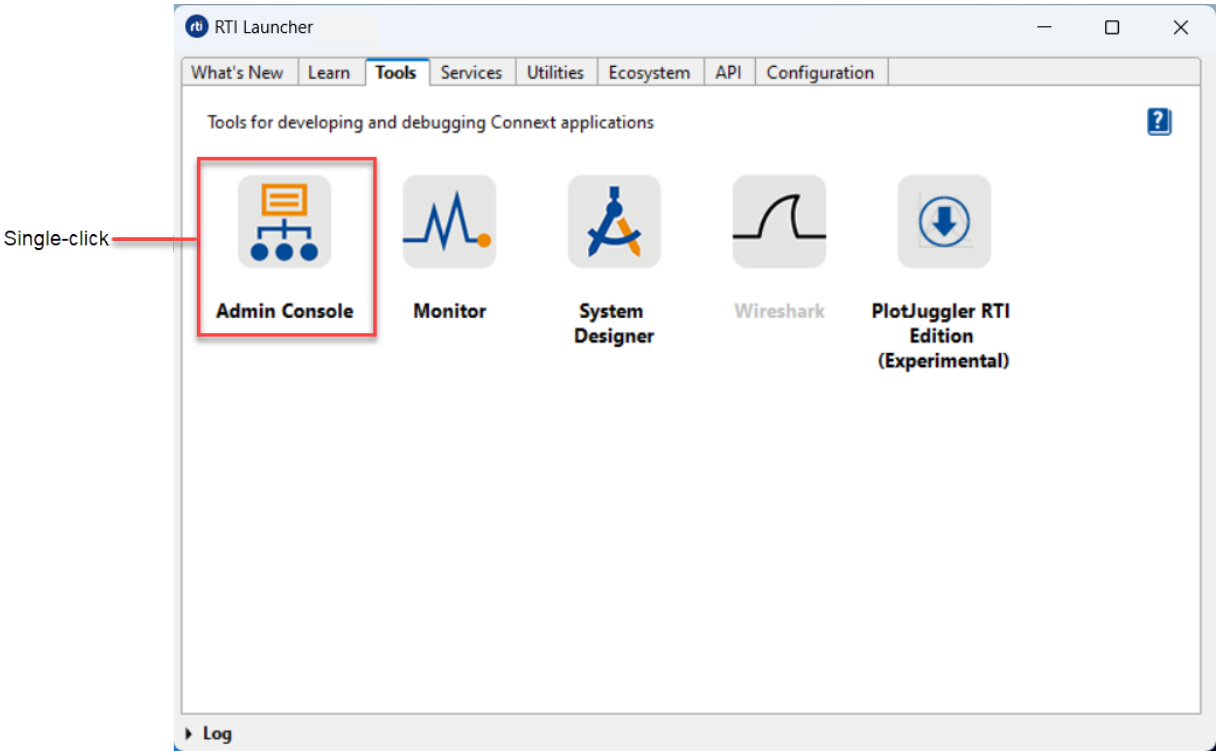
2.4 Hands-On 2: Viewing Your Data

Now that you've created applications that are publishing data, you can visualize that data using the *Admin Console* tool.

Note: The applications from Hands-On 1 above should be running while you perform the *Admin Console* steps below. See *Run the Applications*.

2.4.1 Open Admin Console

Start by opening *Admin Console* from *RTI Launcher*:



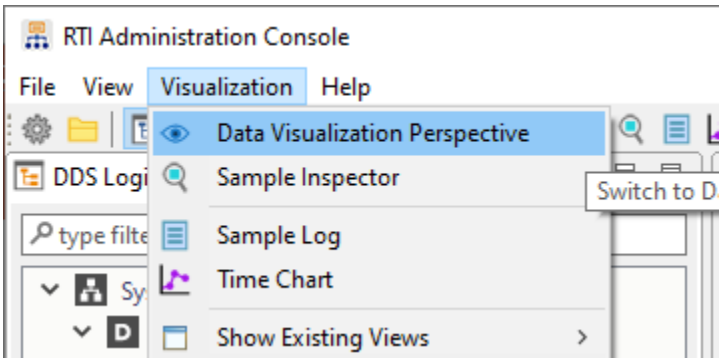
The following sections guide you through *Admin Console* for the purposes of this exercise, but you can find more information in the Admin Console User's Manual.

2.4.2 Choose Automatically Join

You may be prompted to automatically or manually join domains when you first open *Admin Console*. For the purposes of this exercise, choose to automatically join (the default). We'll discuss domains in a later module.

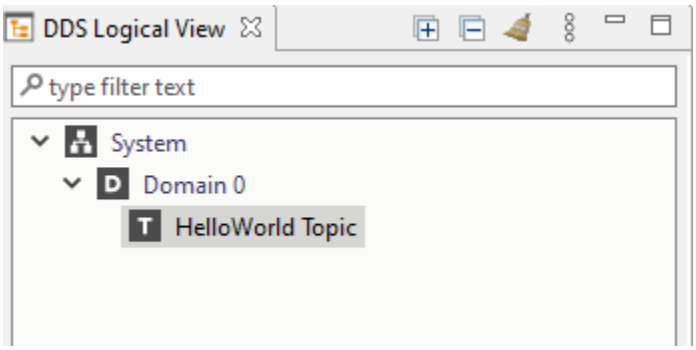
2.4.3 Switch to Data Visualization Perspective

Select the Data Visualization Perspective entry under the Visualization menu. (You may need to close a Welcome tab first.) If you can't select the Data Visualization Perspective menu item, you may already be in that mode.

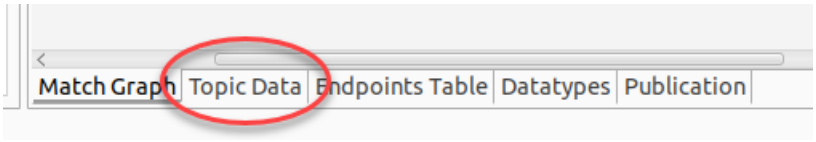


2.4.4 Open Topic Data Tab

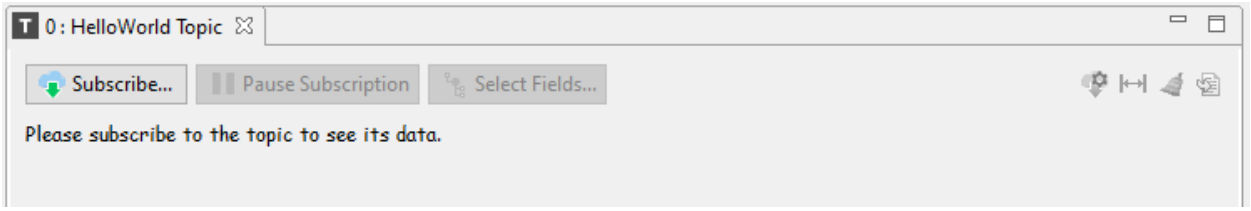
From the DDS Logical View, click on the "HelloWorld Topic" to open the Topic View.



Select the Topic Data tab at the bottom of the window.



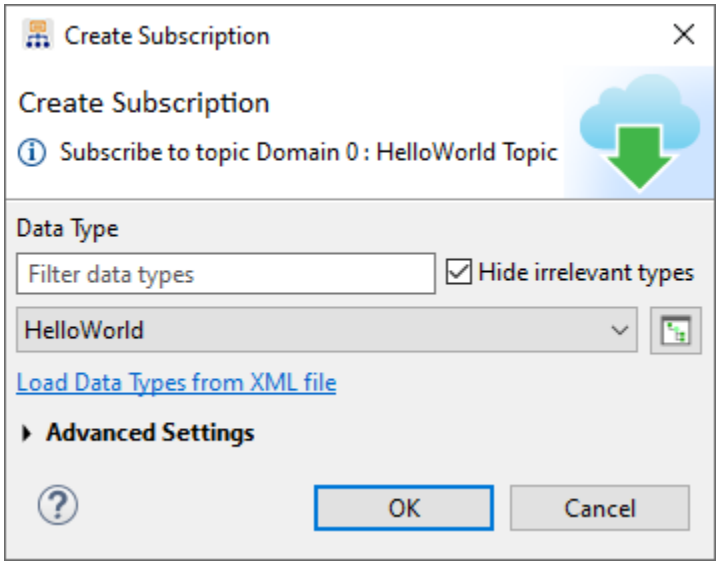
Click Subscribe:



2.4.5 Subscribe to "HelloWorld Topic"

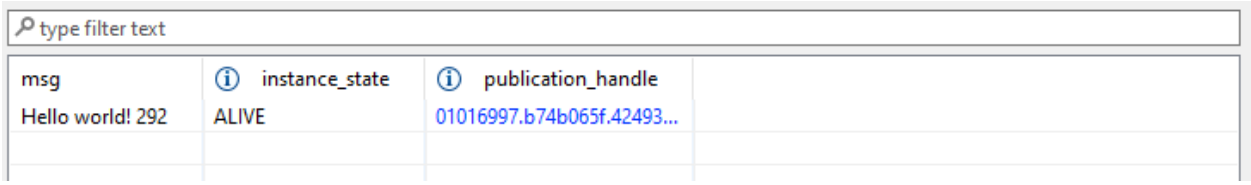
The *Admin Console* tool itself is a DDS application, so it must also create *DataReaders* to subscribe to the "HelloWorld Topic." (See the *Admin Console* online Help for more information about this.) When you subscribe to the "HelloWorld Topic" in *Admin Console*, *Admin Console* will create a *DataReader* for this *Topic*.

Click on the **Subscribe…** button or right-click on the *Topic* in the DDS Logical View and select Subscribe. This will open the **Create Subscription** dialog seen below. Click OK to subscribe to the *Topic*.



2.4.6 Use Topic Data Tab

After the *Topic* is subscribed to, you will see your "Hello world" message.

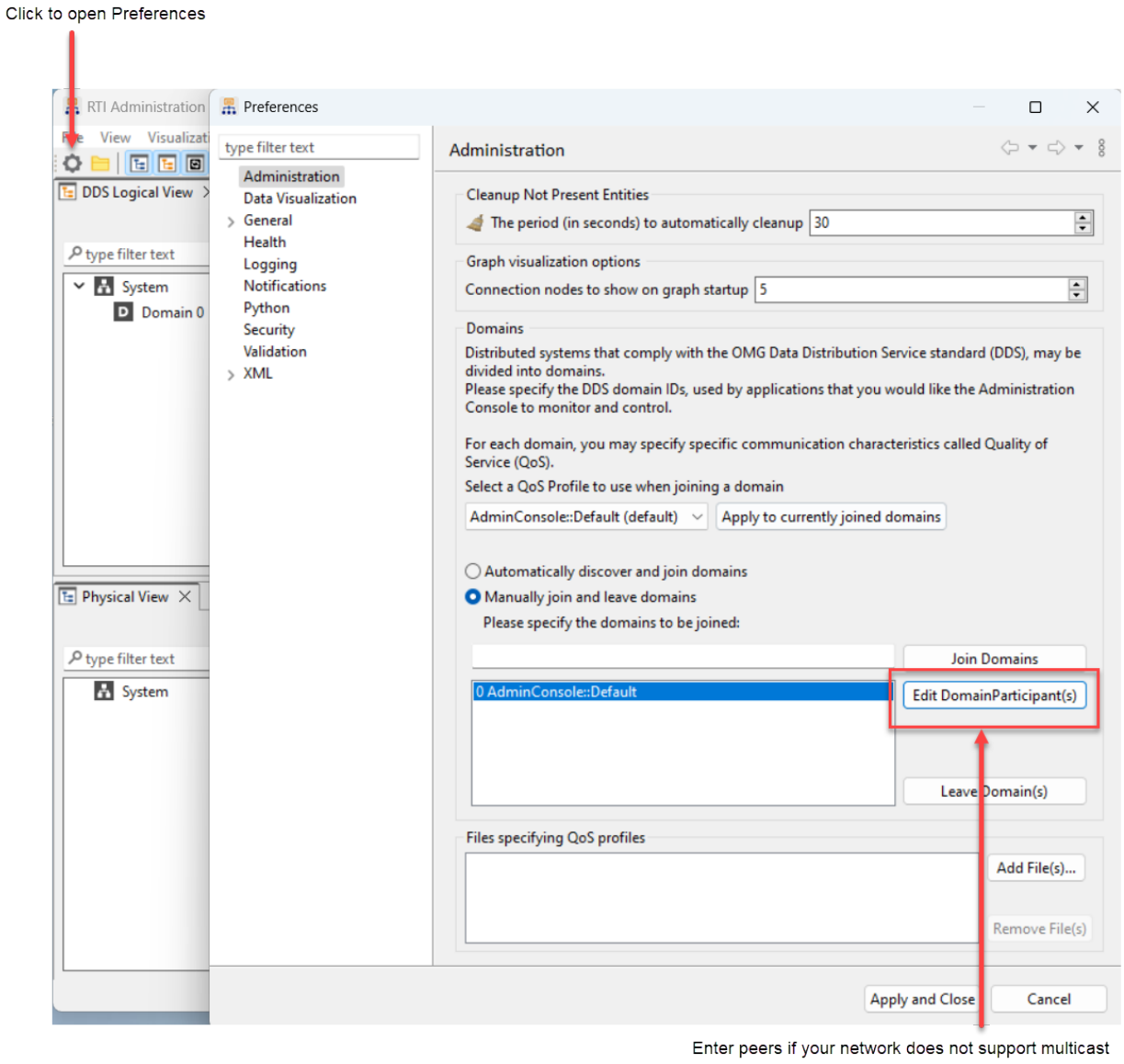


In the *Admin Console* tool, you're also able to inspect *DataWriters* and *DataReaders*, and even chart your live data.

2.4.7 Use Admin Console across Machines

Admin Console is like any other *Connext* application. It discovers what *DataWriters* and *DataReaders* are on the network, whether they're on the same machine as *Admin Console* or on different machines.

If you have trouble viewing data in *Admin Console* across machines, the same troubleshooting tips apply to *Admin Console* as apply to *DataWriters* and *DataReaders* in *How do I set my discovery peers?*. If your network supports multicast, you'll see data on the other machines in *Admin Console*; otherwise, specify the IP addresses of the remote machines you want *Admin Console* to communicate with.



2.5 Next Steps

Congratulations! You've written your first DDS application, which publishes HelloWorld data. In this exercise, you've experienced a quick overview of the development process from defining a data type and using the code generator, to building an example application and using *Connext Professional* tools to see that data is being published. We'll continue to build on these skills and to use these tools in more depth in subsequent exercises.

The next module takes a deeper dive into data types, including a little more information about how to define common types. We will be starting to work with an example that's more complex than a "hello world"—we're going to be using a chocolate factory example to showcase DDS concepts. See *Data Types*.

Chapter 3

Data Types

Prerequisites	• *Publish/Subscribe*, including: – *DataWriters*, *DataReaders*, and *Topics* – Using the code generator • Repository cloned from GitHub here
Time to complete	30 minutes
Concepts covered in this module	• Typed data • Interface Definition language (IDL) • Introduction to data flows • Streaming data

The first step in creating a DDS application is to define the interface between applications. In DDS, the interface is the data itself rather than the bits and bytes that make up a protocol. In DDS, the *Topic* written by a *DataWriter* and read by a *DataReader* is associated with one data type. For example, in *Publish/Subscribe*, the data type was named **HelloWorld** and contained a single string. The "HelloWorld Topic" was associated with the **HelloWorld** data type.

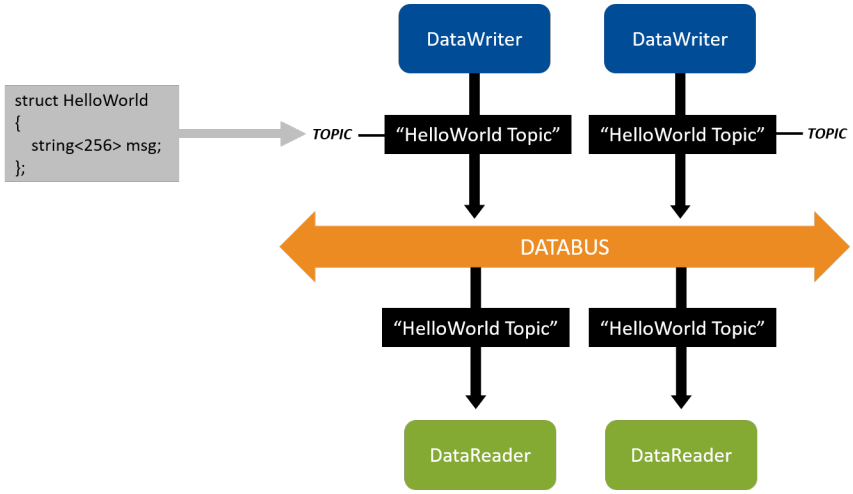


Figure 3.1: In *Publish/Subscribe*, you started two applications that published the "HelloWorld Topic" and two that subscribed to the "HelloWorld Topic." The "HelloWorld Topic" uses the **HelloWorld** data type.

The same data type can be reused across multiple *Topics*. For example, a data type named **Temperature** might be associated with the *Topics* "ChocolateTemperature" and "FactoryTemperature." Although "ChocolateTemperature" and "FactoryTemperature" measure two different things, they are both temperature data types. Thus, the data type can be reused for both of these *Topics*.

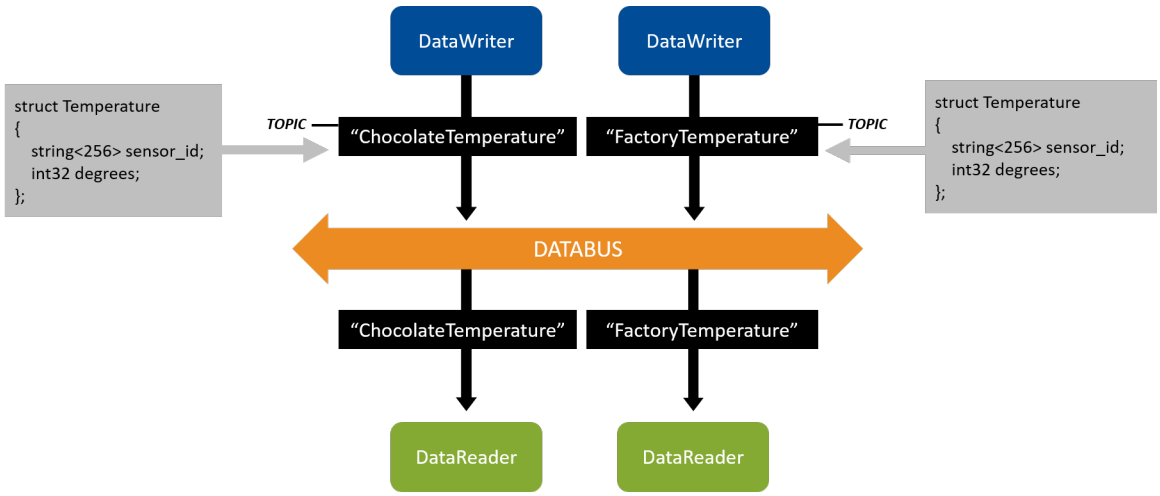


Figure 3.2: Multiple Topics can share the same data type.

In *Publish/Subscribe*, you opened an IDL (.idl) file that contained the **HelloWorld** data type. IDL is the "Interface Definition Language," defined by the Object Management Group. It allows you to define data types in a way that is not specific to the language your applications are written in, enabling applications that are written in C, C++, Java, etc., to communicate.

3.1 Common IDL Types

A few common IDL types are listed in the table below to help you get started. Many more are supported by *Connext*. More information on working with IDL data types can be found in Creating User Data Types with IDL, in the RTI Connext DDS Core Libraries User's Manual.

Table 3.1: Some IDL Types

Type	Parameters	Description
`boolean`		Boolean value (true or false)
`char`		8-bit quantity
`double`		64-bit double precision floating-point number
`enum`		Enumerated value
`float`		32-bit single precision floating-point number
`int16`		16-bit numeric type
`int32`		32-bit numeric type
`int64`		64-bit numeric type
`octet`		8-bit type, used for storing binary data that should not be serialized/deserialized by the middleware. Usually a sequence.
`sequence`	`<type, max length>`	Sequence of type, with a maximum of max length elements. Max length is optional.
`string`	`<max length>`	String that may contain data from length 0 to max length. Max length is optional.
`struct`		Structure containing other types

3.2 Introduction to Data Flows

To design your data types, decide how many you need and what each one will be used for. Consider the relationships of all your application data. Some questions to consider:

- Is this data produced and consumed in the same places?
- Can this data logically be described by the same *Topic*?
- Does this data have the same data flow characteristics?

To answer the third question, it's important to discuss what data flow characteristics are. Some of these characteristics include:

- How frequently data is sent
- Whether data is sent periodically or asynchronously
- Whether it is okay to miss an update

There are additional data flow characteristics that we will cover later when we talk about Quality of Service (in *Basic QoS*). An example of a common data flow pattern is "Streaming Sensor Data."

Tip: Streaming sensor data:

- Usually sent rapidly
- Usually sent periodically
- When data is lost over the network, it is more important to get the next update rather than wait for the lost update

Other data flows include "State Data" and "Event and Alarm Data."

All these data flows will be discussed in more detail in *Basic QoS*.

3.3 Hands-On 1: Streaming Data

This Hands-On will use an example similar to the "Hello World" example in *Publish/Subscribe*, but with a few modifications. (Instructions in the following exercises are a little less detailed because we assume you have already performed the exercises in *Publish/Subscribe*.)

3.3.1 Run Code Generator

The code for this example is in the directory `3_streaming_data`. (See *Clone Repository*.)

1. In `3_streaming_data`, open `chocolate_factory.idl` to see the definition for our temperature type. (There is also a ChocolateLotState type that we will use later).

   In the IDL file, we've changed the data type from a message string to a **Temperature** that includes both a sensor ID and degrees:

   ```
   // Temperature data type
   struct Temperature {
       // ID of the sensor sending the temperature
       string<256> sensor_id;

       // Degrees in Celsius
       int32 degrees;
   };
   ```

2. In the directory `csharp/ChocolateFactoryPublisher`, open the `ChocolateFactoryPublisher.cs` file to see that we've changed the *Topic* to "ChocolateTemperature":

   ```
   // A Topic has a name and a datatype. Create a Topic named
   // "ChocolateTemperature" with type Temperature
   Topic<Temperature> topic = participant.CreateTopic<Temperature>(
   ↪"ChocolateTemperature");
   ```

   We have modified the application so that you can specify a "sensor ID" at the command-line when running your application, by passing `-i <some sensor name>`. In that file, we've also modified

the data being sent so that it includes both that sensor ID and a temperature ranging between 30 and 32 degrees:

```
// Modify the data to be written here
sample.sensor_id = sensorId;
sample.degrees = rand.Next(30, 33);  // Random number between 30 and 32
```

3. *Run Code Generator* to generate C# code for the types defined in `chocolate_factory.idl`. In this exercise we already have application code, so we will only generate the type files and project files.

   Linux

```
$ cd 3_streaming_data
$ rtiddsgen -language c# -platform net5 -create typefiles -create␣
↪makefiles -d csharp chocolate_factory.idl
```

   macOS

```
$ cd 3_streaming_data
$ rtiddsgen -language c# -platform net5 -create typefiles -create␣
↪makefiles -d csharp chocolate_factory.idl
```

   Windows

```
> cd 3_streaming_data
> rtiddsgen -language c# -platform net5 -create typefiles -create␣
↪makefiles -d csharp -ppDisable chocolate_factory.idl
```

   `-ppDisable` disables the preprocessor. It is necessary for running *rtiddsgen* on a Windows system if the preprocessor is not in your path. You can only use `-ppDisable` if your IDL is simple, as it is here—otherwise you must add the preprocessor to your path. See Command-Line Arguments for rtiddsgen, in the RTI Connext DDS Code Generator User's Manual if you want more information.

3.3.2 Modify for Streaming Data

In this step, you will modify your applications to support one of the common design patterns that most applications need: Streaming Data. This pattern is characterized by:

- Data that is sent frequently and periodically.
- No need for reliability: if a sample is lost on the network, it is better to drop it than possibly delay the next one.

This pattern is usually seen with sensor data.

Definition

A **sample** is a single data update sent or received over DDS. For example: temperature = 32.

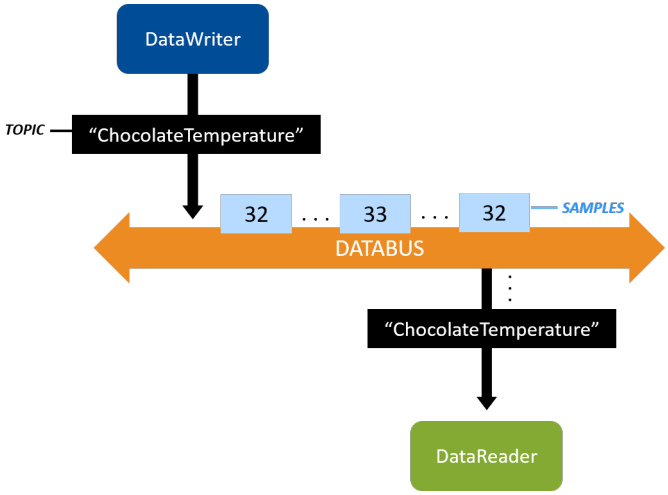


Figure 3.3: DataWriter sending temperature samples

To make your application illustrate streaming data:

1. Change `ChocolateFactoryPublisher.cs` from 4 seconds to 100 milliseconds, as shown below:

```
writer.Write(sample);

// Exercise #1.1: Change this to sleep 100 ms in between writing␣
↪temperatures
Thread.Sleep(100);
```

   (If you're using Windows and need help on modifying this file in your IDE, see the `README_<architecture>.txt` file generated with the code, in the `3_streaming_data/<language>` directory.)

2. Open the `USER_QOS_PROFILES.xml` file, in the same directory that contains the `ChocolateFactoryPublisher.cs` and `ChocolateFactorySubscriber.cs` files. We will cover Quality of Service (QoS) in greater depth in a later module, but for now we will use this file to change our *DataWriter* and *DataReader* to use QoS appropriate for streaming data. Do this by changing the `base_name` attribute from `BuiltinQosLib::Generic.StrictReliable` to `BuiltinQosLib::Pattern.Streaming`:

```
<!--
  QoS profile used to configure reliable communication between the
  DataWriter and DataReader created in the example code.

  base_name:
  Communication is reliable because this profile inherits from
  the built-in profile "BuiltinQosLib::Generic.StrictReliable"

  is_default_qos:
  These QoS profiles will be used as the default, as long as this
  file is in the working directory when running the example.
```

```
-->
<!-- Exercise #1.2: Use Streaming profile -->
<qos_profile name="ChocolateTemperatureProfile"
             base_name="BuiltinQosLib::Pattern.Streaming"
             is_default_qos="true">
```

Tip: This XML file is loaded from your working directory when you run your applications—this is why we specify that you run your applications from the `3_streaming_data/<language>` directory. Notice that the profile contains the attribute `is_default_qos`—this means that this profile will be used by default by the *DataWriter* and *DataReader*, as long as it is in your working directory. Later when we talk about QoS, we will show you how to specify a particular QoS profile instead of loading the default.

This modification to the QoS XML file will change the way *Connext* delivers your data from being reliable to "best effort." We will cover QoS in more depth in a later module.

3.3.3 Run the Applications

1. From within the `3_streaming_data/csharp` directory, enter the following command, optionally specifying your own sensor ID to send with the data, such as "MySensor1":

```
$ dotnet run -- --pub --sensor-id <some string>
```

Note: If you see a build or license error, see *Troubleshooting*.

You should run from the `3_streaming_data/csharp` directory because the examples use Quality of Service (QoS) information from the file `USER_QOS_PROFILES.xml` in that directory. We'll talk more about QoS in a later module.

Note also that the arguments to the application are specified after `--`.

You can also build and run from Visual Studio and Visual Studio Code as outlined in the previous section (*Compile Your Changes*).

---

2. Open another command prompt window, and from within the `3_streaming_data/csharp` directory, enter the following command:

```
$ dotnet run -- --sub
```

After modifying the publishing and subscribing applications as described above, and running both applications from the `3_streaming_data/csharp` directory where you generated code, you should see data rapidly arriving:

```
ChocolateTemperature subscriber sleeping for 4 sec...
[sensor_id: MySensor1, degrees: 32]
ChocolateTemperature subscriber sleeping for 4 sec...
[sensor_id: MySensor1, degrees: 32]
```

```
ChocolateTemperature subscriber sleeping for 4 sec...
[sensor_id: MySensor1, degrees: 30]
ChocolateTemperature subscriber sleeping for 4 sec...
[sensor_id: MySensor1, degrees: 31]
```

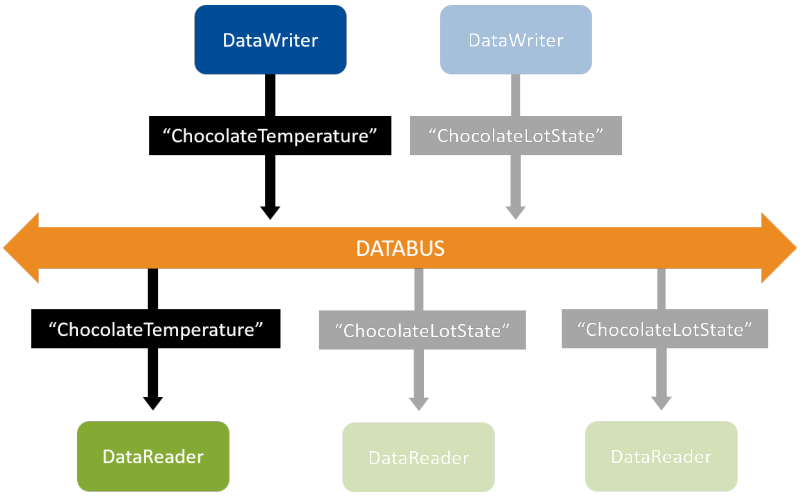


Figure 3.4: In this exercise, a *DataWriter* of the "ChocolateTemperature" *Topic* communicates with a *DataReader* of the "ChocolateTemperature" *Topic*. In the next Hands-On, you will add a "ChocolateLotState" Topic.

Congratulations! You now have streaming Temperature data.

3.4 Publishers, Subscribers, and DomainParticipants

Before we go any farther, it's important that we define a few more objects that you will see in your DDS applications. You may have noticed some of these objects already in the code, and you'll be using one of them in the next Hands-On. These objects are: *Publishers*, *Subscribers*, and *DomainParticipants*. Most of the time in these hands-on exercises we will ignore these, and focus on *DataWriters*, *DataReaders*, and *Topics*. But it's important to know that these other objects exist in every application.

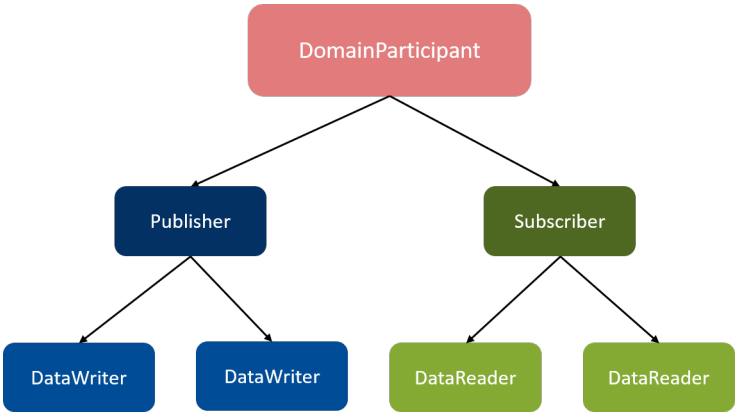


Figure 3.5: *DomainParticipants* create and manage *Publishers* and *Subscribers*. *Publishers* create and manage *DataWriters*. *Subscribers* create and manage *DataReaders*. *DataWriters* and *DataReaders* send and receive your data.

Definition

- A *DomainParticipant* object in *Connext* is used to create and manage one or more *Publishers* and *Subscribers*. The *DomainParticipant* is a container for most other objects, and is responsible for the discovery process. In most applications, you will have only one *DomainParticipant*, even if you have many *DataWriters* and *DataReaders*.
- A *Publisher* object in *Connext* is used to create and manage one or more *DataWriters*. A *Subscriber* object is used to create and manage one or more *DataReaders*. For more information, see Publishers, in the RTI Connext Core Libraries User's Manual and Subscribers, in the RTI Connext Core Libraries User's Manual.

We will be using the *Publisher* object in your **temperature_publisher** application to create a new *DataWriter* in the next Hands-On section.

We will see *DomainParticipants* again when we talk about QoS, and then when we talk about discovery and domains. Since they are used to create nearly every other DDS object in your system, they're one of the first objects you create when creating a DDS application. *DomainParticipants* also create *Topics*, which get used by your *DataWriters* and *DataReaders*. You'll see that when you add a second *Topic* in the next Hands-On.

Note: It's a common beginner's mistake to create one *DomainParticipant* per *DataWriter* or *DataReader*. As you can see, it's not necessary. You typically create one *DomainParticipant* per application. It's also a bad idea to use more than you need, because *DomainParticipants* use significant resources such as threads, and they use network bandwidth for discovery. We'll talk more about *DomainParticipants* in a later module on discovery.

3.5 Hands-On 2: Add a Second DataWriter

Now that you have created your first streaming data, we will add another *DataWriter* to the **chocolate_factory_publisher** application. This will give you an idea how to add a new *DataWriter* or *DataReader*, which will be useful because the code in the next module will have more-complex applications with multiple *DataReaders* and *DataWriters*.

3.5.1 Add the New DataWriter

Every *DataWriter* needs to write on a *Topic*, and this new *DataWriter* will use a different *Topic* and data type than the temperature *DataWriter*. This new *DataWriter* will write the *Topic* "ChocolateLotState" with the data type `ChocolateLotState` that is defined in the IDL file. We will use this new "ChocolateLotState" *Topic* again in the next module.

1. Stop running both of the applications from the previous Hands-On if you haven't already.
2. Add a new *Topic*.

   Inside of `ChocolateFactoryPublisher.cs` you should see this comment:

   ```
   // Exercise #2.1: Add new Topic
   ```

   Add the following code after the comment to create the new *Topic*:

   ```
   Topic<ChocolateLotState> lotStateTopic = participant.CreateTopic
   ↪<ChocolateLotState>(
       "ChocolateLotState");
   ```

3. Now, create the new *DataWriter* using that *Topic*. Look for this comment:

   ```
   // Exercise #2.2: Add new DataWriter and data sample
   ```

   Right after the comment, add the new *DataWriter* that writes on the new *Topic*, as well as the sample to write:

   ```
   DataWriter<ChocolateLotState> lotStateWriter =
       publisher.CreateDataWriter(lotStateTopic);
   var lotStateSample = new ChocolateLotState();
   ```

4. Finally, set some values in the ChocolateLotState data, and write the sample. Look for this comment:

   ```
   // Exercise #2.3 Write data with new ChocolateLotState DataWriter
   ```

   Add the following code after the comment:

   ```
   lotStateSample.lot_id = (uint) (count % 100);
   lotStateSample.lot_status = LotStatusKind.WAITING;
   lotStateWriter.Write(lotStateSample);
   ```

5. Now, compile and run the **chocolate_factoryPublisher** application from the `3_streaming_data/csharp` directory where you generated code. You do not need to

run the **chocolate_factorySubscriber** application, because next we will show you another way to visualize your data.

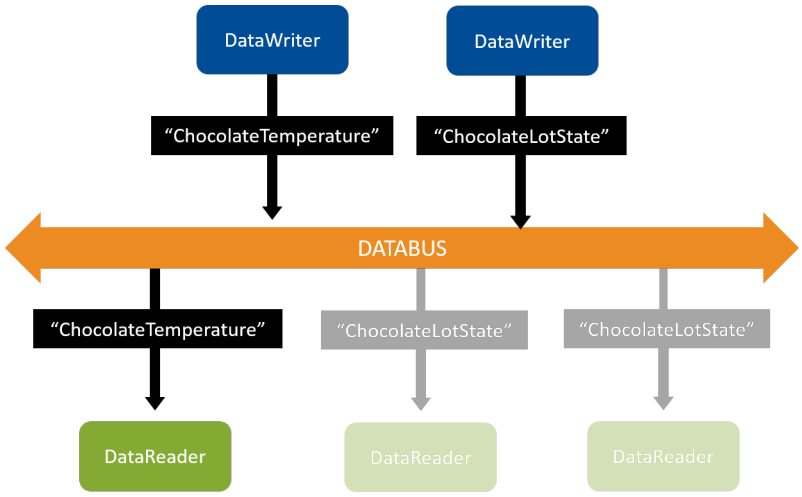


Figure 3.6: You added a second *DataWriter* that writes on the "ChocolateLotState" Topic.

Congratulations! You have added a second *DataWriter* that writes on a new *Topic* with a new data type! In the next module, you will continue adding to these applications to make them more complete.

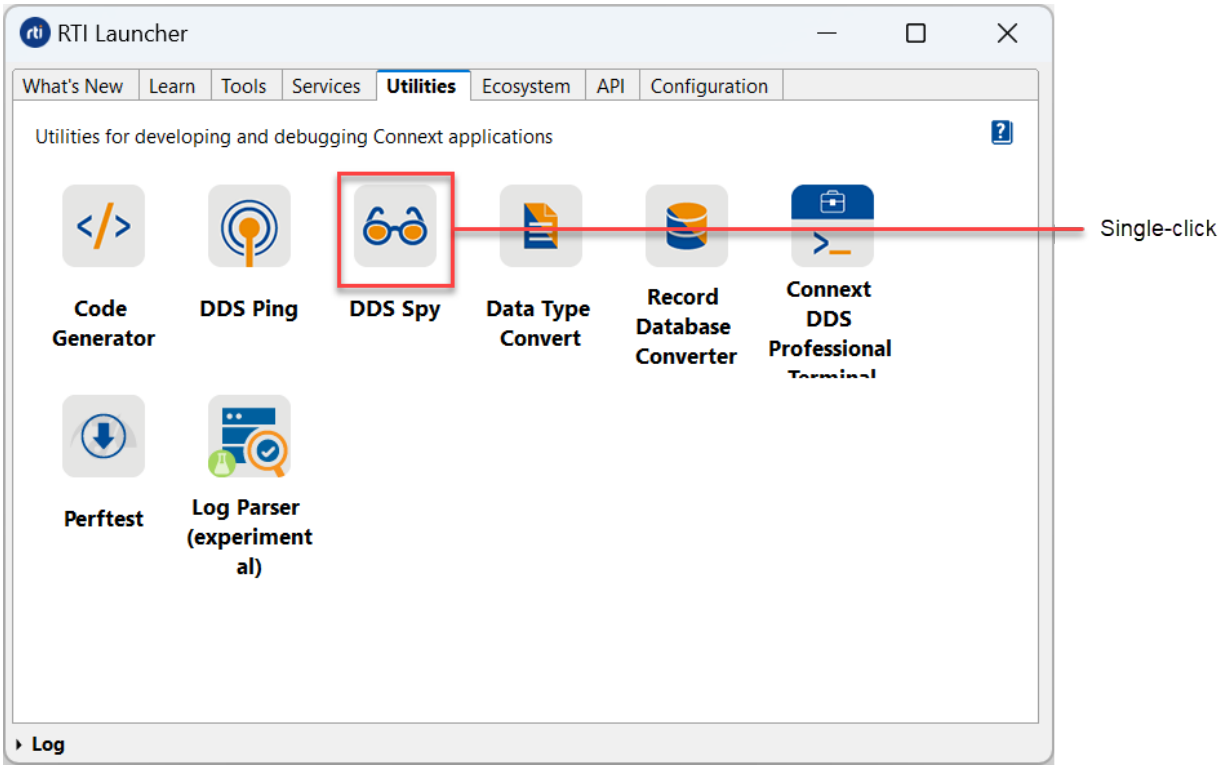
3.5.2 Visualize the Data in rtiddsspy

The *rtiddsspy* utility is a quick way to visualize data when you just need a simple text view. This utility does two things:

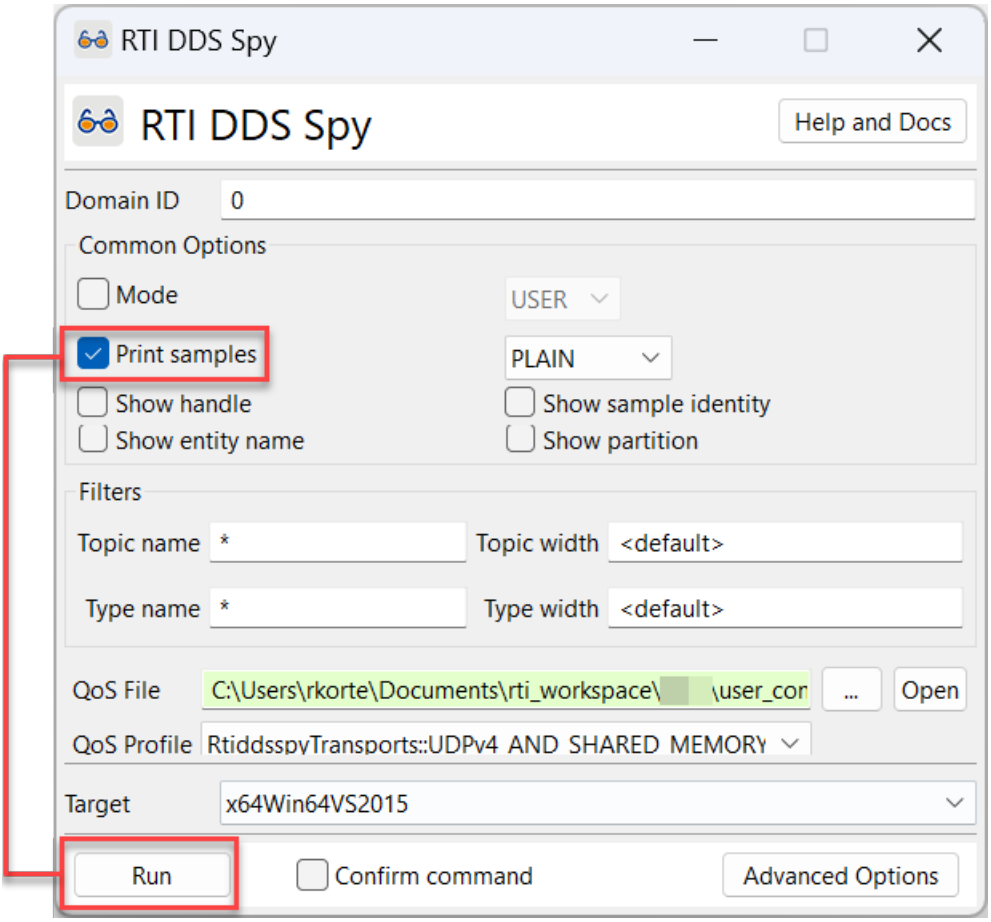
1. Displays the *DataWriters* and *DataReaders* in your system, but in a text format rather than the graphical format of *Admin Console*.
2. Automatically creates *DataReaders* for any *Topic* being written on the network and prints out messages when its *DataReaders* receive data.

rtiddsspy does both of these without requiring very much configuration, making it a convenient tool for debugging when your applications are not communicating, or when you need to quickly see your data. Unlike *Admin Console*, *rtiddsspy* can be run directly on an embedded machine, which makes it useful if you need to debug applications that are not on the same network as a Windows, Linux, or macOS machine.

1. To open *rtiddsspy*, start by opening the *Launcher* tool. (*rtiddsspy* can also be run from the command-line, but *Launcher* provides a useful front-end).
2. Click on the Utilities tab.
3. Click on the DDS Spy icon.



4. In the dialog box that appears, select “Print samples” and click “Run.”



rtiddsspy will show you:

- That it has discovered two *DataWriters*
- The data being published by the two *DataWriters*

```
RTI DDS Spy built with DDS version:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
rtiddsspy is listening for data, press CTRL+C to stop it.

22:20:59 New writer        from 10.30.1.134     : topic="ChocolateTemperature
↪" type="Temperature" name="ChocolateTemperatureDataWriter"
22:20:59 New writer        from 10.30.1.134     : topic="ChocolateLotState"␣
↪type="ChocolateLotState" name="ChocolateTemperatureDataWriter"
22:21:37 New data          from 10.30.1.134     : topic="ChocolateLotState"␣
↪type="ChocolateLotState"
lot_id: 46
lot_status: WAITING

22:21:37 New data          from 10.30.1.134     : topic="ChocolateTemperature
↪" type="Temperature"
sensor_id: "MySensor1"
degrees: 30
```

```
22:21:38 New data          from 10.30.1.134     : topic="ChocolateLotState"␣
↪type="ChocolateLotState"
lot_id: 47
lot_status: WAITING

22:21:38 New data          from 10.30.1.134     : topic="ChocolateTemperature
↪" type="Temperature"
sensor_id: "MySensor1"
degrees: 31

22:21:38 New data          from 10.30.1.134     : topic="ChocolateLotState"␣
↪type="ChocolateLotState"
lot_id: 48
lot_status: WAITING
```

The first two lines indicate that *rtiddsspy* has discovered the two *DataWriters* in your **chocolate_factory_publisher** application. The subsequent lines indicate that *rtiddsspy* is receiving data. Since you selected the "Print Samples" option in *Launcher*, you can also see the contents of the ChocolateLotState data and the Temperature data your *DataWriters* are writing.

After you stop (CTRL+C) *rtiddsspy*, you will see the number of *DataWriters* and *DataReaders* discovered, and samples received:

```
---- Statistics ----
Discovered 10 DataWriters and 7 DataReaders
Received samples (Data, Dispose, NoWriters):
        366, 0, 0       (Topic="ChocolateTemperature"  Type="Temperature")
        366, 0, 0       (Topic="ChocolateLotState"  Type="ChocolateLotState")
```

There are additional values that *rtiddsspy* can display if you use keys and instances (which we haven't talked about yet). For an overview of all the output of *rtiddsspy*, see the RTI DDS Spy documentation.

3.6 Next Steps

Next, we will look a little farther into data design with *Keys and Instances*, then dive into more detail about Quality of Service (QoS) in *Basic QoS*.

Chapter 4

Keys and Instances

Prerequisites	• *Data Types*, including: – Typed data – Interface Definition Language (IDL) – Introduction to data flows – Streaming data • Repository cloned from GitHub here
Time to complete	1 hour
Concepts covered in this module	• Definition of an instance • Benefits of using instances • How key fields identify an instance • How to write a new instance • Instance lifecycles

So far, we've talked about samples. A *DataWriter*, for example, publishes samples of a particular *Topic*. Sometimes, we want to use one *Topic* to publish samples of data for several different objects, such as flights or sensors. *Connext* uses "instances" to represent these real-world objects. (See Table 4.1.)

When you need to represent multiple objects within a DDS *Topic*, you use a key to establish instances. A key in DDS is similar to a primary key in a database—it is a unique identifier of something within your data. An instance is the object identified by the key. A key can be composed of multiple fields in your data as long as they uniquely identify the object you are representing. For example, in an air traffic control system, the key fields might be the airline name and flight number. Samples would be the updated locations of each flight "instance." See other examples of keys, instances, and samples in the following table.

Table 4.1: Examples of Instances and Keys in Distributed Systems

Instance	Key	Data Type	Samples

Commercial flight being tracked	Airline name and flight number, such as:
	Airline: "United Airlines"
	Flight number: 901
		`@key string␣ ↪airline`
		`@key int16␣ ↪flight_num`
		`float latitude`
		`float longitude`
			UA, 901, 37.7749, -122.4194
			UA, 901, 37.7748, -122.4195

Sensor sending data, such as an individual temperature sensor	Unique identifier of that sensor, such as: "tempering-machine-1" or "FirstFloorSensor1"	`@key string␣ ↪sensor_id` `int32 temperature`	**tempering-machine-1,** temperature = 175 **tempering-machine-1,** temperature = 176

Car being monitored	Vehicle identification number (VIN) of the car	`@key string VIN`
		`float latitude`
		`float longitude`
			JH4DA9370MS016526, 37.7749, -122.4194
			JH4DA9370MS016526, 37.7748, -122.4195

Chocolate lot being processed in a factory	Chocolate lot identifier; likely an incrementing number that rolls over	`@key uint32 lot_ ↪num` `LotStatusKind␣ ↪state`	**1,** waiting for cocoa **1,** waiting for sugar

To specify one or more key fields, annotate them with "@key" in the IDL file. For example:

```
// Temperature data type
struct Temperature {

    // Unique ID of the sensor sending the temperature. Each time a sample is
    // written with a new ID, it represents a new sensor instance.
    @key
    string<256> sensor_id;

    // Degrees in Celsius
    int32 degrees;

};
```

You add an `@key` annotation before each field that is part of the unique identifier of your instance.

4.1 Why and How Do We Use Instances?

Not every data model requires that you use instances. You're already dividing up your data into multiple *Topics*. So why use instances at all?

- **Less memory and discovery time**

  Creating a new instance is lighter-weight than creating a new *DataWriter*/*DataReader*/*Topic*. For example, if you're representing airline flights, you could create a new *DataWriter*, *DataReader*, and *Topic* each time a new flight takes off. At a major airport, that's over a thousand flights per day! The problem with a one-*Topic*-per-flight system is that it uses more memory than necessary, and it takes more time for discovery. Using instances to represent unique flights requires less memory, and the instances do not need to be discovered the way *DataWriters* and *DataReaders* discover each other. (See *Discovery*.)

- **Lifecycle**

  Instances allow you to model the behavior of real-world objects that come and go. For example, you can use the instance lifecycle to detect an event such as a flight landing or a chocolate lot finishing. We will go into this in more detail when we talk about the *Instance Lifecycle* below.

- **QoS**

  Several Quality of Service (QoS) policies are applied per-instance. This is a huge benefit, and we'll talk about this in greater detail later in *Basic QoS*.

4.1.1 Writing an Instance

To send a sample of an instance, all you need to do is make sure the key fields are set to the unique ID of your instance. Let's refer back to the following example type, representing temperature sensor data:

```
// Temperature data type
struct Temperature {

    // Unique ID of the sensor sending the temperature. Each time a sample is
    // written with a new ID, it represents a new sensor instance.
    @key
    string<256> sensor_id;

    // Degrees in Celsius
    int32 degrees;

};
```

For the purposes of this example, assume that each physical sensor in our distributed system has a unique ID assigned to it, and this ID maps to the `sensor_id` field in our type. By marking `sensor_id` with the `@key` annotation, we have marked it a key field, and therefore each unique `sensor_id` string we write will represent a different DDS instance.

If we want to write values for multiple sensors, we can change our code so the application takes an ID as a command-line parameter. Then, it can use that ID from the command line as part of the string that becomes the unique sensor ID—for example, `TemperingMachine-<id>`.

```
// Modify the data to be written here
// Specify the sensor instance sending the temperature. ID is passed at
// the command line. Each unique "TemperingMachine-<id>" is a
// unique instance.
temperature.sensor_id = "TemperingMachine-" + sensorId;
temperature.degrees = 32;

writer.Write(temperature)
```

Each time you pass a new ID parameter to the application, you have a new instance in your system. Any time a *DataWriter* writes a sample with a unique value in the sample's key fields, it is writing to a unique instance. A *DataWriter* can write multiple instances just by setting the key fields to unique values.

In our current example, each *DataWriter* writes a single instance; however, as shown in Figure 4.2, the number of instances is not directly related to the number of *DataWriters*. One *DataWriter* can write many instances. Or multiple *DataWriters* can write one or more instances.

You have the flexibility to design your system however you want. Any *DataWriter* of a given *Topic* can be responsible for updating one or more instances within that *Topic*, depending on your requirements.

Tip: Remember: a "sample" is a single update of data. Every time your application calls the *DataWriter*'s write method, the *DataWriter* sends a sample. This is true whether you have instances in your system or not.

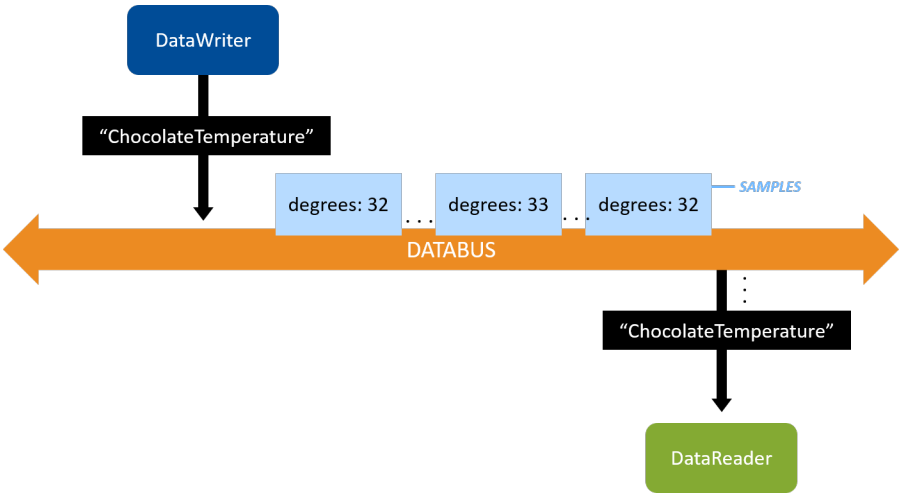


Figure 4.1: Samples without instances

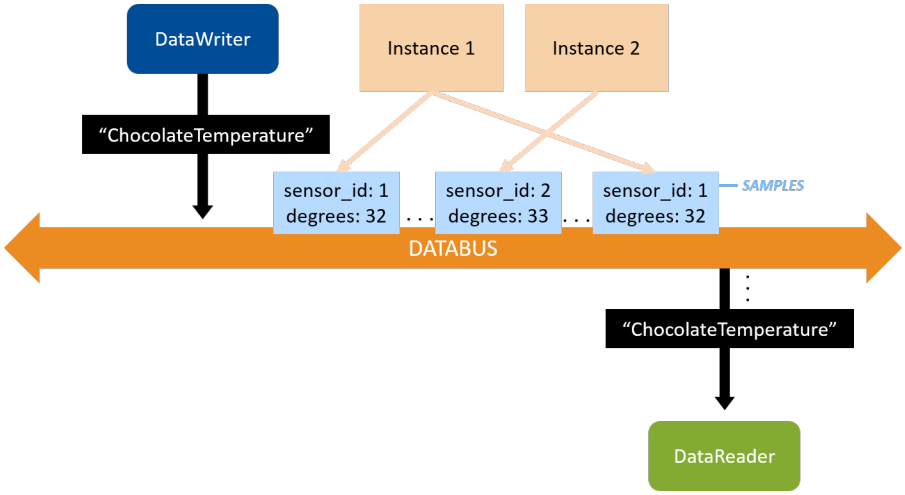


Figure 4.2: Samples with instances

4.1.2 Reading an Instance

An instance lifecycle event will set the `data_available` status to true, similar to what we have previously seen when a new sample is available for a *DataReader*. In the code we have seen so far, the WaitSet wakes when the `data_available` status becomes true and when we can process the sample. When the application gets the `data_available` notification from an instance lifecycle event, retrieving an instance is identical to retrieving a sample (except that you have some additional options). You may remember the code from "Hello World", where a *DataReader* calls `take()` to retrieve data from the middleware, and then iterates over a collection of all the samples:

```
// Take all samples
using var samples = reader.Take();
foreach (var data in samples)
{
      if (sample.Info.ValidData)
      {
          Console.WriteLine(sample.Data);
      }
}
```

In a system with instances, the `Take()` call will return a sequence of samples for all of the instances in the system. (There is also a `Select()` call that can be used to retrieve samples for a particular instance.)

Take a closer look at that code, and you will notice there is a line where we check that the sample is valid:

```
if (sample.Info.ValidData)
```

You may be wondering: what does it mean for the sample to *not* contain valid data? The answer is: a sample can contain data or it can contain information about the instance's lifecycle. The `data_available` event notifies the *DataReader* about these updates in the same way, so the "valid" data flag indicates whether the update is a sample containing data or a sample containing an instance lifecycle update.

4.1.3 Instance Lifecycle

An instance can have the following states, which are all part of the instance lifecycle:

- **Alive**: A *DataReader* has received samples for the instance. (A *DataWriter* has written the instance, and the *DataWriter* is still in the system.)
- **Not alive, disposed**: All *DataReaders* in the system have been notified via a *DataWriter* API call that this instance has been "disposed." For example, you might set up a system in which once a flight has landed, it is disposed (we don't need to track it anymore).
- **Not alive, no writers**: The *DataReader* is no longer receiving samples for the instance. (Every *DataWriter* that wrote that instance has left the system or declared it is no longer writing that instance.)

We can use the state of instances in our application (i.e., the instance lifecycle) to trigger specific logic or to track specific events. What does it mean at a system level when an instance is not alive because of no writers, or if it is disposed? This depends on your system—DDS notifies the *DataReaders* that the instance has changed state, and you can decide what it means in your system. In the next section, *Example: Chocolate Factory*, we'll look at one way that you can use instance states in a chocolate factory example.

All of the information about an instance's lifecycle is part of the **SampleInfo**, which you access using `sample.Info`, the same way we do in the previous example code to check if the sample is valid. Take a look at Instance States, in the RTI Connext Core Libraries User's Manual to see the state diagram describing the instance lifecycle. To review the state data that is specific to instances, you use the `sample.Info.State.Instance` property—you can query this state to see if the instance is `Alive`, `NotAliveDisposed`, or `NotAliveNoWriters`.

So, what is the typical lifecycle of an instance? An instance first becomes alive; this happens when a *DataReader* receives a sample for an instance for the first time. Then the instance may receive updates for some period of time from *DataWriters* publishing to that instance. If an instance becomes not alive, the instance will transition to either "Not alive, no writers" or "Not alive, disposed". An instance may become not alive for a variety of reasons, which are detailed in the following table.

Table 4.2: Instance State Transitions

State	How Change Occurs
Alive	• Any time the instance is written
Not alive, disposed	• Any single *DataWriter* that has written this instance calls `DisposeInstance()`
Not alive, no writers	• All *DataWriters* that have written this instance have been shut down (or lost liveliness; see LIVELINESS QosPolicy, in the RTI Connext Core Libraries User's Manual). • All *DataWriters* that have written this instance call `UnregisterInstance()`. This API indicates that a *DataWriter* is no longer updating a particular instance. For more information on this and the difference between "unregistered" and "disposed," see Managing Instances (Working with Keyed Data Types), in the RTI Connext Core Libraries User's Manual.

4.2 Example: Chocolate Factory

This example illustrates the use of instance states in the context of a chocolate factory, where different stations add ingredients to a chocolate lot, and where the final stage is a tempering station that heats and then cools the chocolate to a specific temperature.

In the chocolate factory that we are creating, there will be two data types: Temperature and ChocolateLotState. We saw these types earlier, in *Data Types*.

Since a temperature reading doesn't "go away" like a flight does, we will not use the instance lifecycle for the "Temperature" *Topic*. Instead, we will focus on the "ChocolateLotState" *Topic*, which *can* utilize instance lifecycle events.

4.2.1 Chocolate Factory: System Overview

We will start building this system with the following applications:

- Monitoring/Control application:
  - Starts off processing a chocolate lot by writing a sample of the "ChocolateLotState" *Topic*, saying that a chocolate lot is ready to be processed.
  - Monitors the "ChocolateLotState" *Topic* as it's updated by the different stations.
  - Finally, reads the "ChocolateTemperature" *Topic* to check that tempering is done correctly.
- Tempering Station application:

  - Writes to the "ChocolateTemperature" *Topic* to let the Monitoring/Control application know the current tempering temperature.
  - Monitors the "ChocolateLotState" *Topic* to see if it needs to process a lot.
  - Processes the lot and updates the state of the lot by writing to the "ChocolateLotState" *Topic*.
- Ingredient Station application (not implemented until a later module):
  - Monitors the "ChocolateLotState" *Topic* to see if it needs to process a lot.
  - Processes the lot (adds an ingredient) and updates the state of the lot by writing to the "ChocolateLotState" *Topic*.

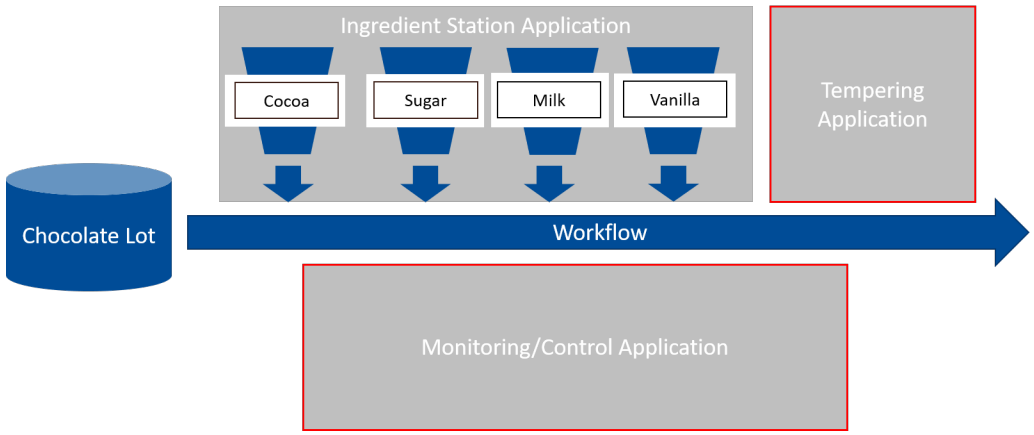


Figure 4.3: There are three applications in this chocolate factory. We will illustrate only the Tempering Application and the Monitoring/Control Application for now.

4.2.2 Chocolate Factory: Data Overview

A chocolate lot is processed by various stations described above. At each station, the chocolate lot transitions through three states:

- Waiting at station
- Processing at station
- Completed by station

To represent the state of each chocolate lot that we are processing, we are going to be using a more complex version of the ChocolateLotState data type than we saw in the last module.

Recall that in the IDL file, the ChocolateLotState data type uses a lot ID (`lot_id`) as the key field:

```
struct ChocolateLotState {
    // Unique ID of the chocolate lot being produced.
    // rolls over each day.
    @key
    int32 lot_id;

    // Which station is producing the status
```

```
    StationKind station;

    // This will be the same as the current station if the station producing
    // the status is currently processing the lot.
    StationKind next_station;

    // Current status of the chocolate lot: Waiting/Processing/Completed
    LotStatusKind lot_status;

};
```

As a chocolate lot receives ingredients from stations in the factory, the stations update the "ChocolateLotState" *Topic*.

The fields `station` and `next_station` in the IDL file represent the current station processing the lot, and the next station that should process the lot. If there is no current station (because the lot is waiting for the first station), `station` will be INVALID_CONTROLLER. If the lot is processing at a station and is not ready to be sent to the next station, the `next_station` field will be INVALID_CONTROLLER. The stations are represented by an enumeration in the IDL (not shown here).

The `lot_status` field describes the status of the lot at a given controller: WAITING, PROCESSING, or COMPLETED. This status is represented in an enumeration in the IDL (not shown here).

When a chocolate lot finishes, the tempering application disposes the instance with `lotStatusWriter.DisposeInstance()`.

Note: Disposing an instance does not necessarily free up memory. There is some subtlety about memory management when using instances, so when you get past the basics, it's a good idea to review Instance Memory Management, in the RTI Connext DDS Core Libraries User's Manual.

4.3 Hands-On 1: Build the Applications and View in Admin Console

To keep this example from getting too complex, we are focusing on just the Monitoring/Control and Tempering applications right now. These applications have more than a single *DataReader* or *DataWriter*, as you can see below:

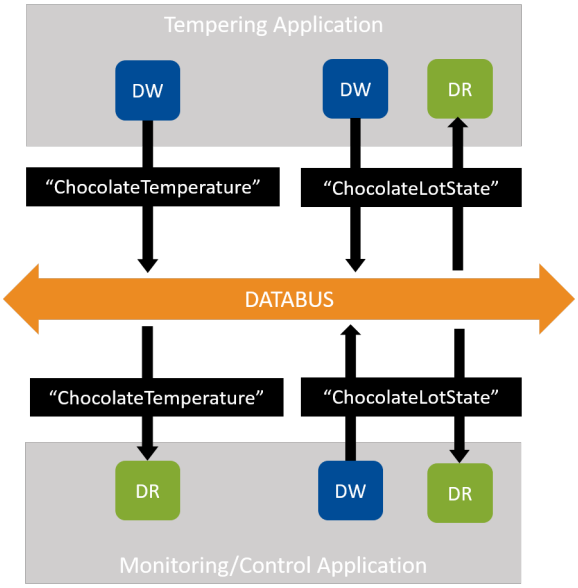


Figure 4.4: Our applications have more than one *DataReader* or *DataWriter*.

Let's look at the "ChocolateLotState" *DataWriters* and *DataReaders* more closely:

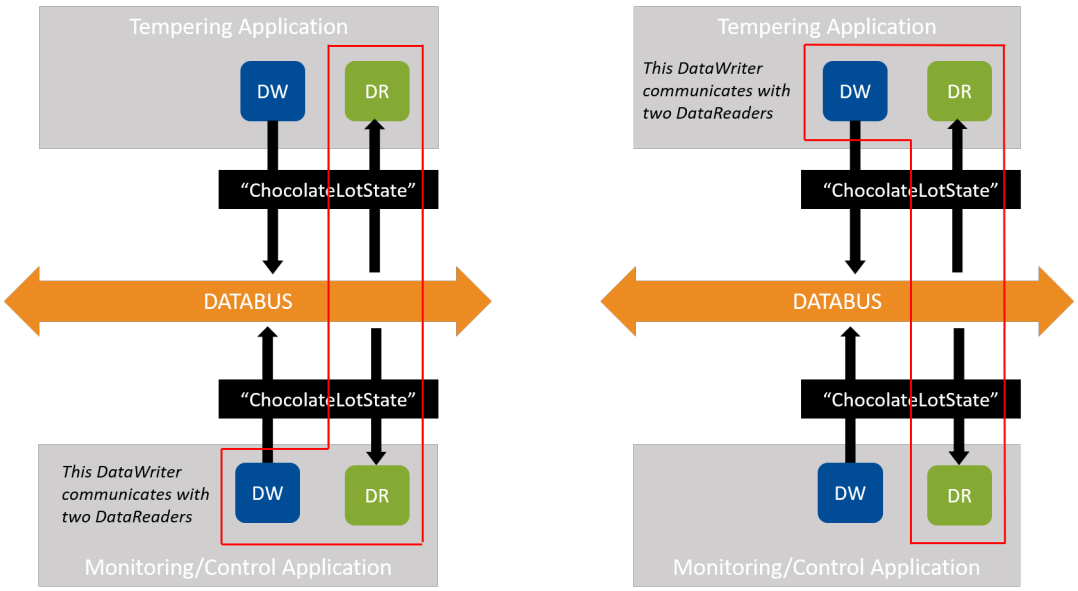


Figure 4.5: In our example, the *DataWriter* in each application communicates with two *DataReaders*—one in its own application and one in another application

Here's a view of the "ChocolateLotState" *DataWriters* and *DataReaders* with samples:

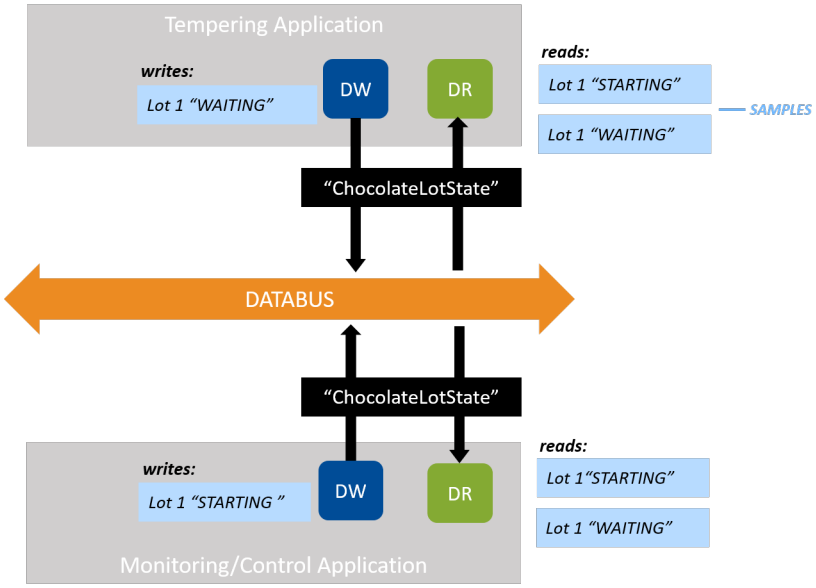


Figure 4.6: Notice that both *DataWriters* are communicating with both *DataReaders*

(Actually, the *DataReader* in the Tempering application only cares about the chocolate lot state if the next station is itself. Right now, the code tells that *DataReader* to ignore any next station state that isn't the Tempering application, but later we will use content filtering to do this instead.)

Note: Figure 4.5 and Figure 4.6 demonstrate that it doesn't matter where the matching *DataWriters* and *DataReaders* are. *They could be in the same application or different applications.* As long as they match in *Topic* and Quality of Service (more on *Basic QoS* later), they can communicate.

4.3.1 Build the Applications

In this exercise, you'll be working in the directory `4_keys_instances/csharp`. This directory was created when you cloned the getting started repository from GitHub in the first module, in *Clone Repository*.

Unlike the previous examples where we built a single application, this directory contains a solution (`KeysInstances.sln`) made of three C# projects:

- `MonitoringCtrlApplication/MonitoringCtrlApplication.csproj`
- `TemperingApplication/TemperingApplication.csproj`
- `ChocolateFactoryTypes/ChocolateFactoryTypes.csproj`

The first two build an application each, and reference the third one, which builds a library with the types we will generate from `chocolate_factory.idl`.

1. Run the Code Generator (*rtiddsgen*) to generate the C# code for the IDL file:

   Linux

```
$ cd 4_keys_instances
$ rtiddsgen -language c# -d csharp/ChocolateFactoryTypes chocolate_
↪factory.idl
```

macOS

```
$ cd 4_keys_instances
$ rtiddsgen -language c# -d csharp/ChocolateFactoryTypes chocolate_
↪factory.idl
```

Windows

```
> cd 4_keys_instances
> rtiddsgen -language c# -d csharp/ChocolateFactoryTypes -ppDisable␣
↪chocolate_factory.idl
```

`-ppDisable` disables the preprocessor. It is necessary for running *rtiddsgen* on a Windows system if the preprocessor is not in your path. You can only use `-ppDisable` if your IDL is simple, as it is here—otherwise you must add the preprocessor to your path. See Command-Line Arguments for rtiddsgen, in the RTI Connext DDS Code Generator User's Manual if you want more information.

We're generating the code in the `csharp/ChocolateFactoryTypes` directory, which already contains a project file (`ChocolateFactoryTypes.csproj`) that will compile the types as a .NET library.

For more information, see *Run Code Generator* in the previous example.

2. Configure the NuGet package source.

   In this exercise the `rtiddsgen` command we just ran only generates the type files, not the project files or `NuGet.Config`. In order to have `dotnet` find the *Connext* package, you have two options:

   - Copy the NuGet.Config file that you generated in the previous exercise from `2_hello_world/csharp` or `3_streaming_data/csharp` to `4_keys_instances/csharp`, or
   - Configure NuGet globally by running the following command:

     ```
     $ dotnet nuget add source <installdir>/lib/dotnet --name RTI
     ```

   If you don't configure the package source, the package will be fetched from nuget.org (which requires a license to run).

3. Using the dotnet CLI from `4_keys_instances/csharp`, enter:

   ```
   $ dotnet build
   ```

   These application projects are configured to build .NET 5 applications. If you want to build, for example, .NET Core 3.1 applications, edit `MonitoringCtrlApplication.csproj` and `TemperingApplication.csproj` and change `<TargetFramework>net5</TargetFramework>` to `<TargetFramework>netcoreapp3.1</TargetFramework>`.

4.3.2 Run Multiple Copies of the Tempering Application

For now, we will focus on visualizing the Tempering application's instances. Run multiple copies of the Tempering application. Be sure to specify different sensor IDs at the command line:

1. Open a terminal window and run the Tempering application with a sensor ID as a parameter:

```
$ dotnet run --project TemperingApplication -- --sensor-id 1
```

2. Open a second terminal window and run the Tempering application with a different sensor ID as a parameter:

```
$ dotnet run --project TemperingApplication -- --sensor-id 2
```

Note: You should run from the `4_keys_instances/csharp` directory because the examples use Quality of Service (QoS) information from the file `USER_QOS_PROFILES.xml` in that directory. We'll talk more about QoS in a later module.

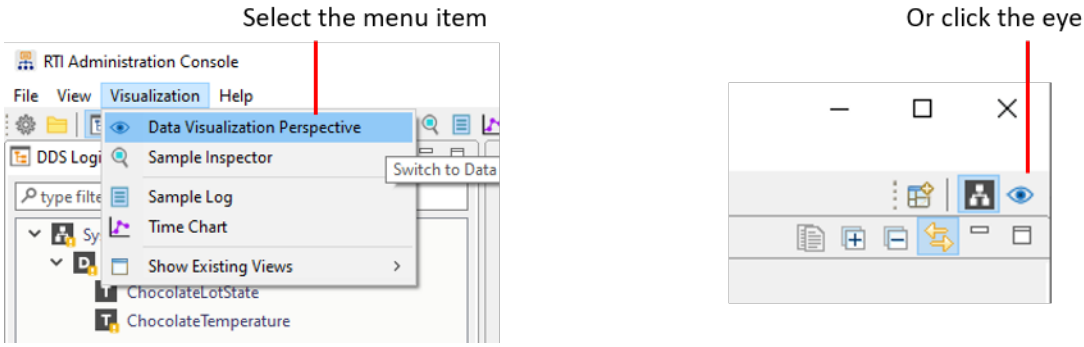
Note that `--project TemperingApplication` tells dotnet to run the project in the `TemperingApplication` directory. Arguments to the application (`--sensor-id 2`) are specified after `--`.

You should see output similar to the following in both windows:

```
ChocolateTemperature Sensor with ID: 1 starting
Waiting for lot
Waiting for lot
Waiting for lot
Waiting for lot
Waiting for lot
Waiting for lot
Waiting for lot
Waiting for lot
```

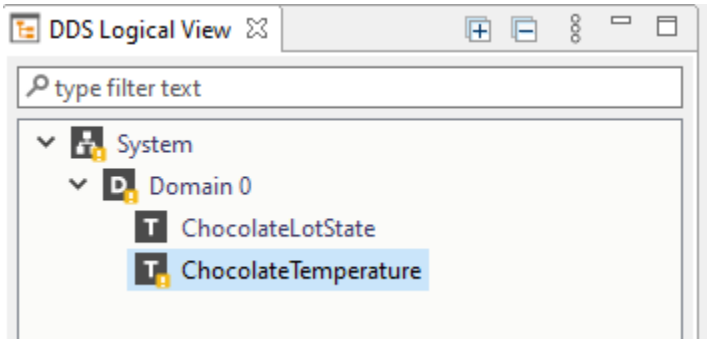
4.3.3 View the Data in Admin Console

1. Make sure your two Tempering applications from the previous step are still running.
2. Like you did in *Hands-On 2: Viewing Your Data*, open up *Admin Console* and switch to the Data Visualization Perspective.



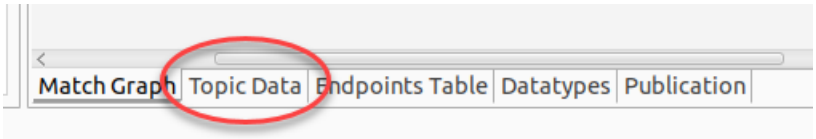
(If the Data Visualization Perspective is grayed out, you may already be in that view.)

3. Select the "ChocolateTemperature" *Topic* in the Logical View:



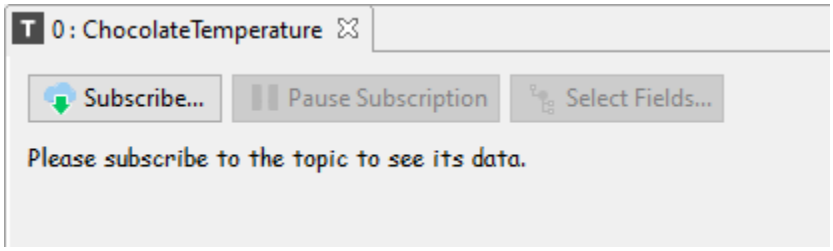
Note: Don't worry about the warnings. We'll look at those in *Hands-On 4: Debugging the System and Completing the Application*.

4. Select the Topic Data tab at the bottom of the window.

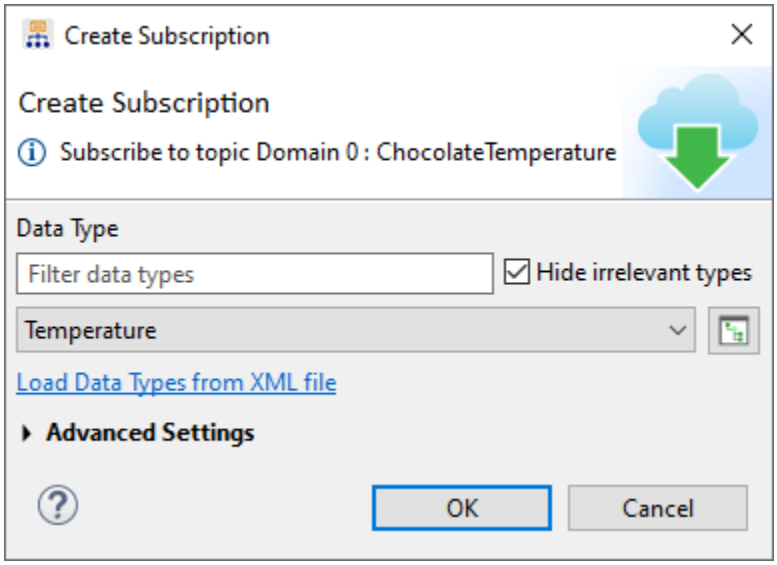


5. Subscribe to the "ChocolateTemperature" *Topic*.

   Click the Subscribe button:

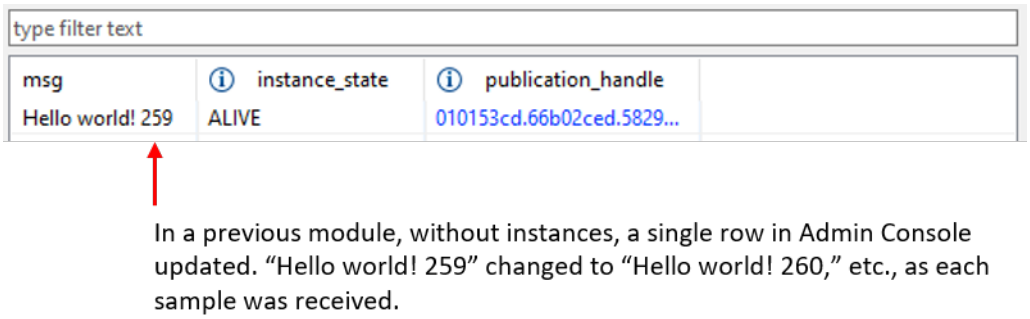


Then click OK:



6. View the samples coming in.

   Recall that in the "Hello World" hands-on exercise, we did not yet have instances. Every sample updated in a single row in *Admin Console*:



   With instances, however, you see multiple rows in *Admin Console*:

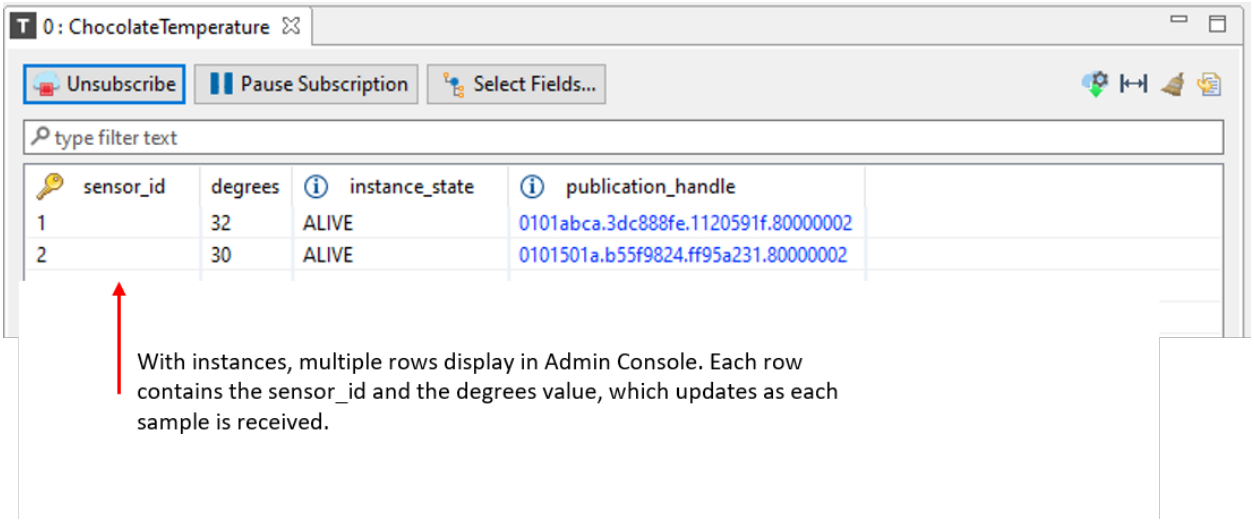


Figure 4.7: Since `sensor_id` is a key field for the ChocolateTemperature data type, Admin Console understands that every sample is an update to a particular `sensor_id` instance. Therefore, Admin Console can display each instance separately, and you can see the degrees value from each sensor displayed separately.

Admin Console can show you one row per instance because it recognizes each instance as a different object.

7. Click Unsubscribe.

   In *Hands-On 4: Debugging the System and Completing the Application*, we will debug your system using *Admin Console*. We do not want *Admin Console* subscribing to any data for that exercise.

4.4 Hands-On 2: Run Both Applications

4.4.1 Run Monitoring and Tempering Applications

1. In the previous hands-on, you ran multiple copies of the tempering application—quit them now if they're still running.
2. From any command prompt window, run the Monitoring/Control application, which you already built in *Build the Applications*:

```
$ dotnet run --project MonitoringCtrlApplication
```

   You should see output like the following:

```
Starting lot:
[lot_id: 0 next_station: TEMPERING_CONTROLLER]

Starting lot:
[lot_id: 1 next_station: TEMPERING_CONTROLLER]
Received lot update:
[lot_id: 1, station: INVALID_CONTROLLER, next_station: TEMPERING_
```

```
↪CONTROLLER, lot_status: WAITING]

Starting lot:
[lot_id: 2 next_station: TEMPERING_CONTROLLER]
Received lot update:
[lot_id: 2, station: INVALID_CONTROLLER, next_station: TEMPERING_
↪CONTROLLER, lot_status: WAITING]
```

This output shows the two things that the Monitoring/Control application is doing:

- Starts the chocolate lots by writing a sample, which indicates that the next station is the TEMPERING_CONTROLLER. This line shows the application in its control capacity ("I'm starting the lot"):

```
Starting lot:
[lot_id: 1 next_station: StationKind::TEMPERING_CONTROLLER]
```

- Reads the current state of the chocolate lot and prints that to the console. This line shows the application in its monitoring capacity ("right now, the lot is starting").

```
[lot_id: 1, station: StationKind::INVALID_CONTROLLER , next_station:
StationKind::TEMPERING_CONTROLLER , lot_status:
LotStatusKind::WAITING ]
```

Since the Monitoring/Control application is the only application running right now, the lots will always be WAITING. In the next step, this will change when you start the Tempering application.

3. In your other command prompt window, run the Tempering application with a sensor ID as a parameter:

```
$ dotnet run --project TemperingApplication -- --sensor-id 1
```

You should see output like the following:

```
Waiting for lot
ChocolateTemperature Sensor with ID: 1 starting
Processing lot #16
Waiting for lot
Waiting for lot
Processing lot #17
Waiting for lot
Waiting for lot
Processing lot #18
```

Note: You may notice that the Monitoring/Control application starts with lot #0, but the Tempering application's *DataReader* does not receive notifications about lot #0, or any lots from before it starts up. We will talk about this more when we talk about the Durability QoS in *Basic QoS*.

---

4. Review the output of the Monitoring/Control application now that you have started the Tempering application. You can see that the lot states changed to "PROCESSING" at the tempering station:

```
Starting lot:
[lot_id: 16 next_station: StationKind::TEMPERING_CONTROLLER ]
Received Lot Update:
[lot_id: 16, station: StationKind::INVALID_CONTROLLER , next_station:
StationKind::TEMPERING_CONTROLLER , lot_status:
LotStatusKind::WAITING ]
Received Lot Update:
[lot_id: 16, station: StationKind::TEMPERING_CONTROLLER ,
next_station: StationKind::INVALID_CONTROLLER , lot_status:
LotStatusKind::PROCESSING ]

Starting lot:
[lot_id: 17 next_station: StationKind::TEMPERING_CONTROLLER ]
Received Lot Update:
[lot_id: 17, station: StationKind::INVALID_CONTROLLER , next_station:
StationKind::TEMPERING_CONTROLLER , lot_status:
LotStatusKind::WAITING ]
Received Lot Update:
[lot_id: 17, station: StationKind::TEMPERING_CONTROLLER ,
next_station: StationKind::INVALID_CONTROLLER , lot_status:
LotStatusKind::PROCESSING ]
```

5. Quit both applications.

4.4.2 Review the Tempering Application Code

This code is significantly more complex than the previous examples, because we are starting to build applications that are closer to real-world applications, with multiple *DataWriters* and *DataReaders*. Let's start by examining the `TemperingApplication.cs` code.

Some of this file should look familiar to you. This application is writing temperature data in the `PublishTemperature` method. Even though a real tempering machine would raise the temperature of chocolate and then lower it to around freezing, in this example we're just sending a "temperature" that's close to freezing (in Fahrenheit).

```
private void PublishTemperature(
    DataWriter<Temperature> writer,
    string sensorId)
{
    // Create temperature sample for writing
    var temperature = new Temperature();
    while (!shutdownRequested)
    {
        // Modify the data to be written here
        temperature.sensor_id = sensorId;
        temperature.degrees = rand.Next(30, 33); // Random value between 30␣
↪and 32

        writer.Write(temperature);

        Thread.Sleep(100);
```

```
    }
}
```

Now take a look at the `ProcessLot` method. This looks similar to the `ProcessData` methods you have seen in the previous hands-on exercises. However, this function takes both a *DataWriter* and a *DataReader* instead of just a *DataReader*. This function calls `Take()` to retrieve data from the *DataReader*'s queue just like we did in the previous hands-on exercises. However, after taking that data, it updates the state of the lot to PROCESSING. In this example, instead of actually processing the lot, we're going to sleep for 5 seconds to represent the processing.

```
// Take all samples. Samples are loaned to application, loan is
// returned when LoanedSamples is Disposed. ValidData iterates only over
// samples such that sample.Info.ValidData is true.
using var samples = lotStateReader.Take();
foreach (var sample in samples.ValidData())
{
      if (sample.next_station == StationKind.TEMPERING_CONTROLLER)
      {
         Console.WriteLine("Processing lot #" + sample.lot_id);

         // Send an update that the tempering station is processing lot
         var updatedState = new ChocolateLotState(sample)
         {
            lot_status = LotStatusKind.PROCESSING,
            next_station = StationKind.INVALID_CONTROLLER,
            station = StationKind.TEMPERING_CONTROLLER
         };
         lotStateWriter.Write(updatedState);

         // "Processing" the lot.
         Thread.Sleep(5000);

         // Exercise #3.1: Since this is the last step in processing,
         // notify the monitoring application that the lot is complete
         // using a dispose
      }
}
```

4.5 Hands-On 3: Dispose the ChocolateLotState

The Tempering application will use the NOT_ALIVE_DISPOSED instance state to indicate that a chocolate lot has finished processing.

4.5.1 Add Code to Tempering Application to Dispose ChocolateLotState

Add a call to `DisposeInstance()` for the ChocolateLotState data in the Tempering application, since it is the last step in the chocolate factory.

1. In `TemperingApplication.cs`, find the comment:

```
// Exercise #3.1: Since this is the last step in processing,
// notify the monitoring application that the lot is complete
// using a dispose
```

2. Add the following code after the comment to dispose the ChocolateLotState:

```
lotStateWriter.DisposeInstance(
   lotStateWriter.LookupInstance(updatedState));
Console.WriteLine("Lot completed");
```

4.5.2 Detect the Dispose in the Monitoring Application

Open `MonitoringCtrlApplication.cs` to look at the Monitoring/Control application. Note that it does two things right now:

1. Kicks off processing chocolate lots, by sending the first update to the "ChocolateLotState" *Topic*. Since we are skipping all the ingredient station steps (for now) in this example, the Monitoring/Control application sends the lot directly to wait at the tempering machine, as you can see in the `publish_start_lot` function:

```
private void PublishStartLot(
    DataWriter<ChocolateLotState> writer,
    uint lotsToProcess)
{
    ...
    sample.lot_id = count % 100;
    sample.lot_status = LotStatusKind.WAITING;
    sample.next_station = StationKind.TEMPERING_CONTROLLER;

    Console.WriteLine("Starting lot:");
    Console.WriteLine($"[lot_id: {sample.lot_id} next_station: {sample.
↪next_station}]");
    writer.Write(sample);
    ...
}
```

2. Monitors the lot state, and prints out the current state of the lots, as you can see in the `monitor_lot_state` function:

```
private int MonitorLotState(DataReader<ChocolateLotState> reader)
{
    int samplesRead = 0;
    using var samples = reader.Take();
    foreach (var sample in samples)
```

```
    {
        Console.WriteLine("Received Lot Update: ");
        if (sample.Info.ValidData)
        {
            Console.WriteLine(sample.Data);
            samplesRead++;
        }
        else
        {
            // Exercise #3.2: Detect that a lot is complete by checking␣
↪for
            // the disposed state.
        }
    }

    return samplesRead;
}
```

Add code to check the instance state and print the ID of the completed lot:

1. In `MonitoringCtrlApplication.cs`, in the `MonitorLotState` method, find the comment:

```
// Exercise #3.2: Detect that a lot is complete by checking for
// the disposed state.
```

2. Add the following code after the comment:

```
if (sample.Info.State.Instance == InstanceState.NotAliveDisposed)
{
   // Create a sample to fill in the key values associated
   // with the instance
   var keyHolder = new ChocolateLotState();
   reader.GetKeyValue(keyHolder, sample.Info.InstanceHandle);
   Console.WriteLine($"[lot_id: {keyHolder.lot_id} is completed]");
}
```

The code you just added:

- Checks that the instance state was NOT_ALIVE_DISPOSED.
- Creates a temporary ChocolateLotState object, and passes it to `reader.GetKeyValue()` to get the value of the key fields associated with the instance handle.
- Prints out the ID of the lot that is completed.

4.5.3 Run the Applications

In one command terminal:

```
$ dotnet run --project MonitoringCtrlApplication
```

In another terminal:

```
$ dotnet run --project TemperingApplication -- --sensor-id 1
```

You should see the following output from the Tempering application:

```
Waiting for lot
Processing lot #1
Lot completed
Waiting for lot
Waiting for lot
Processing lot #2
Lot completed
```

You should see the following output from the Monitoring/Control application:

```
Starting lot:
[lot_id: 1 next_station: StationKind::TEMPERING_CONTROLLER ]
Received Lot Update:
[lot_id: 1, station: StationKind::INVALID_CONTROLLER , next_station:
StationKind::TEMPERING_CONTROLLER , lot_status:
LotStatusKind::WAITING ]
Received Lot Update:
[lot_id: 1, station: StationKind::TEMPERING_CONTROLLER ,
next_station: StationKind::INVALID_CONTROLLER , lot_status:
LotStatusKind::PROCESSING ]
Received Lot Update:
[lot_id: 1 is completed]

Starting lot:
[lot_id: 2 next_station: StationKind::TEMPERING_CONTROLLER ]
Received Lot Update:
[lot_id: 2, station: StationKind::INVALID_CONTROLLER , next_station:
StationKind::TEMPERING_CONTROLLER , lot_status:
LotStatusKind::WAITING ]
Received Lot Update:
[lot_id: 2, station: StationKind::TEMPERING_CONTROLLER ,
next_station: StationKind::INVALID_CONTROLLER , lot_status:
LotStatusKind::PROCESSING ]
Received Lot Update:
[lot_id: 2 is completed]
```

Notice that when we first ran the Monitoring/Control application, we saw WAITING and PROCESSING statuses. Now we also see a "completed" status. The "completed" status occurs because we added the code to dispose the instance in the Tempering application and because we checked for the disposed status in the Monitoring/Control application.

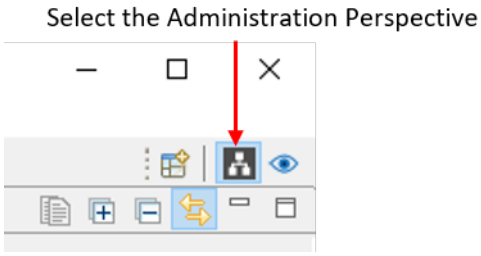
4.6 Hands-On 4: Debugging the System and Completing the Application

This hands-on is not specific to using instances, but it will make you more familiar with how to create multiple *DataReaders* and *DataWriters* in an application. This will help you as you continue with upcoming modules, and the exercises become more complex.

4.6.1 Debug in Admin Console

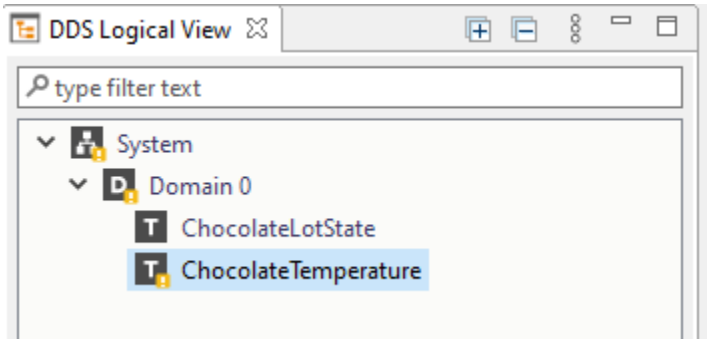
So far, we have only used *Admin Console* for data visualization. Now we will use it for system debugging.

1. Make sure your applications (**monitoring_ctrl_application** and **tempering_application**) are running.
2. Like you did in *Hands-On 2: Viewing Your Data*, in the first module, open up the *Admin Console* tool.
3. Click Unsubscribe if you haven't already in Hands-On 1.
4. Choose the Administration Perspective toolbar icon in the top right corner of the window.

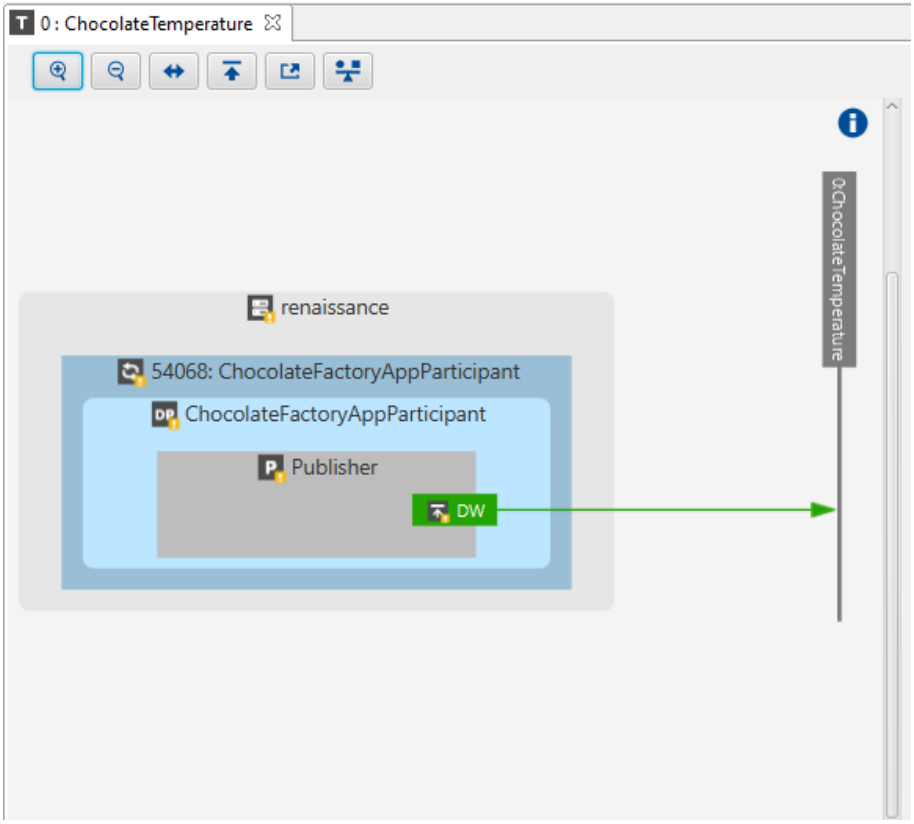


   You should see a Logical View pane in the upper left.

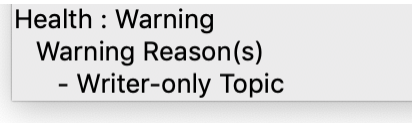
5. Notice that one of your *Topics* has a warning:



6. Click on the *Topic* with a warning, and you should see a visualization of that *Topic* and *DataWriter* in the main window:



7. Hover your mouse over the writer (DW). You should see this warning message:



The reason for this warning is because there are no *DataReaders* reading the "ChocolateTemperature" *Topic*.

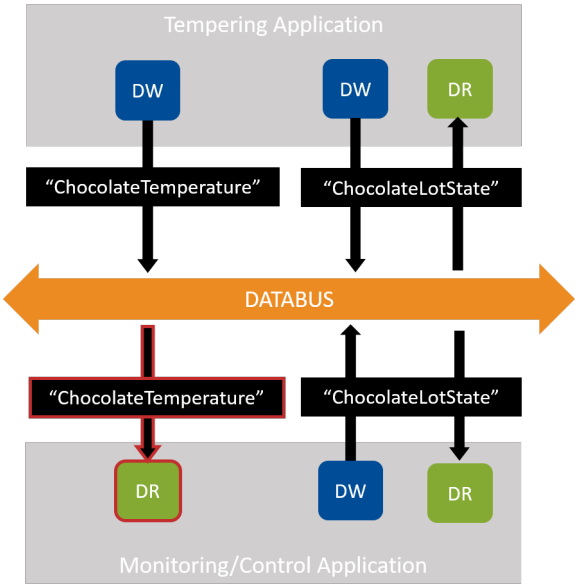


Figure 4.8: Using Admin Console, we identified that the *DataReader* for the "ChocolateTemperature" Topic is missing from the Monitoring/Control Application.

If you look more carefully at the code in `MonitoringCtrlApplication.cs`, you'll see that although the Monitoring/Control application reads and writes to the "ChocolateLotState" *Topic*, it never creates a *DataReader* to read the "ChocolateTemperature" *Topic*!

8. Quit the applications.

4.6.2 Add the ChocolateTemperature DataReader

In this exercise, you will add the missing *DataReader*.

First, review the `RunExample` function in `MonitoringCtrlApplication.cs`. Notice that it uses a single *DomainParticipant* to create a *Publisher* and a *Subscriber*. (See *Publishers, Subscribers, and DomainParticipants*.)

```
DomainParticipant participant = DomainParticipantFactory.Instance
    .CreateParticipant(domainId);

...
// A Publisher allows an application to create one or more DataWriters
// Publisher QoS is configured in USER_QOS_PROFILES.xml
Publisher publisher = participant.CreatePublisher();

...
// A Subscriber allows an application to create one or more DataReaders
// Subscriber QoS is configured in USER_QOS_PROFILES.xml
Subscriber subscriber = participant.CreateSubscriber();
```

In the following steps, you're going to add a second *DataReader* to the Monitoring/Control application that reads the ChocolateTemperature *Topic*. In *Hands-On 2: Add a Second DataWriter* (in Section 3), you used

a single *Publisher* to create multiple *DataWriters*. In this example, you will use a single *Subscriber* to create multiple *DataReaders*.

To add a second *DataReader* to the Monitoring/Control application:

1. Create a "ChocolateTemperature" *Topic* that the *DataReader* will be reading. (Right now, this application only has the "ChocolateLotState" *Topic* defined, so you must add a "ChocolateTemperature" *Topic* to be used by the new *DataReader*.)

   In `MonitoringCtrlApplication.cs`, find this comment in the code:

   ```
   // Exercise #4.1: Add a Topic for Temperature to this application
   ```

   Add the following code immediately after that comment to create a "ChocolateTemperature" Topic, which uses the Temperature data type:

   ```
   Topic<Temperature> temperatureTopic = participant.CreateTopic
   ↪<Temperature>(
         CHOCOLATE_TEMPERATURE_TOPIC.Value);
   ```

   Tip: It is a best practice to use a variable defined inside the IDL file while creating the *Topic* instead of passing a string literal. Using a variable ensures that our *Topic* names are identical in our applications. (Recall that if they're not identical, our *DataWriter* and *DataReader* won't communicate.) It is convenient to use the IDL file to define the variable, since the IDL file is used by all of our applications. Here is what the *Topic* name looks like in the IDL file, which can be found in the `4_keys_instances` directory:

   ```
   const string CHOCOLATE_TEMPERATURE_TOPIC = "ChocolateTemperature";
   ```

   Recall that one data type can be associated with several *Topic* names. Therefore, you may be defining more *Topic* names than data types in your IDL.

   ---

2. Add a new *DataReader*.

   Now that we have a "ChocolateTemperature" *Topic*, we can add a new *DataReader*. Find this comment in the code:

   ```
   // Exercise #4.2: Add a DataReader for Temperature to this application
   ```

   Remember from *Details of Receiving Data* (in Section 2) that the StatusCondition defines a condition we can attach to a WaitSet. The WaitSet will wake when the StatusCondition becomes true.

   Add a new temperature *DataReader* and set up its StatusCondition by adding this code immediately after the comment:

   ```
   DataReader<Temperature> temperatureReader =
         subscriber.CreateDataReader(temperatureTopic);

   // Obtain the DataReader's Status Condition
   StatusCondition temperatureStatusCondition = temperatureReader.
   ↪StatusCondition;
   ```

```
// Enable the 'data available' status.
temperatureStatusCondition.EnabledStatuses = StatusMask.DataAvailable;

// Associate a handler with the status condition. This will run when the
// condition is triggered, in the context of the dispatch call (see␣
↪below)
temperatureStatusCondition.Triggered += _ =>␣
↪MonitorTemperature(temperatureReader);
```

3. Associate the new *DataReader*'s status condition with the WaitSet. Find this comment in the code:

```
// Exercise #4.3: Add the new DataReader's StatusCondition to the Waitset
```

Add this line below the comment:

```
waitset.AttachCondition(temperatureStatusCondition);
```

4. Add a function that processes the temperature data and prints a message if it ever exceeds 32 degrees. Find this comment in the code:

```
// Exercise #4.4: Add monitor_temperature function
```

Add the following function to print out a message if the temperature exceeds 32 degrees immediately after that comment:

```
private void MonitorTemperature(DataReader<Temperature> reader)
{
    using var samples = reader.Take();
    foreach (var data in samples.ValidData())
    {
        // Receive updates from tempering station about chocolate␣
↪temperature.
        // Only an error if over 32 degrees Fahrenheit.
        if (data.degrees > 32)
        {
            Console.WriteLine("Temperature high: " + data);
        }
    }
}
```

4.6.3 Run the Applications

Recompile and run the Tempering and Monitoring/Control applications again.

Notice that *Admin Console* no longer shows a warning, because the Monitoring/Control application now has a *DataReader* that is reading the "ChocolateTemperature" *Topic* that the Tempering application is writing.

Note: We've intentionally written the applications in a way that you won't actually see temperature output

printed to the console, to keep the code simpler, but we'll change this in a later exercise.

Congratulations! In this module, you have learned to do the following:

- Change an instance's state to NOT_ALIVE_DISPOSED.
- Get notified of the state change.
- Debug your system using *Admin Console*.
- Add a new *DataReader* to an application.

4.7 Next Steps

Next we will look at how to configure the behavior of your *DataWriters* and *DataReaders* using Quality of Service.

Chapter 5

Basic QoS

Prerequisites	• *Keys and Instances*, including: – Instances – Key fields • Repository cloned from GitHub here
Time to complete	1 hour
Concepts covered in this module	• Reliability • Durability • History depth • Deadline • QoS compatibility and matching • QoS profiles

Connext provides a variety of configuration options to change how your data is delivered and to fine-tune the performance of your system. These configuration options are called Quality of Service, or QoS. QoS settings are configured on *DataWriters* and *DataReaders* (and on other DDS objects, such as *Publishers* and *Subscribers*).

Some of the basic QoS policies configured on *DataWriters* and *DataReaders* include the following:

- **Reliability QoS Policy:** Should the arrival of each sample be guaranteed, or is best-effort enough and the risk of missing a sample acceptable?
- **History QoS Policy:** How many samples should be stored for reliability purposes?
- **Resource Limits QoS Policy:** What is the maximum allowed size of a *DataWriter*'s or *DataReader*'s queue due to memory constraints?
- **Durability QoS Policy:** Should samples be stored and automatically sent to new *DataReaders* as they start up? If yes, how many samples?
- **Deadline QoS Policy:** How do we detect that streaming data is being sent at an acceptable rate?

There are many more QoS policies that control discovery, fault-tolerance, and more. We will focus on just a few in this module. For a broader look at the QoS policies available, see the QoS Reference Guide. Although we will be focusing on QoS policies that are set on *DataWriters* and *DataReaders*, you can set QoS policies on other DDS objects; in *Hands-On 1: Update One QoS Profile in the Monitoring/Control Application*, we will set a QoS policy on a *DomainParticipant*.

5.1 Request-Offered QoS Policies

Some QoS policies are "Request-Offered," meaning that a *DataWriter* offers a level of service, and a *DataReader* requests a level of service. If the *DataWriter* offers a level of service that's the same as or higher than the *DataReader* requests, the QoS policies are matching. If the *DataReader* requests a higher level of service than the *DataWriter* is offering, the QoS policies are considered "incompatible," and the *DataWriter* will not send data to the *DataReader*. Request-Offered semantics are abbreviated with the term "RxO."

One example of a Request-Offered QoS policy is Reliability. Reliable delivery is considered a higher level of service than best-effort delivery. Although a *DataWriter* may offer reliable delivery, not every *DataReader* it's communicating with needs reliability. Let's look at an example of a system that tracks aircraft. Some *DataReaders*—such as those in the air traffic control application—need the aircraft locations reliably, meaning they cannot miss an update. Other *DataReaders*—such as those in an application that updates flight times for customers to view departures and arrivals—do not need to receive every flight position update. The radar application *DataWriter* will send aircraft positions to all the *DataReaders* that need it, but not all of those *DataReaders* need it reliably.

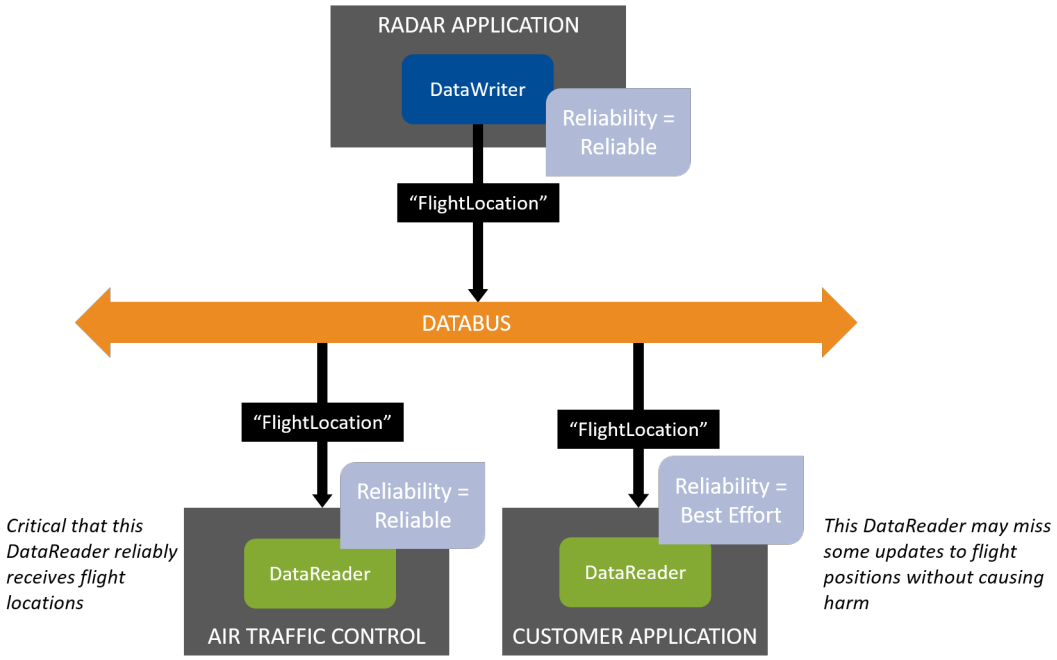


Figure 5.1: The Customer Application's *DataReader* does not need flight location data reliably. The Radar Application's *DataWriter* can send to both *DataReaders* because its Reliability QoS Policy is the same as or higher than theirs.

Now imagine that the system in Figure 5.1 is misconfigured, so the *DataWriter* offers Best Effort data, which is a lower level of service. In this situation, the Air Traffic Control Application can't receive data reliably, even though it is critical that it receive every update. This is an error in the configuration of the system, and *Connext* treats it that way: the *DataWriter* and *DataReader* of these applications will not communicate, and the *DataReader* and *DataWriter* will instead be notified that they have incompatible QoS policies.

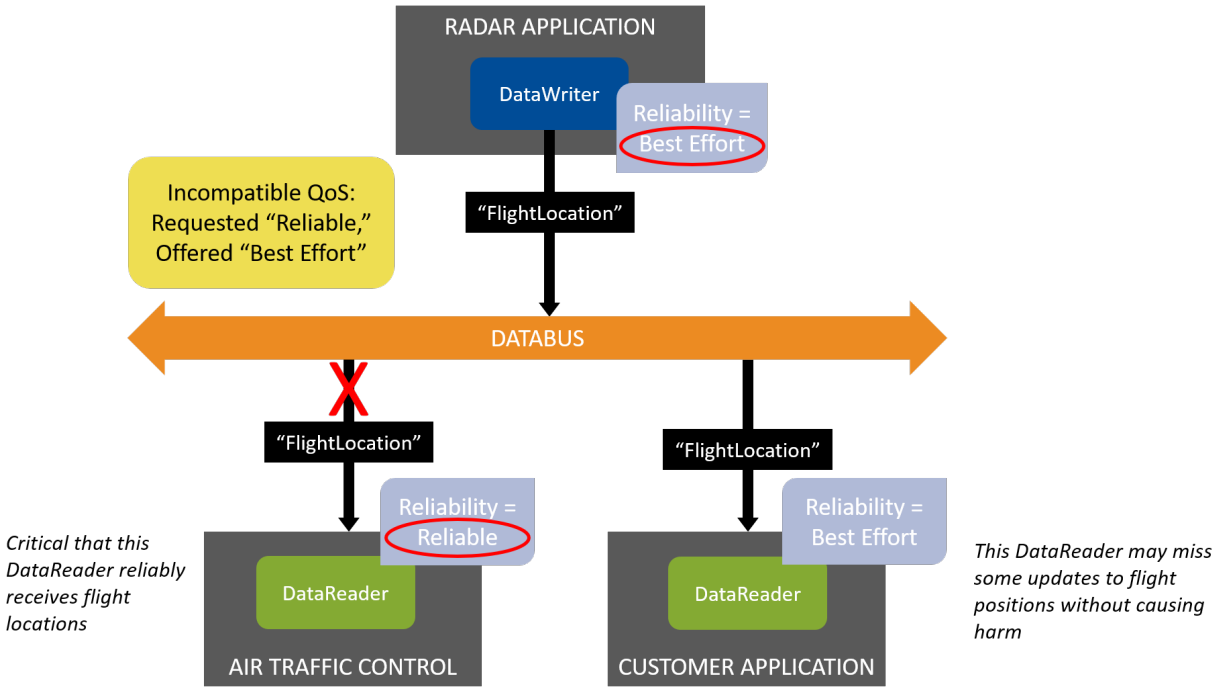


Figure 5.2: The *DataWriter* is misconfigured to offer only Best Effort reliability, and now it does not match with the Reliable *DataReader*. The *DataWriter* is still compatible with the Customer Application's *DataReader*.

In previous modules, our applications have only been notified of data being available (see *Details of Receiving Data*). In *Hands-On 3: Incompatible QoS Notification*, we will update one of the applications to receive incompatible QoS notifications in addition to Data Available notifications.

Not all Quality of Service policies have Request-Offered semantics. For example, the History QoS Policy that we will discuss below is not request-offered: *DataWriters* and *DataReaders* can have their own history settings, independent of each other; therefore, their History QoS policies do not need to match. You can check which QoS policies do and do not have request-offered semantics by looking at the RxO column in the QoS Reference Guide.

5.2 Some Basic QoS Policies

5.2.1 Reliability and History QoS Policies

The Reliability QoS Policy and History QoS Policy work together to determine how reliably data gets sent.

"Best Effort" Reliability

In *Data Types*, we introduced one data flow pattern where the QoS setting isn't Reliable, called "Streaming Sensor Data." Remember that Streaming Sensor Data has these characteristics:

- Usually sent rapidly
- Usually sent periodically
- When data is lost over the network, it is more important to get the next update than to wait for retransmission of the lost update

Because of these characteristics, streaming sensor data is generally not configured to be reliable. It is configured with "Best Effort" reliability. It is one end of the spectrum of how reliably your data can be sent.

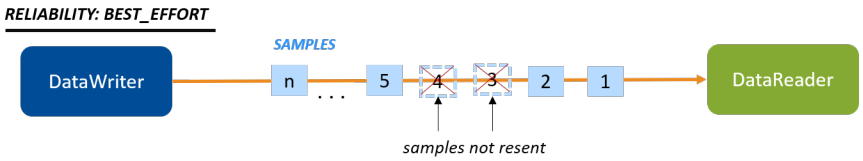


Figure 5.3: Best-Effort Reliability: Samples that the *DataReader* did not receive are not resent.

Table 5.1: Reliability QoS Policy

Reliability (RxO)	Level (Lowest to Highest)	Definition
kind	BEST_EFFORT	Do not send data reliably. If samples are lost, they are not resent.
	RELIABLE	Send data reliably. Resend samples lost on the network, depending on the History QoS Policy and Resource Limits QoS Policy settings.

"Reliable" Reliability + "Keep All" History

The other end of the spectrum is a pattern called "Event and Alarm Data." The typical characteristics of event and alarm data are:

- May be sent rapidly
- Sent intermittently
- Important to see every update for events and alarms that occur

Event and Alarm Data requires a level of reliability where all data is kept for reliable *DataReaders*. This means that *Connext* will try to resend all data not received by existing *DataReaders*, and it will maintain a queue of data that has not been delivered to the *DataReaders*. It also means that a *DataWriter* will not overwrite data in its queue until all *DataReaders* have acknowledged they received the previously sent data (or have gone offline). This level of reliability is set up using:

- Reliability **kind** = RELIABLE
- History **kind** = KEEP_ALL

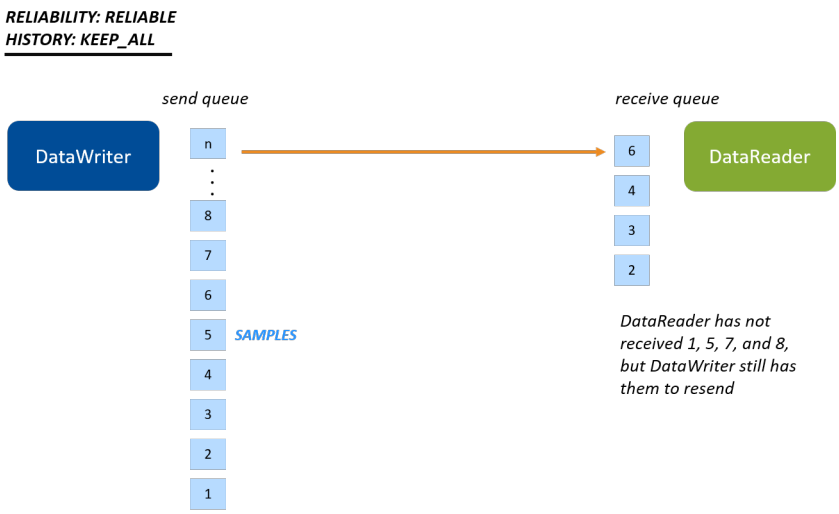


Figure 5.4: Keep-All Reliability: Now there is a queue, and all samples are kept in the queue. Samples cannot be overwritten until they are received. (More details and caveats are explained in Reliability Models for Sending Data, in the RTI Connext Core Libraries User's Manual.)

"Reliable" Reliability + "Keep Last" History

There is one more reliability configuration in-between "Best Effort" and "Keep-All" called Keep-Last Reliability.

Keep-Last Reliability is a configuration where the last N number of samples sent are reliably delivered. This allows the *DataWriter* to overwrite older samples with newer samples, even if some *DataReaders* have not received those older samples. This configuration is set up using:

- Reliability **kind** = RELIABLE

- History **kind** = KEEP_LAST
- History **depth** = Number of samples to keep for each instance

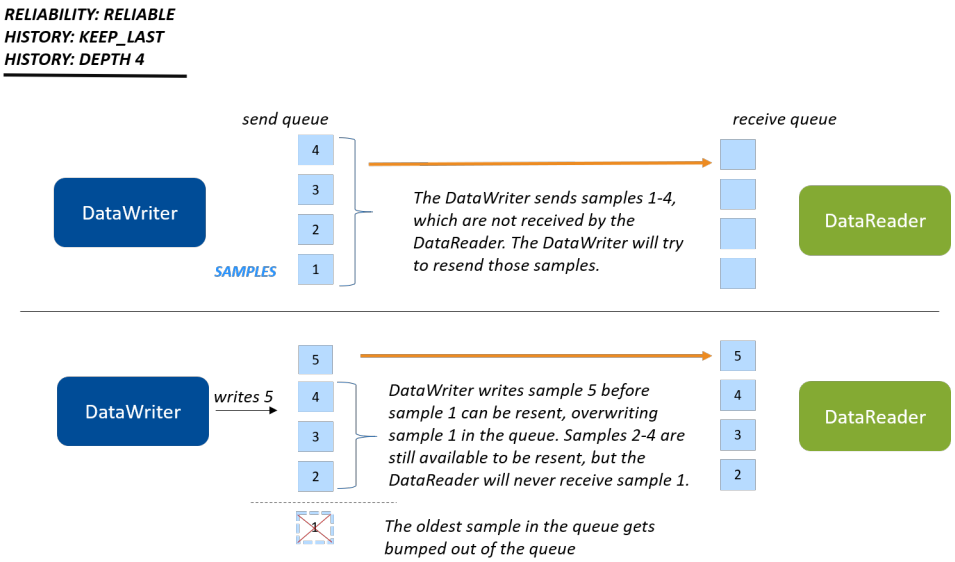


Figure 5.5: Keep-Last Reliability (unkeyed or with a single instance): In this example, you have four slots open on the writer side to keep samples that have not yet been received by the reader.

Imagine that the *DataWriter* in Figure 5.5 rapidly writes 10 samples. Four of those samples will overwrite the four samples that are kept in the queue; the next four will overwrite those four, and so on. The *DataReader* might accept those samples as rapidly as the *DataWriter* writes them—if not, some of them might be lost.

There is one important feature of how the History **depth** works that makes it important for many design patterns: the History **depth** QoS setting is applied *per instance*. This means that you specify a History **depth** of N samples, and *Connext* will reliably deliver the last N samples *of each instance* (for example, of each flight).

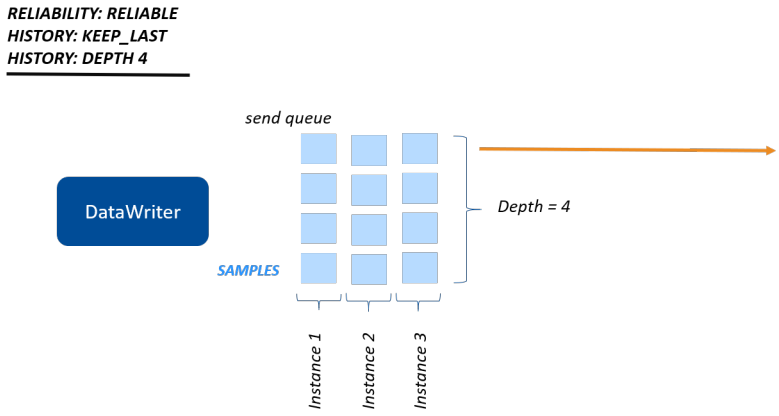


Figure 5.6: Keep-Last Reliability (with multiple instances): A history of four samples **per instance** are available to be reliably delivered to a *DataReader*.

We've been focused on the *DataWriter*'s queue when talking about Keep-Last Reliability, but the behavior is the same for the *DataReader*: the *DataReader* queue keeps History **depth** samples for each instance, and when a new sample arrives, it is allowed to overwrite an existing sample in the *DataReader*'s queue for that instance.

On the *DataWriter* side, the History **depth** controls how many samples to keep around until all matching *DataReaders* have fully acknowledged the samples. There is another depth, **writer_depth**, in the Durability QoS Policy, that controls what subset of the historical samples to send to *DataReaders* that come late to the system. We'll learn more about that in *Durability QoS Policy*.

Table 5.2: History QoS Policy

History (Not RxO)	Value	Definition
kind	KEEP_LAST	Keep the last **depth** number of samples per instance in the queue until they are reliably delivered
	KEEP_ALL	Keep all samples (subject to Resource Limits that we will discuss next) until they are reliably delivered
depth	<integer value>	How many samples to keep per instance for reliable delivery if Keep Last is specified

Summary

You have a range of reliability options using the Reliability and History QoS policies, for various data patterns:

- Streaming data like "ChocolateTemperature" that does not need reliability at all
- State data like "ChocolateLotState", where *DataReaders* generally want to reliably receive the latest state and can accept missing some state updates when the state is changing rapidly
- Event and Alarm data that needs guaranteed delivery of every sample

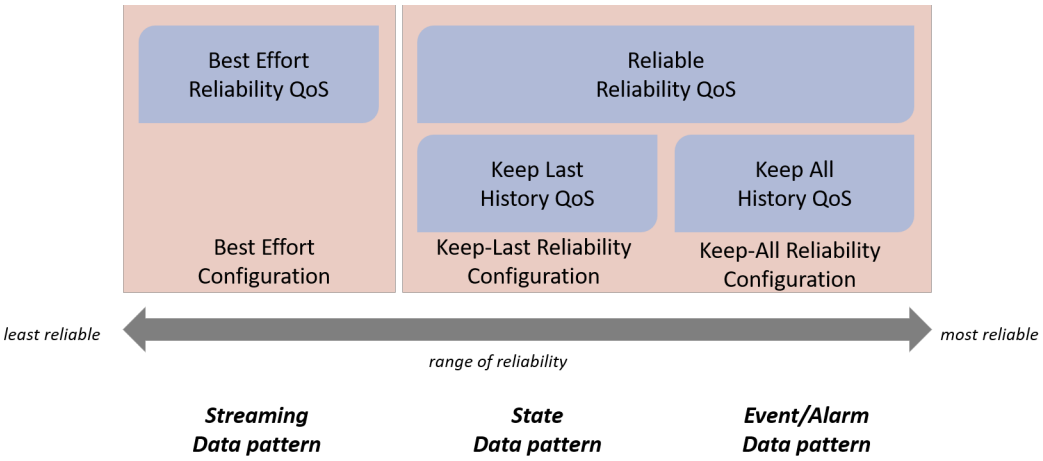


Figure 5.7: A range of reliability options between Best Effort and Reliable. If a RELIABLE **kind** is selected, the History QoS Policy comes into play.

5.2.2 Resource Limits QoS Policy

Even in a system where you need strict Keep-All Reliability, there may be a limit to the number of samples that you want to keep in a *DataWriter*'s or *DataReader*'s queue at one time, because of memory resource constraints. You may want to set a maximum number of samples allowed in a *DataWriter* or *DataReader* queue so that it does not grow indefinitely.

The Resource Limits QoS Policy contains several limits, but the one we will focus on is **max_samples**. This setting limits the total number of samples in a *DataWriter*'s or *DataReader*'s queue across all instances.

If a *DataWriter* or *DataReader* has its History **kind** set to KEEP_ALL, it is not allowed to overwrite samples in its queue to make room for new samples. So what happens if a *DataWriter* or *DataReader* exceeds its **max_samples** resource limit? The *DataWriter* and *DataReader* handle this situation slightly differently.

DataWriters with Keep-All Reliability handle hitting their Resource Limits by blocking the `write()` call, waiting for an empty slot in the queue as the *DataReaders* receive the reliable data. *DataReaders* with Keep-All Reliability handle hitting their Resource Limits by rejecting any samples that arrive when their queue is full, and notifying the application so that the application can call `take()` to remove samples from the queue.

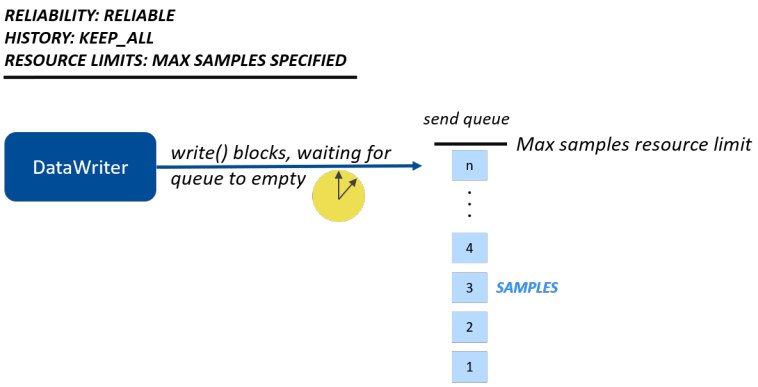


Figure 5.8: *DataWriter* View: The *DataWriter*'s `write()` call will block if it hits its resource limits, and one or more *DataReaders* have not received all the data. If the `write()` call times out without being able to send data, `write()` will return or throw an error. If you're interested in the details of the reliability protocol and how *DataWriters* handle non-responsive *DataReaders* without becoming blocked forever, see Reliability Models for Sending Data, in the RTI Connext DDS Core Libraries User's Manual.

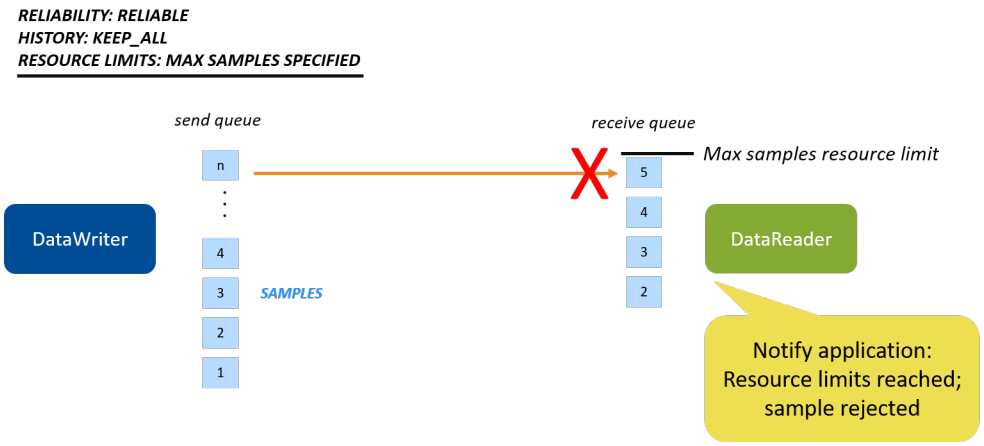


Figure 5.9: *DataReader* view: The *DataReader* rejects samples if it has reached its resource limits.

Table 5.3: Resource Limits QoS Policy

Resource Limits (Not RxO)	Value	Definition
max_samples	<integer value>	The maximum samples allowed in a *DataWriter*'s or *DataReader*'s queue across all instances
max_instances	<integer value>	The maximum number of instances a *DataWriter* can write or a *DataReader* can read
max_samples_per_instance	<integer value>	The maximum number of samples allowed for each instance in a *DataWriter*'s or *DataReader*'s queue

We've covered only a few resource limits here. For more information, see RESOURCE_LIMITS QosPolicy, in the RTI Connext Core Libraries User's Manual.

5.2.3 Durability QoS Policy

The Durability QoS Policy specifies whether data will be delivered to a *DataReader* that was not known to the *DataWriter* at the time the data was written (also called a "late-joining" *DataReader*). Perhaps the *DataReader* wasn't there or hadn't been discovered at the time the samples were written. This QoS policy is used in multiple patterns, including the State Data and Last-Value Cache patterns.

One example of State Data is the ChocolateLotState data in the chocolate factory. Recall that the applications in our example update the state of the lot, and the Monitoring/Control application monitors that state.

The typical characteristics of state data are:

- Typically state data does not change rapidly and periodically (otherwise, it would be streaming data).
- Applications monitoring state data would like to reliably receive that data.
- Applications monitoring state data need to know the current state. It is more important to receive the current state than to receive every state update that has ever happened, even if that means missing some state updates.

These requirements of the State Data pattern can be met by setting History **kind** to KEEP_LAST, Durability **kind** to TRANSIENT_LOCAL or higher, and Durability **writer_depth** to the number of samples you want delivered to late-joining *DataReaders*.

Typically when an application needs state data, it wants to receive the current states as soon as it starts up. The one thing we've been missing in our example is that if an application with a *DataReader* starts up late, right now it doesn't find out the current state of the chocolate lots in the system. This is especially obvious if you start the Monitoring/Control application before you start the Tempering Station application:

```
$ dotnet run --project TemperingApplication
waiting for lot
Processing lot #3
waiting for lot
waiting for lot
waiting for lot
Processing lot #4
```

The Monitoring/Control application has sent lots #0-4 to the Tempering application—but the first thing the Tempering application sees is lot #3! It lost the notifications about all the previous lots. We will fix this in one of the hands-on exercises later in this module, by using the Durability QoS Policy together with the Reliability and History QoS policies. We'll use a QoS profile that will specify a higher Durability level than the default, so the Tempering application's *DataReader* receives ChocolateLotState updates that were written before it started.

The lowest level of durability is "Volatile," which means that historical data is not sent to any late-joining *DataReader*. ("Historical" data is any data sent by a *DataWriter* before it discovers a *DataReader*.) "Volatile" is the default durability setting. The next level of durability is "Transient Local," which means that historical data is automatically maintained in the *DataWriter*'s queue, up to the **writer_depth**.

By default, the Durability **writer_depth** is the same as the History **depth**, but you can set it to be a subset of the History **depth**, as shown in Figure 5.10.

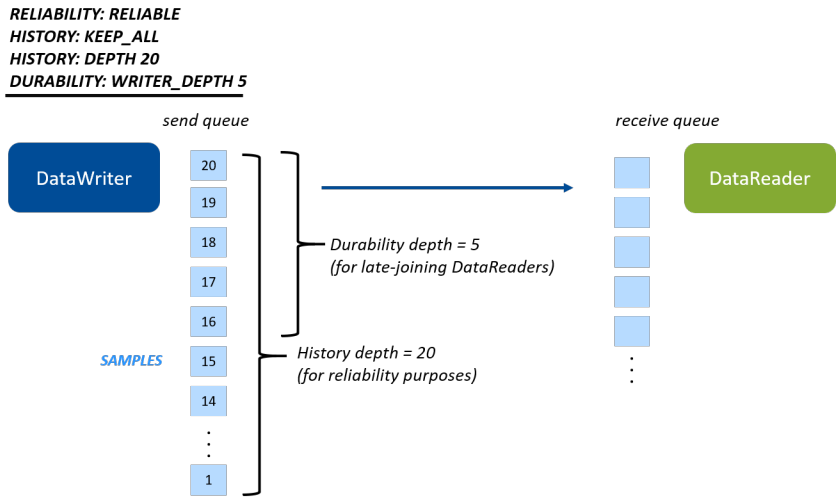


Figure 5.10: History **depth** determines how many samples to keep for reliability purposes (for example, for redelivering to *DataReaders* that haven't acknowledged them yet). Durability **writer_depth** determines what subset of the History **depth** samples to deliver to late-joining *DataReaders*.

Late-joining *DataReaders* that also use reliability and Transient Local durability are automatically sent historical data, up to the Durability **writer_depth**, when they discover the *DataWriter*. "Transient Local" is the setting we will use in the hands-on exercise later in this module. This setting will ensure that the Tempering application receives notifications about all previous lots when it starts up late.

Typically, a **writer_depth** of 1 is used in the State Data pattern:

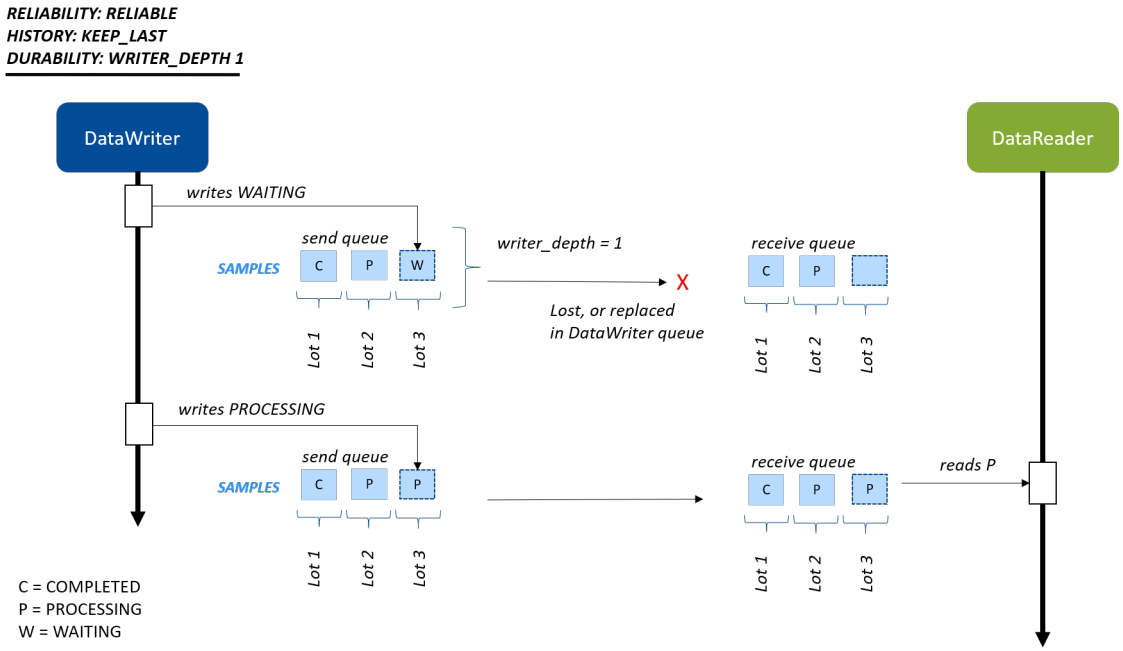


Figure 5.11: State Data pattern using reliability with a writer depth of 1. The late-joining *DataReader* doesn't need to know all or several previous states, just the most recent state.

In Figure 5.11, the *DataWriter* is updating the ChocolateLotState only when that state changes. In the real world, this may happen infrequently, depending on how long it takes the station to process a lot. If the ChocolateLotState *DataWriter* was using Best Effort and the *DataReader* missed an update, the *DataWriter* would not necessarily send new data right away—so the *DataReader* might not receive the state of the lot for a long time. With a Reliability **kind** of RELIABLE and a Durability **writer_depth** of 1 (as well as a TRANSIENT_LOCAL **kind** or higher *Durability QoS Policy*), the *DataReader* has at least the last available state.

It may not be obvious why this is so powerful at first, but this is the basis for the State Data pattern and we'll see it at work in the last hands-on exercise in this module.

The Durability QoS Policy is a Request-Offered policy. For example, if the *DataReader* requests "Transient Local" durability, but the *DataWriter* is set to "Volatile" durability, the entities are not compatible and they will not communicate.

Table 5.4: Durability QoS Policy

Durability (RxO)	Value (Lowest to Highest)	Definition
kind	VOLATILE	Do not save or deliver historical DDS samples. (Historical samples are samples that were written before the *DataReader* was discovered by the *DataWriter*.)
	TRANSIENT_LOCAL	Save and deliver historical DDS samples if the *DataWriter* still exists.
	TRANSIENT_DURABILITY	Save and deliver historical DDS samples using *RTI Persistence Service* to store samples in volatile memory.
	PERSISTENT_DURABILITY	Save and deliver historical DDS samples using *RTI Persistence Service* to store samples in non-volatile memory.
writer_depth	<integer value>	How many samples are stored by the *DataWriter* application for sending to late-joining *DataReaders* (*DataReaders* that are found after the DDS samples were initially written). Must be <= to the History **depth**.

As you can see in the table above, Durability has two additional **kinds**: TRANSIENT and PERSISTENT. These levels of durability allow historical data to be available to late-joining *DataReaders* even if the original *DataWriter* is no longer in the system (because it has been shut down, for example). These higher levels of durability require that data is also stored by *RTI Persistence Service*. These QoS settings are sometimes used as part of the Event and Alarm Data pattern in systems where you need to guarantee delivery of Events even if the *DataWriter* no longer exists. They might also be used in a State Data pattern where state synchronization is important: for example, a *DataWriter* changes the state of a system (writes the result of some process) and then ends, and the next application must read that state in order to continue working with it.

Tip: To summarize:

- The History QoS Policy controls how many samples to keep for reliable delivery.
- The Resource Limits QoS Policy limits the total resources that can be used for samples in the *DataWriter* and *DataReader* queues.
- The Durability QoS Policy determines whether, and how many, of the most recent samples are delivered to late-joining *DataReaders*.

5.2.4 Deadline QoS Policy

The Deadline QoS Policy is used for periodic and streaming data, and can be configured on both the *DataWriter* and the *DataReader*.

When the Deadline QoS Policy is set on a *DataReader*, it specifies that the *DataReader* expects to receive data within a particular time period, or else the application should be notified. When the Deadline QoS Policy is set on a *DataWriter*, it specifies that the *DataWriter* commits to writing data within a particular time period, or else the application will be notified. For example, if an application hangs instead of writing within the deadline period, *Connext* will notify that application that it's not fulfilling the quality of service it offered. We will look at how to receive these notifications in *Hands-On 3: Incompatible QoS Notification*.

The Deadline QoS Policy is a request-offered QoS: the *DataWriter* must offer the same or shorter deadline than what the *DataReader* requests. Typically, the *DataWriter* is configured with a shorter deadline than the *DataReader*, due to the possibility of network latency.

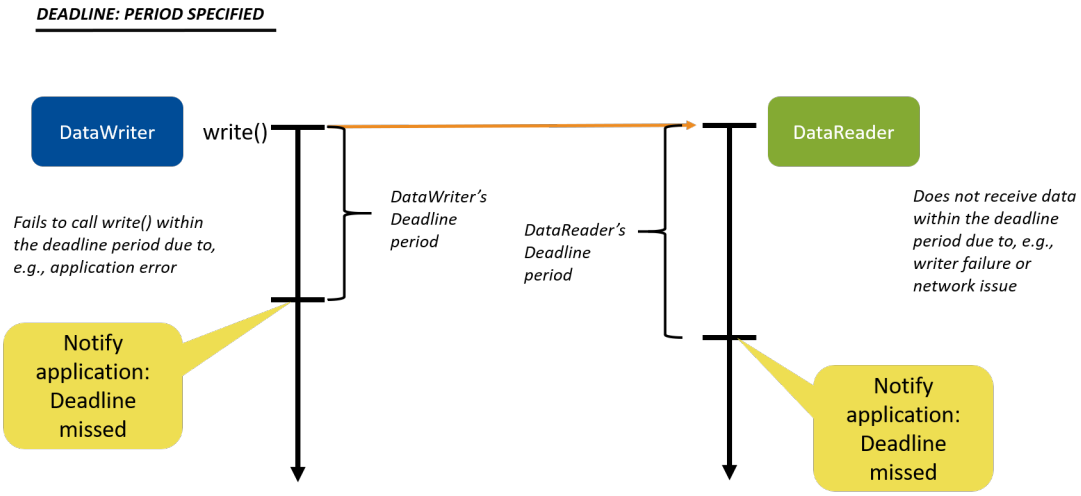


Figure 5.12: Deadline QoS Policy: The *DataWriter* commits to writing data within a particular time period; the *DataReader* expects to receive data within a particular time period.

Table 5.5: Deadline QoS Policy

Deadline (RxO)	Value	Definition
period	deadline time: *DataWriter* deadline must be <= *DataReader* request	Time period in which a *DataWriter* offers to write and a *DataReader* requests to receive periodic samples

5.2.5 QoS Patterns Review

The design patterns you will use for your data are made up of combinations of QoS settings. Here is a review of the patterns we have covered:

Table 5.6: QoS Patterns

Pattern	Reliability kind	History	Durability	Deadline period
Streaming (Periodic) Data	BEST_EFFORT	• **kind**: N/A • **depth**: N/A	• **kind**: VOLATILE • **writer_depth**: N/A	Period in which you expect to send or receive periodic data
Event/Alarm Data	RELIABLE	• **kind**: KEEP_ALL • **depth**: N/A	• **kind**: Various (see below) • **writer_depth**: Number of samples to keep in the queue to be delivered to late-joining *DataReaders*	N/A
State Data	RELIABLE	• **kind**: KEEP_LAST • **depth**: Number of samples to keep in the queue to be reliably delivered	• **kind**: TRANSIENT_LOCAL • **writer_depth**: Number of samples to keep in the queue to be delivered to late-joining *DataReaders*. Typically 1	N/A

Event and Alarm Data may have various levels of Durability **kind**, depending on whether your system design requires Events and Alarms to be available to late-joining *DataReaders*, and whether Events and Alarms must be available even if the original *DataWriter* is no longer in the system.

The Resource Limits QoS Policy that we have discussed is mostly orthogonal to our design patterns: it is ultimately a limit on the maximum memory for samples and instances that your system should use, and that is part of your overall system design. Typically, you design your data flows first, such as Streaming (Perioidic) Data, and set your QoS policies based on that data pattern. Then you refine your QoS settings, overriding individual settings such as resource limits, to make your QoS configuration work for your system.

5.3 QoS Profiles

QoS profiles are groupings of QoS settings defined in an XML file. *Connext* provides many builtin QoS profiles that you can use as a starting point for your systems. Some of these profiles cover the patterns we have just discussed, such as the "Pattern.Streaming" profile we have used before. The pattern-based profiles start with "Pattern" in the name. There are also many that are made up of basic sets of QoS configurations that you can use to build your own patterns. For example, there are QoS profiles based on data flow characteristics such as "Generic.StrictReliable.HighThroughput," which is configured for high-throughput, strictly-reliable data. All of the available Builtin QoS profiles can be viewed in the file `<NDDSHOME>/resource/xml/BuiltinProfiles.documentationONLY.xml`.

Note: As its name implies, the `BuiltinProfiles.documentationONLY.xml` file is only for viewing what the builtin profiles contain. You cannot modify the builtin profiles; however, you can create your own profile that inherits from a builtin profile. Then you can overwrite parts of the inherited profile with your own QoS settings.

In *Modify for Streaming Data* (in *Data Types*), we already changed a QoS profile in the XML to inherit from a builtin QoS profile. In that exercise, though, our code loaded a profile implicitly because `is_default_qos` was set to true in the XML file. This is dangerous, because the default QoS profile may not have all of the settings you want (and some settings you *don't* want). In the next hands-on exercises, we'll review new QoS profiles that inherit from builtin profiles and override some values. We will change the code to load those QoS profiles, allowing our application to specify the profiles it wants to use for individual *DomainParticipants*, *DataWriters*, or *DataReaders*.

Note: Using `is_default_qos="true"` is not a best practice in production applications. It's a convenient way to get you started quickly, but in production applications you should explicitly specify which QoS profile to use, instead of relying on a default.

5.4 Hands-On 1: Update One QoS Profile in the Monitoring/Control Application

In this exercise, you are starting with applications in the repository where we have pre-configured only some of the QoS profiles. We will start by configuring some of the correct QoS profiles for the Monitoring/Control application, but we will intentionally leave one *DataWriter* configured to use the default QoS profile in error. This will allow us to debug incompatible QoS profiles.

We will do the following:

- Hands-On 1:
  - Load the QoS file.
  - Set up a *DomainParticipant* QoS profile, to better help you debug QoS profiles in *Admin Console*. (For more information on *DomainParticipants*, see *Publishers, Subscribers, and DomainParticipants*.)

  - Set up the correct QoS profiles in the *DataReaders*, but not the *DataWriter*.

- Hands-On 2: Debug the incompatible QoS profiles in *Admin Console*.
- Hands-On 3: Add code to help debug incompatible QoS profiles directly in one of the applications.
- Hands-On 4: Change the Monitoring/Control *DataWriter*'s QoS profile to the correct QoS profile, so now the *DataWriters* and *DataReaders* are compatible.

In this exercise, you'll be working in the directory `5_basic_qos/csharp`, which was created when you cloned the getting started repository from GitHub in the first module, in *Clone Repository*.

1. Run *RTI Code Generator* (*rtiddsgen*) to generate the C# code for the IDL file `chocolate_factory.idl` as explained in *Build the Applications*.

2. Within the application, load the `qos_profiles.xml` file from the repository by performing the following steps:

   Open the `MonitoringCtrlApplication.cs` file and look for the comment:

   ```
   // Exercise #1.1: Add QoS provider
   ```

   Add the following code after the comment:

   ```
   // Exercise #1.1: Add QoS provider
   // Loads the QoS from the qos_profiles.xml file.
   var qosProvider = new QosProvider("./qos_profiles.xml");
   ```

   This code explicitly loads an XML QoS file named `qos_profiles.xml` that we have provided for you in the `csharp` directory, instead of relying on the file with a default name (`USER_QOS_PROFILES.xml`) as we did in Section 3 (*Modify for Streaming Data*).

   Tip: QoS profile XML files can be modified and loaded without recompiling the application.

   ---

3. Review the QoS profiles in the `qos_profiles.xml` file.

   This XML file contains the QoS profiles that will be used by your applications. The file defines a QoS library, "ChocolateFactoryLibrary," that contains the QoS profiles we will be using:

   ```
   <qos_library name="ChocolateFactoryLibrary">
   ```

   Take a look at the "TemperingApplication" and "MonitoringControlApplication" qos_profiles in the XML file:

   ```
   <!-- QoS profile to set the participant name for debugging -->
   <qos_profile name="TemperingApplication"
                base_name="BuiltinQosLib::Generic.Common">
       <domain_participant_qos>
           <participant_name>
               <name>TemperingAppParticipant</name>
           </participant_name>
       </domain_participant_qos>
   </qos_profile>
   ```

```
<!-- QoS profile to set the participant name for debugging -->
<qos_profile name="MonitoringControlApplication"
             base_name="BuiltinQosLib::Generic.Common">
    <domain_participant_qos>
        <participant_name>
            <name>MonitoringControlParticipant</name>
        </participant_name>
    </domain_participant_qos>
</qos_profile>
```

We haven't talked about setting QoS profiles on the *DomainParticipants* yet–we will say more about this when we talk about discovery. For now, what's important is that if you set the `participant_name` QoS setting, *Connext* makes that name visible to other applications. Since *DomainParticipants* own all your *DataWriters* and *DataReaders*, giving your *DomainParticipants* unique names makes it easier to tell your *DataWriters* and *DataReaders* apart when you are debugging in *Admin Console*.

These *DomainParticipant* profiles are pretty simple. They inherit from the builtin QoS profile "Generic.Common" by specifying that as the base profile:

```
base_name="BuiltinQosLib::Generic.Common"
```

Then these QoS profiles override the default *DomainParticipant*'s name:

```
<domain_participant_qos>
    <participant_name>
        <name>MonitoringControlParticipant</name>
    </participant_name>
</domain_participant_qos>
```

In the next step, we will change the Monitoring/Control Application's *DomainParticipant* to use this QoS profile to help us debug in *Admin Console*.

Now review the "ChocolateTemperatureProfile" Qos profile, which will be used to specify QoS policies for *DataWriters* and *DataReaders* in our applications:

```
<qos_profile name="ChocolateTemperatureProfile"
 base_name="BuiltinQosLib::Pattern.Streaming">
```

This profile inherits from the "Pattern.Streaming" profile, which provides a basic pattern for *DataWriter* and *DataReader* QoS profiles for streaming data, including the following:

- Reliability **kind** = BEST_EFFORT
- Deadline **period** = 4 seconds

Remember you can review the details of the default profiles like "Pattern.Streaming" in `<NDDSHOME>/resource/xml/BuiltinProfiles.documentationONLY.xml`.

Review the "ChocolateLotStateProfile" QoS profile, which will be used to specify QoS policies for *DataWriters* and *DataReaders* in our applications:

```
<qos_profile name="ChocolateLotStateProfile"
                    base_name="BuiltinQosLib::Pattern.Status">
```

This profile inherits from the "Pattern.Status" profile, which provides the following *DataWriter* and *DataReader* QoS settings:

- Reliability **kind** = RELIABLE
- Durability **kind** = TRANSIENT_LOCAL
- History **kind** = KEEP_LAST
- History **depth** = 1
- Durability **writer_depth** = AUTO_WRITER_DEPTH (i.e., inherited from the History **depth**, which is 1 in this example; AUTO_WRITER_DEPTH is the default setting for **writer_depth**)

This is the State Data pattern described above, in *"Reliable" Reliability + "Keep Last" History*.

4. Add the *DomainParticipant*'s QoS profile in `MonitoringCtrlApplication.cs`.

   Look for the comment:

   ```
   // Exercise #1.2: Load DomainParticipant QoS profile
   ```

   Replace this line:

   ```
   DomainParticipant participant = DomainParticipantFactory.Instance
       .CreateParticipant(domainId);
   ```

   So the code looks like this:

   ```
   // Exercise #1.2: Load DomainParticipant QoS profile
   var participantQos = qosProvider.GetDomainParticipantQos(
       "ChocolateFactoryLibrary::MonitoringControlApplication");
   DomainParticipant participant = DomainParticipantFactory.Instance
       .CreateParticipant(domainId, participantQos);
   ```

   With this code, you have just explicitly loaded the "MonitoringControlApplication" QoS profile, which is part of the ChocolateFactoryLibrary. Remember that this QoS profile inherits from the default *DomainParticipant* QoS profile, but specifies the `participant_name` to be "MonitoringControl-Participant" in the QoS file.

5. Add the *DataReaders* QoS profile in `MonitoringCtrlApplication.cs`.

   Look for the comment:

   ```
   // Exercise #1.3: Update the lotStateReader and temperatureReader
   // to use correct QoS
   ```

   Replace these lines:

   ```
   DataReader<ChocolateLotState> lotStateReader =
         subscriber.CreateDataReader(lotStateTopic);
   ```

```
// Add a DataReader for Temperature to this application
DataReader<Temperature> temperatureReader =
      subscriber.CreateDataReader(temperatureTopic);
```

So the code looks like this:

```
// Exercise #1.3: Update the lot_state_reader and temperature_reader
// to use correct QoS
var readerQos = qosProvider.GetDataReaderQos(
    "ChocolateFactoryLibrary::ChocolateLotStateProfile");
DataReader<ChocolateLotState> lotStateReader =
    subscriber.CreateDataReader(lotStateTopic, readerQos);

// Add a DataReader for Temperature to this application
readerQos = qosProvider.GetDataReaderQos(
    "ChocolateFactoryLibrary::ChocolateTemperatureProfile");
DataReader<Temperature> temperatureReader =
    subscriber.CreateDataReader(temperatureTopic, readerQos);
```

Now that you have added that code, the *DataReaders* are each loading the correct QoS profiles. You have not changed the *DataWriter*, so it is erroneously using the default QoS profile.

6. Run the Monitoring/Control application:

```
$ dotnet run --project MonitoringCtrlApplication
```

In another command prompt window, run the Tempering application with a sensor ID as a parameter:

```
$ dotnet run --project TemperingApplication -- --sensor-id 9
```

Note: Make sure you run `dotnet` from `5_basic_qos/csharp` since that's where the applications expect to find `qos_profiles.xml`.

---

7. Look at the output of the applications:

The Monitoring/Control application starts lots, but we never see lot updates:

```
Starting lot:
[lot_id: 0 next_station: StationKind::TEMPERING_CONTROLLER ]

Starting lot:
[lot_id: 1 next_station: StationKind::TEMPERING_CONTROLLER ]
```

The Tempering application waits for lots, but it never gets any updates about lots:

```
ChocolateTemperature Sensor with ID: 9 starting
Waiting for lot
Waiting for lot
Waiting for lot
Waiting for lot
Waiting for lot
```

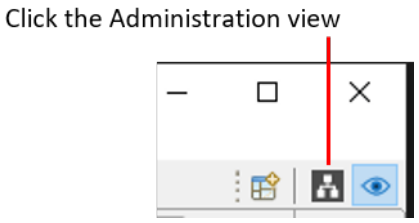
The *DataWriters* and *DataReaders* are not communicating because they have incompatible QoS profiles, which we will debug in the next hands-on exercise.

8. Keep the applications running for the next hands-on exercise.

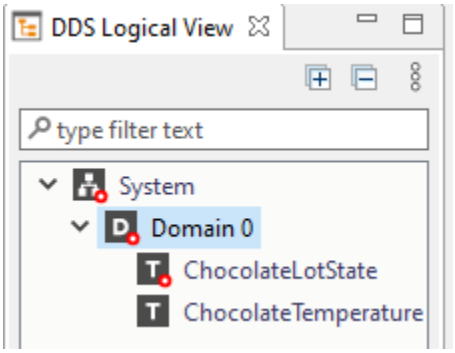
5.5 Hands-On 2: Incompatible QoS in Admin Console

As we mentioned above, the Tempering application is using all correct QoS settings, and we updated the *DataReaders* in the Monitoring/Control application. But we intentionally didn't update the Monitoring/Control application's *DataWriter*. We will debug this problem using *Admin Console*.

1. Make sure the **monitoring_ctrl_application** and **tempering_application** are still running.
2. Open *Admin Console*.
   - Open *Admin Console* from *RTI Launcher*.
   - Choose the Administration view:



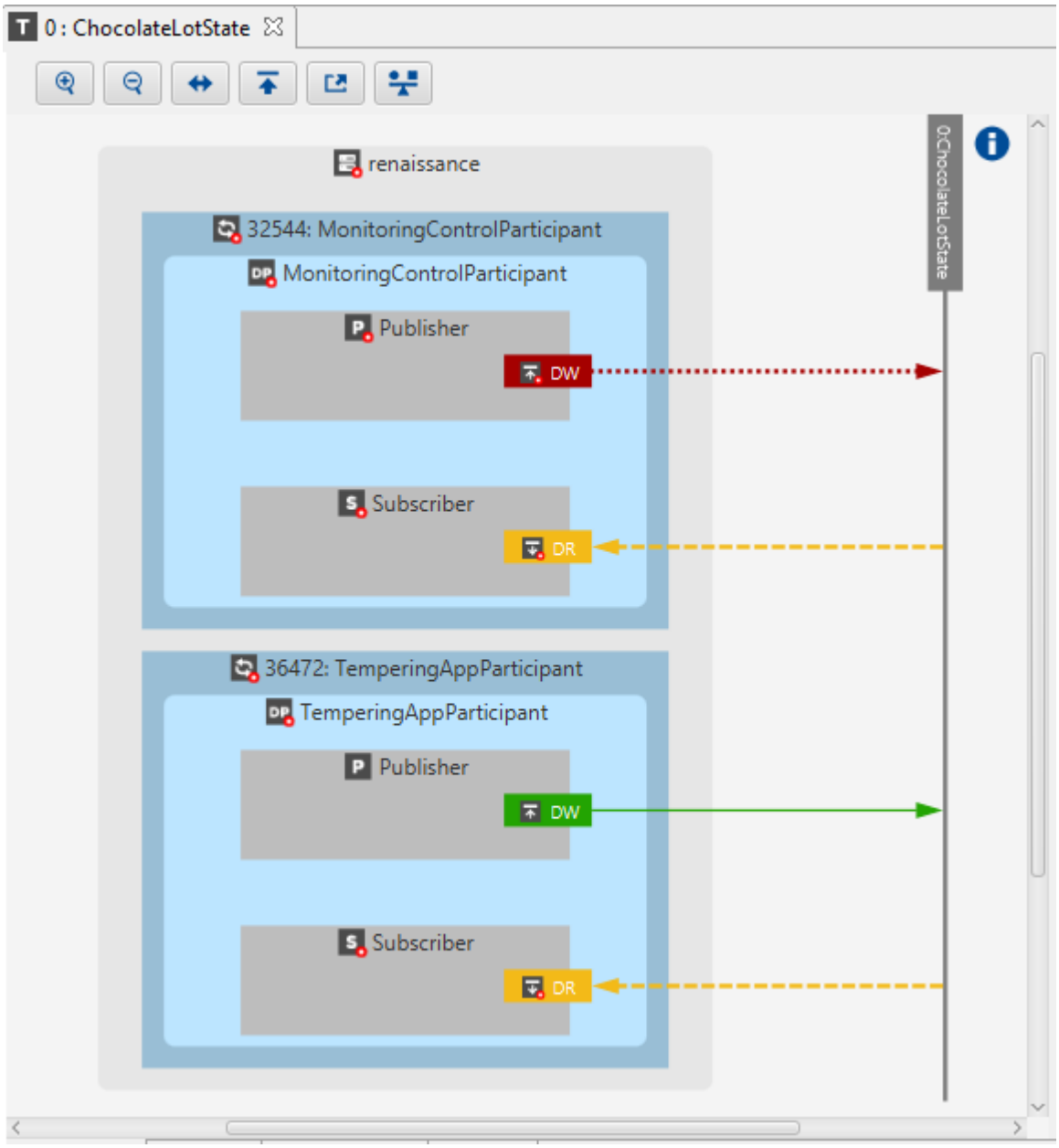
   (The Administration view might be selected for you already, since we used that view in the previous module.)
3. You should immediately notice that there is a red error showing on your "ChocolateLotState" *Topic*.



   If you hover over the ChocolateLotState *Topic*, you will see an error: "Request/offered QoS."
4. Click on the "ChocolateLotState" *Topic* in the DDS Logical View, and you should see a diagram showing the *DataWriters* and *DataReaders* that are writing and reading the *Topic*.

   It's easy to see which application the *DataWriters* and *DataReaders* belong to, because you've set the `participant_name` QoS setting, which allows you to see the names of the *DomainPartici-*

pants (**MonitoringControlParticipant** and **TemperingAppParticipant**) that own the *DataWriters* and *DataReaders* in this system.



In the above view, some *DataWriters* and *DataReaders* are matching, while others are not. The *DataWriter* that is configured correctly is green, the misconfigured one is red. The "ChocolateLotState" *DataReaders* in the system, in yellow, are partially matched: they match with the *DataWriter* in the Tempering application that is configured correctly.

Remember that the "ChocolateLotState" *DataWriters* and *DataReaders* look like Figure 5.13 in our system; the Monitoring/Control application's *DataWriter* is not configured correctly:

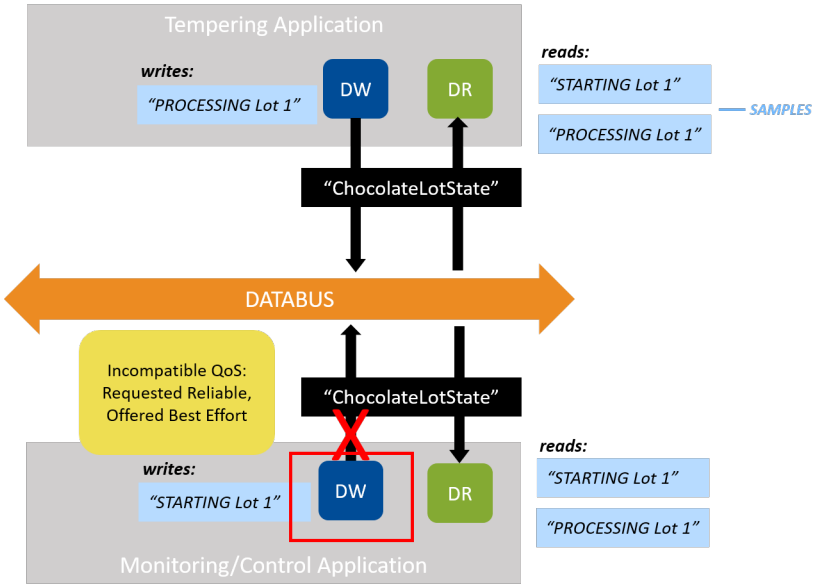
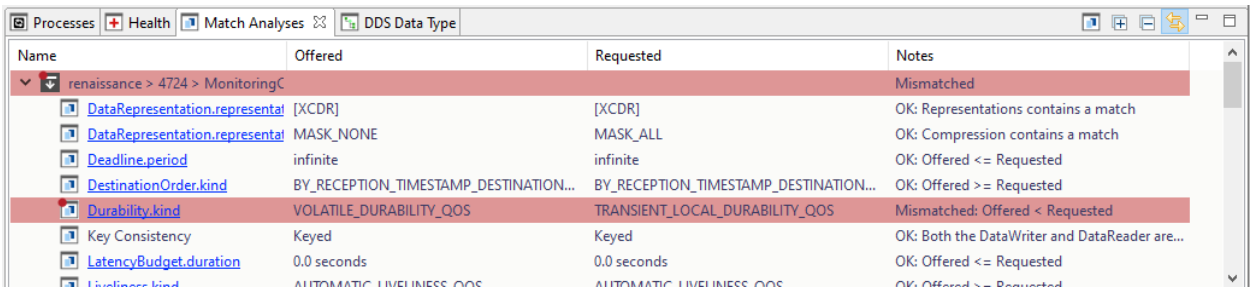


Figure 5.13: Two *DataWriters* and two *DataReaders* should be communicating in our example system, when configured correctly. But the *DataWriter* QoS policy in the Monitoring/Control application is not configured correctly. (See also Figure 4.5.)

5. Click on the red *DataWriter* for the Monitoring/Control application.

   At the bottom of *Admin Console*'s screen, click on the Match Analyses tab:



Notice in the Offered and Requested columns that the misconfigured *DataWriter* is offering Volatile durability (meaning that it will not save data for late-joining *DataReaders*). However, the two *DataReaders* are requesting Transient Local durability (meaning that they need data saved for when they join late—this is the correct setting, since it follows the State Data pattern). The *DataReader* is requesting a higher level of QoS setting ("I need data saved") than the *DataWriter* is offering ("I'm not saving data"). This is a system error, caused by incompatible QoS policies. Scroll down in this view to see the second *DataReader* that is mismatched with the *DataWriter*.

Tip: Any time a *DataReader* requests a higher level of service than the *DataWriter* offers, *Connext* reports it as a system error.

5.6 Hands-On 3: Incompatible QoS Notification

Both your *DataWriters* and *DataReaders* can be notified in your own applications when they discover a *DataReader* or *DataWriter* with incompatible QoS settings. In the `5_basic_qos` directory, we are monitoring the `RequestedIncompatibleQos` *DataReader* event to detect incompatible QoS policies. (This is a change to the Tempering application from the version used in the previous module, in `4_keys_instances`.) In the steps below, you will review this additional code and add new code that prints out the error.

1. Quit both of your applications if you haven't already.
2. Open the `TemperingApplication.cs` file.
3. Review the code. Notice that we have already added code to detect when the *DataReader* discovers a *DataWriter* with an incompatible QoS policy. When this happens, it calls the event handler `OnRequestedIncompatibleQos`:

```
DataReader<ChocolateLotState> lotStateReader = subscriber.
↪CreateDataReader(
    lotStateTopic,
    qos: qosProvider.GetDataReaderQos(
↪"ChocolateFactoryLibrary::ChocolateLotStateProfile"),
    preEnableAction: reader => reader.RequestedIncompatibleQos +=␣
↪OnRequestedIncompatibleQos);
```

   We configure the event handler in a `preEnableAction` to ensure that the event handler is installed before the *DataReader* is enabled and therefore we don't miss any status update.

4. Find the `OnRequestedIncompatibleQos` method, and add code after the comment:

```
// Exercise #3.1 add a message to print when this DataReader discovers an
// incompatible DataWriter
Type incompatiblePolicy = status.LastPolicy;
Console.WriteLine(
      "Discovered DataWriter with incompatible policy: " +␣
↪incompatiblePolicy.Name);
```

   The code you just added will print an error message when the *DataReader* discovers an incompatible *DataWriter*.

5. Rebuild and run both applications.

   When you run the Tempering application, you will now see notifications when it discovers *DataWriters* with incompatible QoS policies:

```
ChocolateTemperature Sensor with ID: 33 starting
waiting for lot
Discovered DataWriter with incompatible policy: Durability
waiting for lot
waiting for lot
```

   (*Connext* prints the error only the first time it notices the incompatibility.)

6. Quit both applications.

5.7 Hands-On 4: Using Correct QoS Profile

Finally, let's go back to the Monitoring/Control application. This is where you added the QoS profile for the *DomainParticipant* and *DataReaders* earlier. Now you will be modifying the *DataWriter*'s QoS profile to use the correct values.

1. Edit `MonitoringCtrlApplication.cs`.

   Find the comment:

   ```
   // Exercise #4.1: Load ChocolateLotState DataWriter QoS profile after
   // debugging incompatible QoS
   ```

   Replace this line:

   ```
   DataWriter<ChocolateLotState> lotStateWriter =
       publisher.CreateDataWriter(lotStateTopic);
   ```

   So the code looks like this:

   ```
   // Exercise #4.1: Load ChocolateLotState DataWriter QoS profile after
   // debugging incompatible QoS
   var writerQos = qosProvider.GetDataWriterQos(
         "ChocolateFactoryLibrary::ChocolateLotStateProfile");
   DataWriter<ChocolateLotState> lotStateWriter =
         publisher.CreateDataWriter(lotStateTopic, writerQos);
   ```

   You have now configured the "ChocolateLotState" *DataWriter* to use the "ChocolateLotStateProfile" QoS profile. Recall that this profile sets a Durability **kind** of TRANSIENT_LOCAL and an effective **writer_depth** of 1, so that late-joining *DataReaders* can get the latest data.

2. Rebuild and run both applications.

   Now you see that they are communicating correctly, and the Tempering application always starts processing lot #0, even if the Tempering application started after the Monitoring/Control application. This is because the Tempering application is using the State Data pattern for its QoS profile: Reliability **kind** = RELIABLE, History **kind** = KEEP_LAST, Durability **kind** = TRANSIENT_LOCAL, Durability **writer_depth** = 1 (inherited from the History **depth** of 1).

   Recall that the Tempering application processes the lot and updates the status of each lot it receives.

   ```
   ChocolateTemperature Sensor with ID: 35 starting
   waiting for lot
   Processing lot #0
   Lot completed
   waiting for lot
   Processing lot #1
   Lot completed
   waiting for lot
   waiting for lot
   Processing lot #2
   Lot completed
   waiting for lot
   ```

Recall that the Monitoring/Control application starts each lot at the TEMPERING CONTROLLER. The TEMPERING CONTROLLER then updates the lot state to indicate it is WAITING, PROCESSING, or completed. (The "current station" is invalid before the tempering controller starts processing the lot, because the Tempering application is the first station, so there is no station before that.)

```
Starting lot:
[lot_id: 0 next_station: StationKind::TEMPERING_CONTROLLER ]
Received Lot Update:
[lot_id: 0, station: StationKind::INVALID_CONTROLLER , next_station:
StationKind::TEMPERING_CONTROLLER , lot_status:
LotStatusKind::WAITING ]
Received Lot Update:
[lot_id: 0, station: StationKind::TEMPERING_CONTROLLER ,
next_station: StationKind::INVALID_CONTROLLER , lot_status:
LotStatusKind::PROCESSING ]
Received Lot Update:
[lot_id: 0 is completed]

Starting lot:
[lot_id: 1 next_station: StationKind::TEMPERING_CONTROLLER ]
Received Lot Update:
[lot_id: 1, station: StationKind::INVALID_CONTROLLER , next_station:
StationKind::TEMPERING_CONTROLLER , lot_status:
LotStatusKind::WAITING ]
Received Lot Update:
[lot_id: 1, station: StationKind::TEMPERING_CONTROLLER ,
next_station: StationKind::INVALID_CONTROLLER , lot_status:
LotStatusKind::PROCESSING ]
Received Lot Update:
[lot_id: 1 is completed]
```

3. If you haven't already, start the Monitoring/Control application before the Tempering application and see that the Tempering application processes all of the lots, starting with 0, even though the Tempering application in this scenario is a late joiner.

   Upon startup, the Tempering application *gets a current view of your entire system*. This is because all of the QoS policies relevant to the State Data pattern are working together: since Reliability **kind** is RELIABLE and Durability **kind** is TRANSIENT_LOCAL, the Tempering application sees all the ChocolateLotState instances even if its *DataReader* is a late-joiner; since Durability **writer_depth** is 1, it sees one sample from each instance. Contrast this with our discussion in *Durability QoS Policy*, where the Monitoring/Control application sent lots #0-4, but the Tempering application saw only lot #3 at the start. This was because we hadn't set Durability yet.

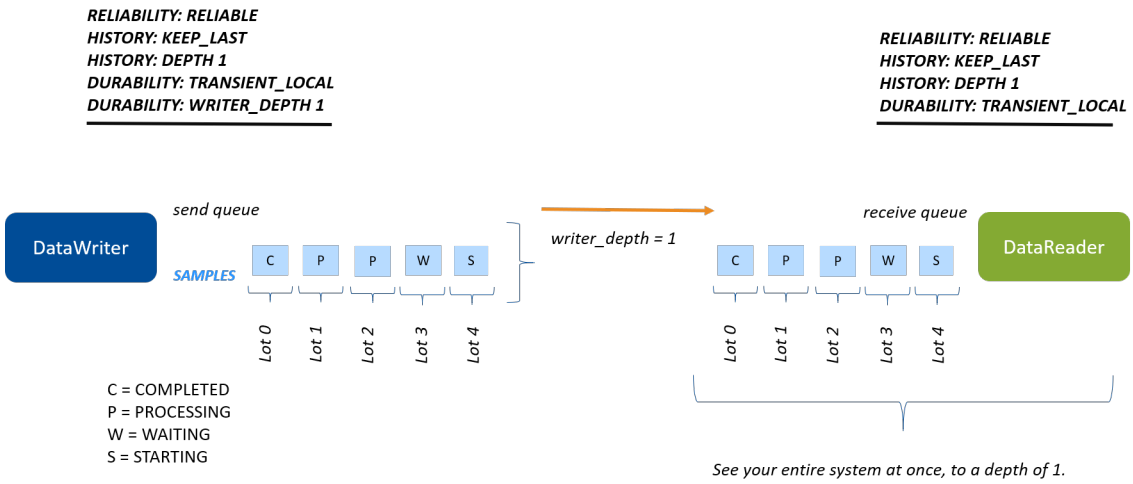


Figure 5.14: Reliability, History, and Durability QoS policies work together so that the late-joining *DataReader* sees one sample per instance.

5.8 Next Steps

Congratulations! You have learned about several of the basic Quality of Service offered by *Connext*. We only scratched the surface of the configuration options you have available, so for further information about QoS policies, you should look at the following documents:

- QoS Reference Guide
- DataWriter QoS, in the RTI Connext Core Libraries User's Manual
- DataReader QoS, in the RTI Connext Core Libraries User's Manual
- Configuring QoS with XML, in the RTI Connext Core Libraries User's Manual

In an upcoming module, we will be looking at filtering data using *ContentFilteredTopics*.

Chapter 6

ContentFilteredTopics

Prerequisites	• *Basic QoS* • Repository cloned from GitHub here
Time to complete	45 minutes
Concepts covered in this module	Content Filtering

In the Publish/Subscribe communication pattern, *DataWriters* send data to *DataReaders* with the same *Topic*. But sometimes a *DataReader* may be interested in only a subset of the data that is being sent. Some examples are:

- A *DataReader* only cares about the temperature when it goes outside of a certain bound.
- A *DataReader* only cares about log messages with levels WARNING or ERROR.
- A *DataReader* only cares about the ChocolateLotState if the *DataReader* belongs to the next station that processes the chocolate lot.

Note that in all of these examples, there might be some *DataReaders* that want to receive a subset of data, and other *DataReaders* that might want to receive all data. For example, in our chocolate factory, an individual station only cares about a chocolate lot if it is the next station to process the lot, but the monitoring application wants to receive every update about every chocolate lot.

Connext offers a mechanism to allow *DataReaders* to filter data so that they only receive the data they are interested in. Applications can do this by creating ContentFilteredTopics instead of normal *Topics*. Since different *DataReaders* may want to receive a different subset of the data, ContentFilteredTopics are specified on the *DataReaders*. No changes are required on *DataWriters* to use content-filters: they continue to use a normal *Topic*, and they can communicate both with *DataReaders* that use a normal *Topic* and *DataReaders* that use a ContentFilteredTopic.

Although ContentFilteredTopics are specified on *DataReaders*, in most cases the *DataWriter* does the filtering, in which case no network bandwidth is used for those filtered samples. (The *DataWriter* finds out during discovery that a *DataReader* has a ContentFilteredTopic.)

Definition

A ContentFilteredTopic is a *Topic* with filtering properties. It makes it possible to subscribe to *Topics* and at the same time specify that you are only interested in a subset of the *Topic's* data.

A ContentFilteredTopic consists of a *Topic*, a filter expression, and optional parameters. The filter expression and parameters can be changed at runtime. You can think of the filter expression as being like the "where" clause in SQL.

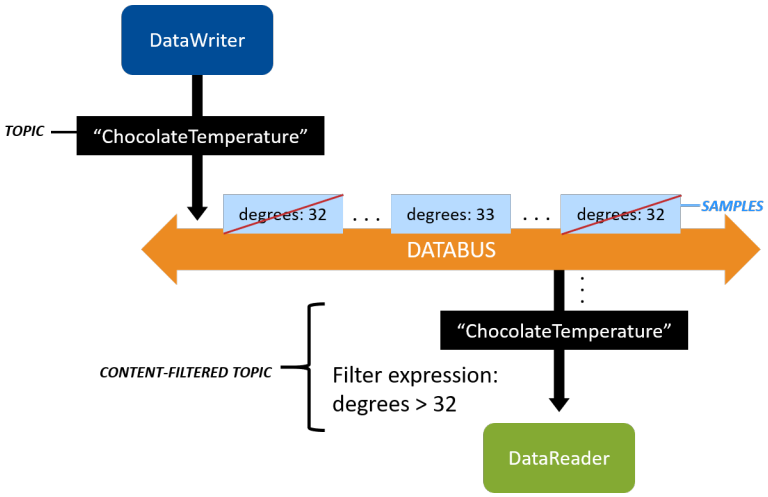


Figure 6.1: A ContentFilteredTopic that only allows samples with degrees > 32.

6.1 The Complete Chocolate Factory

ContentFilteredTopics are the final piece we need to put together the last part of the Chocolate Factory: the ingredient stations.

If you recall, the process the chocolate lot takes through the chocolate factory looks like this:

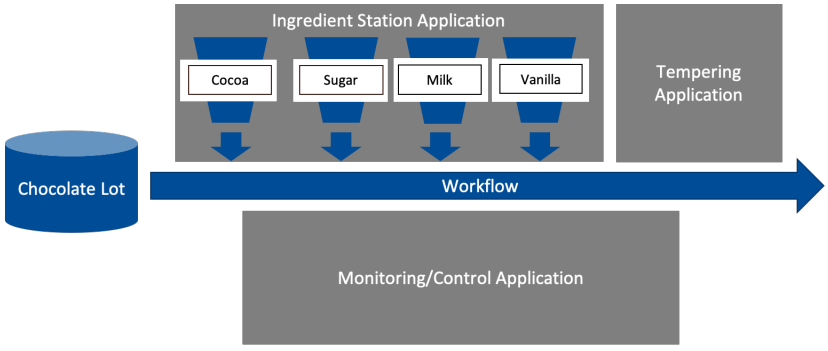


Figure 6.2: *DataReaders* in our system will use content filtering to filter out any ChocolateLotState that doesn't have "next_station" equal to that *DataReader*'s station.

The Monitoring/Control Application starts each lot by writing a sample to the ChocolateLotState *Topic*, with `next_station` set to one of the ingredient stations. Each ingredient station processes the lot by adding an ingredient, and then updates the `next_station`. The last station is the tempering machine, which processes the lot by tempering it, and then calls `dispose()` on the lot instance to say that the lot has completed.

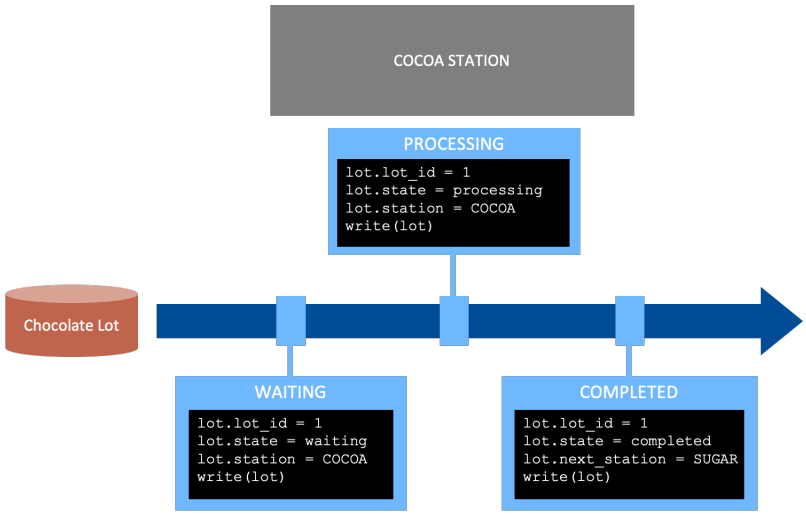


Figure 6.3: A chocolate lot is waiting, processing, or completed by an ingredient station. When a lot is completed, the `next_station` field indicates the next station to process the chocolate lot.

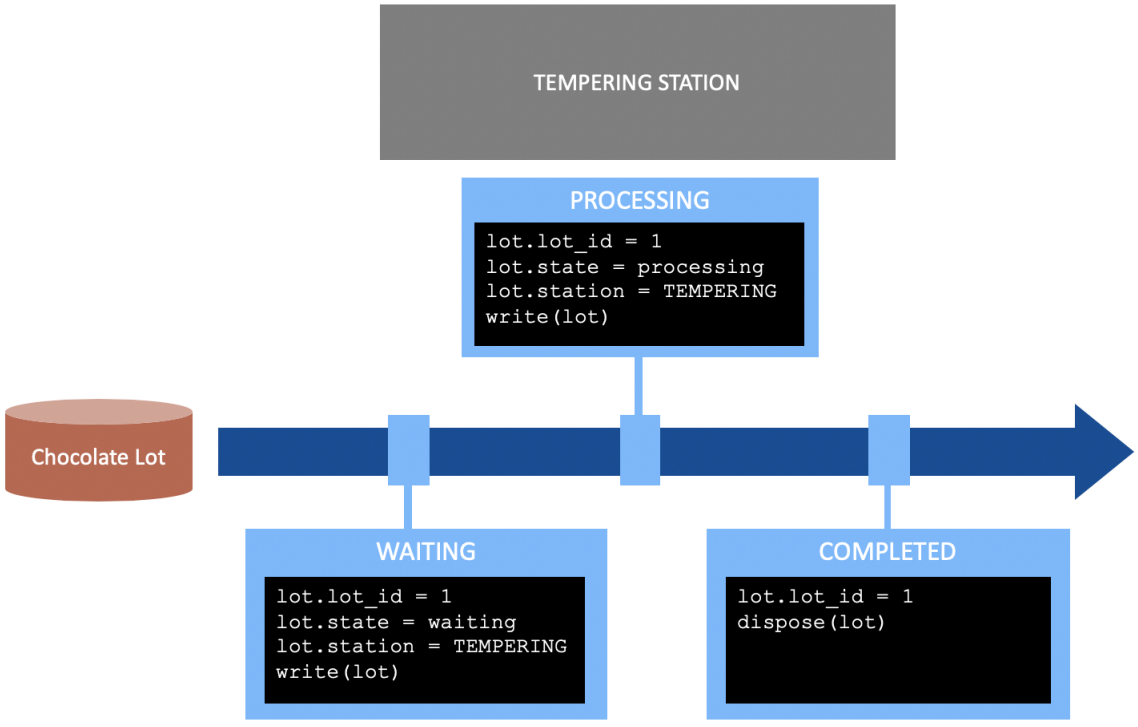


Figure 6.4: A chocolate lot is waiting, processing, or disposed by the tempering application.

The Monitoring/Control Application wants to monitor every state of the chocolate lots as they are processed by the stations. However, an individual station only wants to know about a lot if the `next_station` field indicates it is the station that will process the lot next.

In the previous exercises, we had code in the `ProcessLot` function in `TemperingApplication.cs` to check whether the tempering station was the next station:

```
private void ProcessLot(
    DataReader<ChocolateLotState> lotStateReader,
    DataWriter<ChocolateLotState> lotStateWriter)
{
   using var samples = lotStateReader.Take();
   foreach (var sample in samples.ValidData())
   {
         if (sample.next_station == StationKind.TEMPERING_CONTROLLER)
         {
             ...
         }
   }
}
```

Instead of doing that, we can use a ContentFilteredTopic to filter out all the lot updates where the tempering controller is not the `next_station`. This cleans up the logic when the tempering application receives data, because it no longer needs to check whether a chocolate lot update is intended for it. This can also save network bandwidth because *DataWriters* perform the filtering. When the application uses a ContentFilteredTopic, *Connext* can filter out the data more efficiently than the application, and may not even send updates to an application that is not interested.

6.2 Hands-On 1: Update the ChocolateLotState DataReader with a ContentFilteredTopic

We are going to update the *DataReader* in the Tempering Application to use a ContentFilteredTopic instead of checking whether the data is important to this application every time a *DataWriter* updates the ChocolateLotState. Remember that the Tempering Application only cares about a lot if the `next_station` field indicates that the tempering machine is the next station.

1. Find the examples as described in *Clone Repository*.
2. Generate the C# code for `chocolate_factory.idl` as explained in *Build the Applications*.
3. Create a ContentFilteredTopic that filters out ChocolateLotState data unless the `next_station` field in the data refers to this application.

   Open `TemperingApplication.cs` and look for the comment:

   ```
   // Exercise #1.1: Create a Content-Filtered Topic that filters out
   // chocolate lot state unless the next_station = TEMPERING_CONTROLLER
   ```

   Add the following code after the comment, so it looks like the following:

```
// Exercise #1.1: Create a Content-Filtered Topic that filters out
// chocolate lot state unless the next_station = TEMPERING_CONTROLLER
ContentFilteredTopic<ChocolateLotState> filteredLotStateTopic =
      participant.CreateContentFilteredTopic(
         name: "FilteredLot",
         relatedTopic: lotStateTopic,
         filter: new Filter(
            expression: "next_station = %0",
            parameters: new string[] { "'TEMPERING_CONTROLLER'" }));
```

This code creates a ContentFilteredTopic, using the ChocolateLotState *Topic*, that will filter out all chocolate lot state data that the Tempering Application does not care about.

4. Use the ContentFilteredTopic instead of a plain *Topic*.

   In `TemperingApplication.cs`, look for the comment:

   ```
   // Exercise #1.2: Change the DataReader's Topic to use a
   // Content-Filtered Topic
   ```

   Replace these lines:

   ```
   DataReader<ChocolateLotState> lotStateReader = subscriber.
   ↪CreateDataReader(
         lotStateTopic,
         ...);
   ```

   So the code looks like this (the highlighted line has changed):

   ```
   // Exercise #1.2: Change the DataReader's Topic to use a
   // Content-Filtered Topic
   DataReader<ChocolateLotState> lotStateReader = subscriber.
   ↪CreateDataReader(
         filteredLotStateTopic,
         ...);
   ```

   Now the *DataReader* is using the ContentFilteredTopic. You don't need to make any changes on the *DataWriter* side.

5. Remove the code from the `ProcessLot` that checks that `next_station` is the Tempering Application.

   In `TemperingApplication.cs`, look for the comment:

   ```
   // Exercise #1.3: Remove the check that the Tempering Application is
   // the next_station. This will now be filtered automatically.
   ```

   Delete the following `if` condition:

   ```
   if (sample.next_station == StationKind.TEMPERING_CONTROLLER)
   ```

   So the code looks like this:

```
foreach (var sample in samples.ValidData())
{
   // Exercise #1.3: Remove the check that the Tempering Application is
   // the next_station. This will now be filtered automatically.
   Console.WriteLine("Processing lot " + sample.lot_id);
   ...
   Console.WriteLine("Lot completed");
}
```

Now *Connext* automatically filters out ChocolateLotState data before your application processes it.

6.3 Hands-On 2: Review the Temperature DataReader's ContentFilteredTopic

We have already made a similar change as you made in Hands-On 1 to the Monitoring/Control Application's Temperature *DataReader*: we added content filtering to it. That application now filters out data unless the temperature is outside of the expected range. (And it prints an error when the temperature is above or below that range.)

1. In `MonitoringCtrlApplication.cs`, review the following code:

```
// A Topic has a name and a datatype. Create a Topic with type
// ChocolateLotState.  Topic name is a constant defined in the IDL file.
Topic<ChocolateLotState> lotStateTopic =
      participant.CreateTopic<ChocolateLotState>("ChocolateLotState");
// Add a Topic for Temperature to this application
Topic<Temperature> temperatureTopic =
      participant.CreateTopic<Temperature>("ChocolateTemperature");
ContentFilteredTopic<Temperature> filteredTemperatureTopic =
      participant.CreateContentFilteredTopic(
         name: "FilteredTemperature",
         relatedTopic: temperatureTopic,
         filter: new Filter(
            expression: "degrees > %0 or degrees < %1",
            parameters: new string[] { "32", "30" }));
```

   Notice that we have added a ContentFilteredTopic that will filter out temperature data unless it's outside of the expected range. This uses a slightly more complex filter expression than the one you added to the Tempering Application in *Hands-On 1: Update the ChocolateLotState DataReader with a ContentFilteredTopic*. More information about the possible filter expressions can be found in SQL Filter Expression Notation, in the RTI Connext Core Libraries User's Manual.

2. You can see later in the code that the Temperature *DataReader* has been updated to use the ContentFilteredTopic:

```
DataReader<Temperature> temperatureReader =
    subscriber.CreateDataReader(filteredTemperatureTopic, readerQos);
```

3. The logic in the `MonitorTemperature` function no longer checks whether the temperature is out of range, because temperature data is only received when it's out of range. Now the function looks

like this:

```
foreach (var data in samples.ValidData())
{
      // Receive updates from tempering station about chocolate␣
↪temperature.
      // Only an error if below 30 or over 32 degrees Fahrenheit.
      Console.WriteLine("Temperature high: " + data);
}
```

4. Build and run the applications.

```
$ dotnet build
```

Use the script to start all the applications at once:

Linux

```
$ ./start_all.sh
```

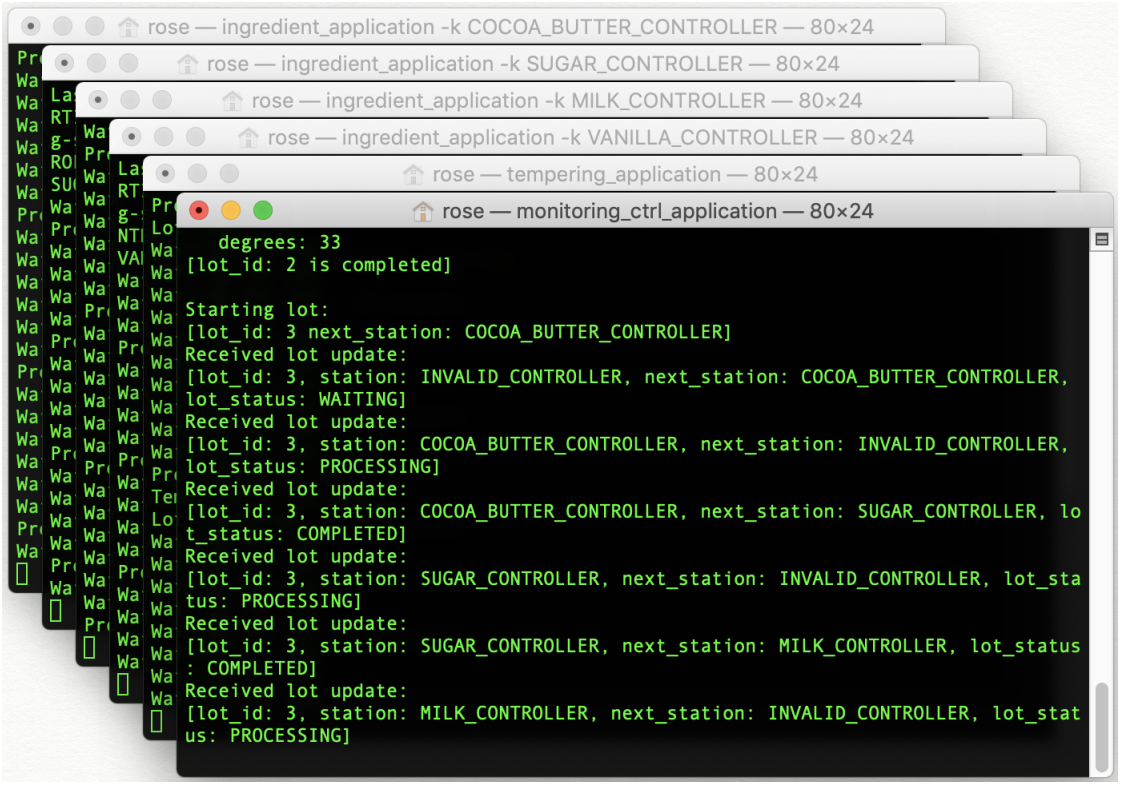
macOS

```
$ ./start_all.sh
```

Windows

```
> start_all.bat
```

The script runs four copies of the Ingredient Application, each one specifying a different type of ingredient it is providing. The script also runs the Tempering Application and the Monitoring/Control Application. You can watch each lot get processed by the ingredient applications, and then the tempering machine.



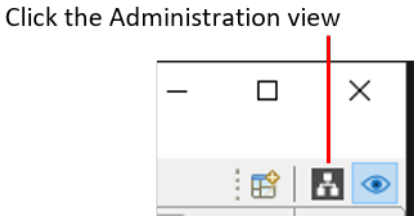
If you do not want to run the script, you can start up each of the applications by opening a new terminal for each application, and starting each application as follows:

```
$ dotnet run -p IngredientApplication -- --station-kind COCOA_BUTTER_
↪CONTROLLER

$ dotnet run -p IngredientApplication -- --station-kind SUGAR_CONTROLLER

$ dotnet run -p IngredientApplication -- --station-kind MILK_CONTROLLER

$ dotnet run -p IngredientApplication -- --station-kind VANILLA_
↪CONTROLLER

$ dotnet run -p TemperingApplication

$ dotnet run -p MonitoringCtrlApplication
```

6.4 Hands-On 3: Review the Larger System in Admin Console

We now have six applications that are publishing and subscribing to data, which is not a very large distributed system. This can start to look overwhelming, so we're going to open *Admin Console* to get an overview of this system. This hands-on exercise does not relate directly to content filtering, but shows how multiple applications start to look in *Admin Console*.

1. Make sure all of your applications are still running.
2. Open *Admin Console*.
   - Open *Admin Console* from *RTI Launcher*.
   - Choose the Administration view:



   (The Administration view might be selected for you already.)
3. In the DDS Logical View, click on the Domain in your system:

4. Explore the chocolate factory system. Even though it looks complex with multiple applications running, you can navigate by clicking on the *Topic* name or the application. This makes it much easier to visualize the larger system.

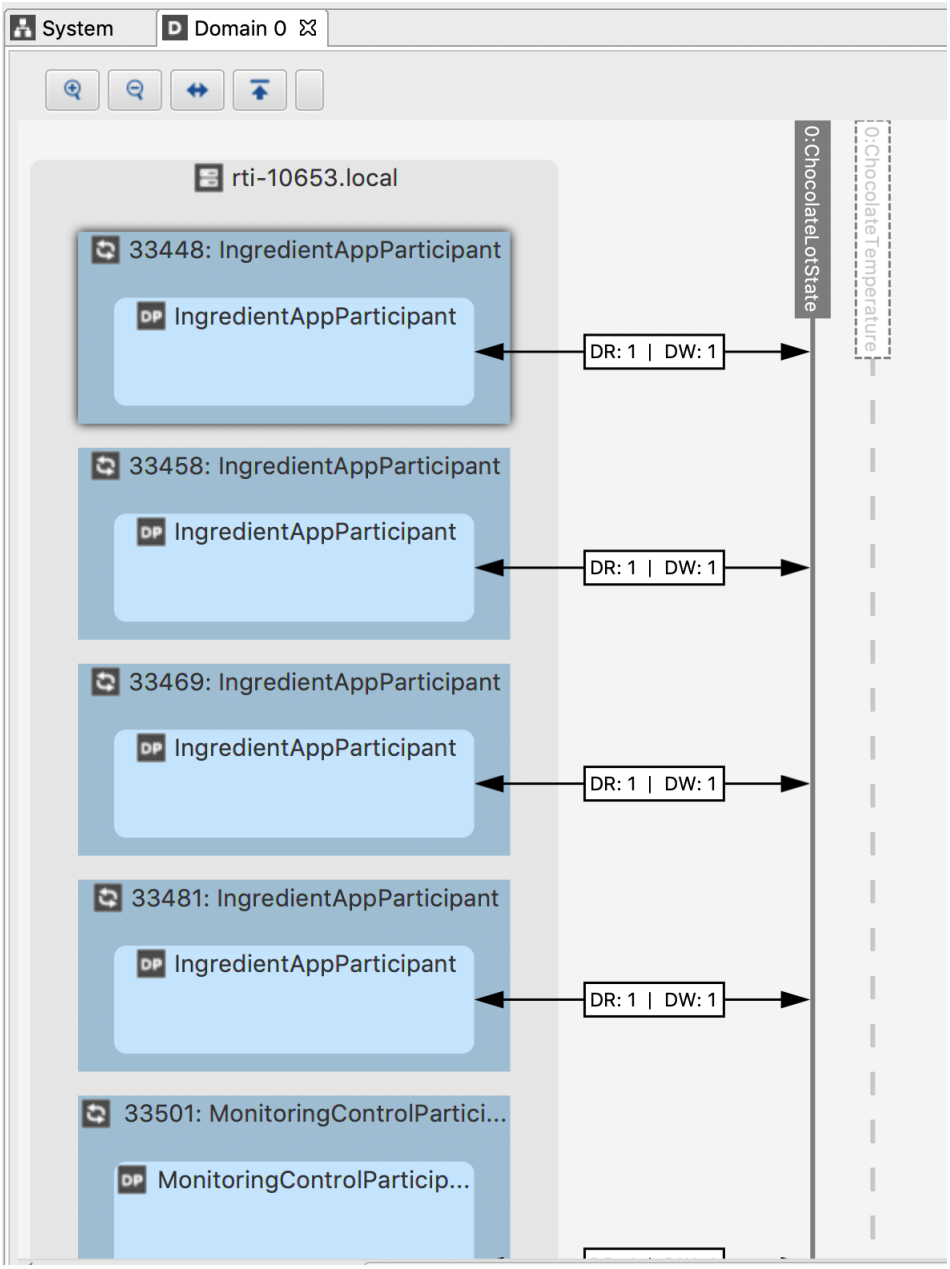


Figure 6.5: Each Ingredient Application shows one *DataReader* and one *DataWriter* communicating on the ChocolateLotState *Topic*

Chapter 7

Discovery

Prerequisites	• *Publish/Subscribe* • Repository cloned from GitHub here
Time to complete	30 minutes
Concepts covered in this module	• Discovery • Domains • DomainParticipants

Before we talk about discovery, we should talk about the *DomainParticipant* entity that you create when you create your DDS application. If you walk through all of the code we have used so far, you will see that creating a *DomainParticipant* is nearly the first step our applications take, and a *DomainParticipant* is used to create all the *Publishers*, *Subscribers* (and ultimately the *DataWriters* and *DataReaders*) your application uses. When you create a *DomainParticipant*, it starts the discovery process.

Definition

Discovery is the process in which *DomainParticipants* find out about other *DomainParticipants* and exchange information about their *DataWriters* and *DataReaders*. When a *DomainParticipant* learns about the *DataWriters* and *DataReaders* belonging to another *DomainParticipant*, it analyzes whether they have matching *Topics*, datatypes, and compatible QoS with its own *DataReaders* and *DataWriters*. If a *DataWriter* and *DataReader* are determined to be compatible, they communicate. By default, discovery starts as soon as you create a *DomainParticipant* in your application.

Discovery is an ongoing process—whenever *DataWriters* and *DataReaders* are created, modified, or deleted, their *DomainParticipant* sends that information to the other *DomainParticipants* in the system. In addition, *DomainParticipants* maintain their liveliness in the system using periodic discovery announcements.

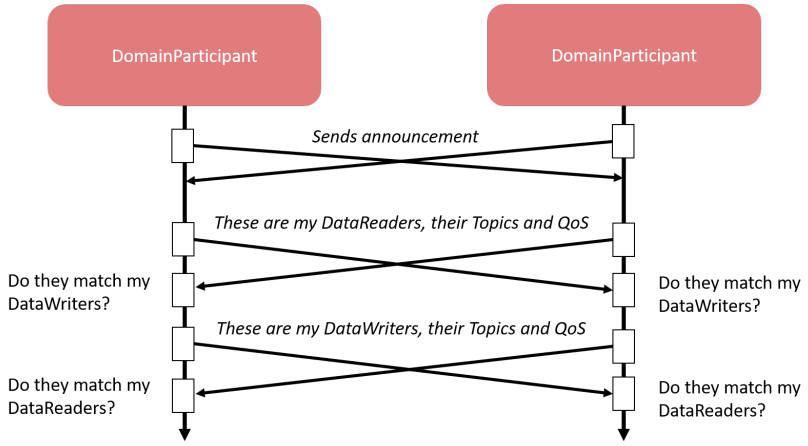


Figure 7.1: Simplified discovery diagram. All *DomainParticipants* in your system will send discovery information at startup (by default), and they will all determine whether they have *DataWriters* and *DataReaders* that match. Any time you create a new *DataWriter* or *DataReader* in your application, its *DomainParticipant* will announce the existence of the new *DataWriter* or *DataReader* to all the *DomainParticipants* it has discovered.

Tip: Notice that creating a *DomainParticipant* means creating discovery traffic over the network. This is one reason we recommend you create as few *DomainParticipants* as possible, according to your system design. Often there is only one *DomainParticipant* per application with multiple *DataReaders* and *DataWriters* underneath it, but that may depend on the number of domains you need.

7.1 Domains

When you create a *DomainParticipant*, you pass it a positive integer value that represents the DDS domain it should communicate in. A DDS domain is a logical DDS network. If you create *DomainParticipants* in different domains, they will not communicate with each other.

Tip: Using multiple domains is not a way to secure your DDS system, but it is a way to keep DDS applications that should not communicate from interfering with each other or from using unnecessary resources storing information about *DataWriters* or *DataReaders* they will never communicate with. For information on securing a DDS system, see the RTI Security Plugins Getting Started Guide.

There are many reasons for creating multiple DDS domains. For example:

- You need to run a production system and a test system on the same network, and you do not want them to communicate with each other.
- You have multiple developers running tests, and you don't want them to collide with each other
- You have different subsystems in your overall system, and they don't need to communicate with each other.

- You are using Security Plugins *(RTI Security Plugins)*, and you want to support both a secure domain and a non-secure domain in your system.

Sometimes, you may have an application that needs to interact in multiple DDS domains. It does that by creating a *DomainParticipant* for each domain it wants to communicate in.

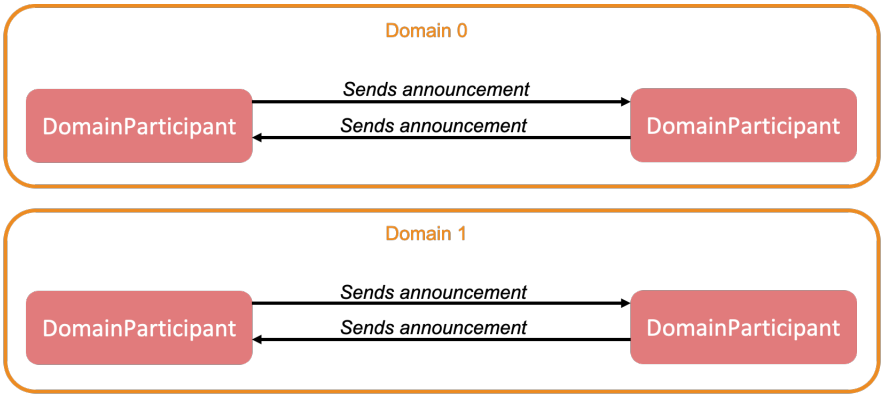


Figure 7.2: *DomainParticipants* in different domains do not send announcements to each other.

7.2 Initial Peers

The initial peers define where a *DomainParticipant* sends its first announcements. These peers are a list of addresses (in string format) that a *DomainParticipant* should attempt to contact. *Connext* can support transports other than UDP, so these strings use a format called a "locator" that allows you to specify which transport as well as the address. The details on how to format locator strings can be found in Discovered RTPS Locators and Changes with IP Mobility, in the RTI Connext DDS Core Libraries User's Manual. Most of the time you will be specifying locators that look like normal host names and IP addresses.

By default, a *DomainParticipant* sends its announcements to:

- A multicast IP address with a port associated with the domain the participant is communicating in.
- Loopback IP address with ports associated with the domain the participant is communicating in.
- A shared memory locator with an ID associated with the domain the participant is communicating in.

If you need to change the initial peers your *DomainParticipants* contact, you can do that a few different ways, but the easiest way is in the XML configuration file:

```
<qos_library>
    <qos_profile name="DefaultProfile">
        <domain_participant_qos>
            <discovery>
                <initial_peers>
                    <!-- Add an element for each machine you want to␣
↪communicate with -->
                    <element>IP address of machine to contact</element>
                </initial_peers>
            </discovery>
        </domain_participant_qos>
```

```
    </qos_profile>
</qos_library>
```

7.3 Hands-On 1: Troubleshooting Discovery

Discovery may not work for a number of reasons. The most common reasons for discovery not working (or appearing not to work) are:

- A firewall is blocking some or all DDS traffic.
- There is a router in your network that is not allowing multicast traffic through.
- Discovery is working, but your *Topic* names are misconfigured.
- Discovery is working, but your QoS profiles are misconfigured.

To troubleshoot whether there is DDS communication going between applications, we will use the *rtiddsping* utility. This is a simple utility that sends ping messages over DDS from a *DataWriter* to a *DataReader*, and can help determine whether an issue is with discovery or with misconfiguration. This is more interesting when run on multiple machines, but you can do this hands-on on a single machine to understand how to do this simple troubleshooting step.

1. In one terminal, run *rtiddsping*:

   Linux

   ```
   $ <NDDSHOME>/bin/rtiddsping
   ```

   macOS

   ```
   $ <NDDSHOME>/bin/rtiddsping
   ```

   Windows

   ```
   > <NDDSHOME>\bin\rtiddsping.bat
   ```

2. In a second terminal, run *rtiddsping* with the `-subscriber` parameter:

   Linux

   ```
   $ <NDDSHOME>/bin/rtiddsping -subscriber
   ```

   macOS

   ```
   $ <NDDSHOME>/bin/rtiddsping -subscriber
   ```

   Windows

   ```
   > <NDDSHOME>\bin\rtiddsping.bat -subscriber
   ```

   After you start both applications, you should see output similar to the following in the *rtiddsping* publishing application:

```
RTI Connext DDS Ping built with DDS version: ...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Sending data... value: 0000000
Found 1 additional ping subscriber(s).
Current subscriber tally is: 1
Sending data... value: 0000001
Sending data... value: 0000002
Sending data... value: 0000003
Sending data... value: 0000004
Sending data... value: 0000005
Sending data... value: 0000006
```

You should see output similar to the following in the *rtiddsping* subscribing application:

```
RTI Connext DDS Ping built with DDS version: 6.1.0 (Core: 1.9a.00, C:
1.9a.00, C++: 1.9a.00)
Copyright 2012 Real-Time Innovations, Inc.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
rtiddsping is listening for data, press CTRL+C to stop it.
Found 1 additional ping publishers(s).
Current publisher tally is: 1
Found 1 additional alive ping publishers(s).
Current alive publisher tally is: 1
rtiddsping, issue received: 0000001
Detected Missed Sample(s) current: 1 cumulative: 1 (50.0)%
rtiddsping, issue received: 0000002
rtiddsping, issue received: 0000003
rtiddsping, issue received: 0000004
rtiddsping, issue received: 0000005
rtiddsping, issue received: 0000006
```

If you do not see "issue received" messages in the subscribing *rtiddsping* application, there is likely an issue with a firewall or with your network. Now, try running *rtiddsping* with the option `-peer`. Specify the IP address of the machine where you are running the other *rtiddsping* application. This will help you determine whether there is an issue with multicast (perhaps due to a router between your machines). If communication occurs when you specify a peer, it is likely that there is no multicast available between your machines. This means that you may need to specify initial peers when running your applications.

For further troubleshooting steps, see this HOWTO article on the RTI community site: HOWTO Do Basic Debugging for System-Level DDS.

7.4 Hands-On 2: Start Applications in Different Domains

All of the applications we have built so far can take a parameter to start in a different domain than the default. Go back to any one of the previous examples, and start one of the applications with `-d 1`. Notice that it doesn't communicate with applications that are not in the same domain.

For example:

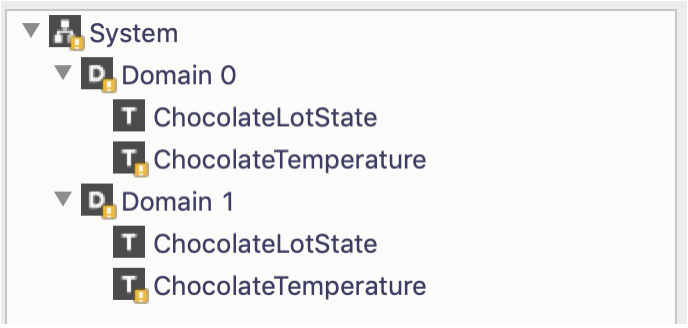
1. Start two applications.

   In the `6_content_filters/csharp/` directory, type the following:

   ```
   $ dotnet run -p MonitoringCtrlApplication -- --domain-id 1

   $ dotnet run -p TemperingApplication -- --domain-id 0
   ```

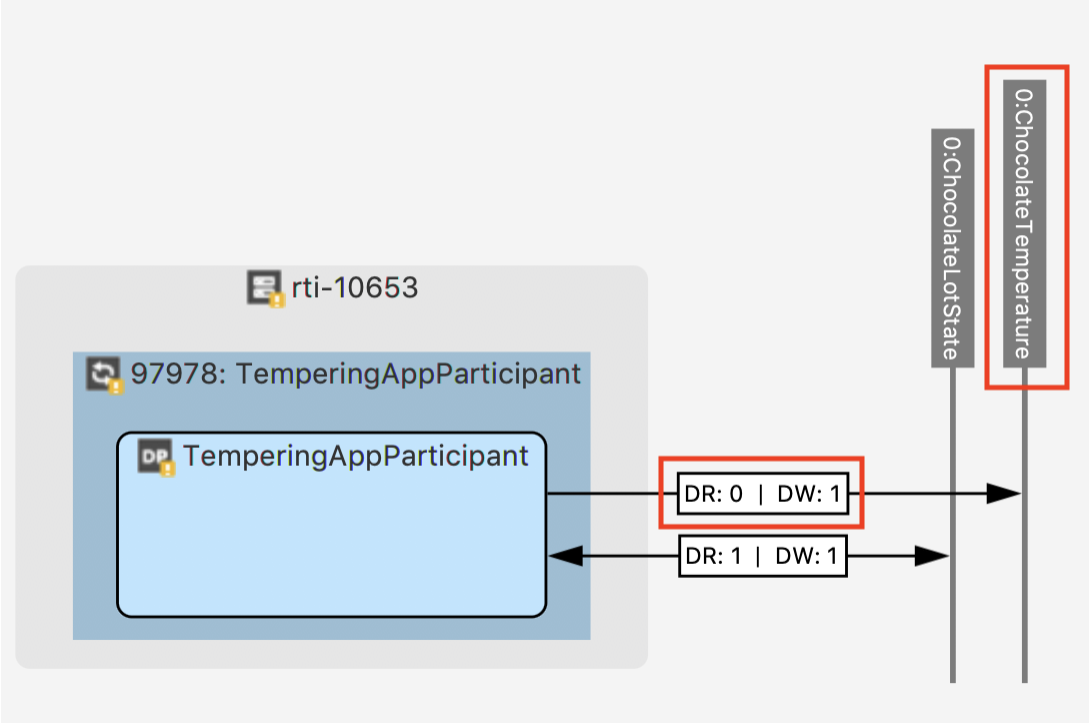
   Notice that the two applications do not communicate, because they are in different domains.

2. Open *Admin Console* and see that there are two domains in your system: domain 0 and domain 1. You can see that the ChocolateLotState and ChocolateTemperature *Topics* exist in both domains.



   Click on domain 0.

   You can see which *DataWriters* and *DataReaders* are communicating within domain 0. For example, you can see there are no *DataReaders* and one *DataWriter* of the ChocolateTemperature Topic:



7.5 Next Steps

Congratulations! You have learned the basics of discovery, domains, and initial peers. You have also learned about one of the utilities that can help you troubleshoot discovery issues. For more details on discovery, see Discovery, in the RTI Connext Core Libraries User's Manual

Chapter 8

Next Steps

We invite you to explore further by referring to the wealth of available information and resources.

8.1 Documentation

After installing *Connext*, you'll find the following resources in `<NDDSHOME>/doc/manuals/connext_dds_professional`[1] and at https://community.rti.com/documentation.

- If you are using *Connext* on an embedded platform, this document specifically addresses these configurations:
  - Addendum for Embedded Systems
- User's Manual: Describes the features of the product and how to use them.

You'll also find API Reference HTML Documentation at `<NDDSHOME>/README.html` and https://community.rti.com/documentation. This extensively cross-referenced documentation, available for all supported programming languages, is your in-depth reference to every operation in *Connext*.

- C API Reference
- C++ Traditional API Reference
- C++ Modern API Reference
- C# API Reference
- Java API Reference
- Python API Reference

[1] `<NDDSHOME>` refers to the main installation directory. See *Paths Mentioned in Documentation*.

8.2 Examples

In addition to the basic examples in your `rti_workspace` directory, you can find over 50 examples on how to use specific *Connext* features in the rticonnextdds-examples section on GitHub. Examples include asynchronous publication, compression, FlatData™, network capture, discovery (including builtin topics), DynamicData, and many more.

8.3 Updates

This Getting Started Guide may sometimes be updated online. Please see the RTI Community documentation website (https://community.rti.com/documentation) for updates or language additions.

Chapter 9

Copyrights and Notices

© 2020-2024 Real-Time Innovations, Inc. All rights reserved. May 2026

Trademarks

RTI, Real-Time Innovations, Connext, Connext Drive, 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 solely under and subject to RTI's standard terms and conditions available at https://www.rti.com/terms and in accordance with your License Acknowledgement Certificate (LAC) and Maintenance and Support Certificate (MSC), except to the extent otherwise agreed to in writing by RTI.

Third-Party Software

RTI software may contain independent, third-party software or code that are subject to third-party license terms and conditions, including open source license terms and conditions. Copies of applicable third-party licenses and notices are located at community.rti.com/documentation. IT IS YOUR RESPONSIBILITY TO ENSURE THAT YOUR USE OF THIRD-PARTY SOFTWARE COMPLIES WITH THE CORRESPONDING THIRD-PARTY LICENSE TERMS AND CONDITIONS.

Notices

Deprecations and Removals

Any deprecations or removals noted in this document serve as notice under the Real-Time Innovations, Inc. Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance and support of RTI's software.

Deprecated means that the item is still supported in the release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported.

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/