RTI Shapes Demo

A Demonstration of RTI Connext DDS

User's Manual

Version 6.0.0

Printed in U.S.A. First printing.
February 2019.

Trademarks

Copy and Use Restrictions

Technical Support

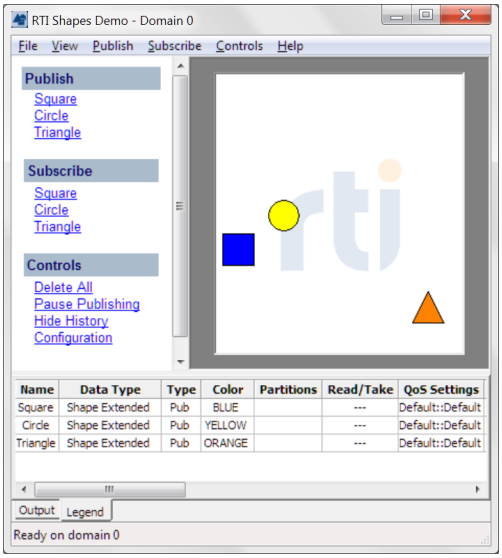
Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089
Phone: (408) 990-7444
Email: support@rti.com
Website: https://support.rti.com/

Contents

Chapter 1 Introduction

Welcome to *RTI® Shapes Demo*! This demonstration application is a self-contained introduction to the elegance and power of publish-subscribe networking. It goes beyond simple publishing and subscribing, however. This demo will also give you a glimpse of the goals and capabilities of *RTI Connext® DDS*. As you will see, *RTI Connext DDS* offers flexibility, performance, and reliability well beyond other networking technologies while addressing the challenge of extremely high-performance distributed networking.

Connext DDS offers flexible and fine-grained control over Quality of Service (QoS) parameters. No one application can showcase all the supported QoS parameters. *Shapes Demo* is intended to provide you with an abbreviated introduction to *Connext DDS* concepts; it covers a small subset of the many QoS parameters available in *Connext DDS*.



Shapes Demo publishes and subscribes to (writes and reads) colored moving shapes, which are displayed in the demo's window. Each copy of *Shapes Demo* can simultaneously publish and subscribe to many topics (shapes).

Shapes Demo also demonstrates the concepts of Extensible types. *Shapes Demo* can publish and subscribe to two different data types: the "Shape" type or the "Shape Extended" type. In a production scenario, your deployed applications are communicating using some existing data type. However, after deployment, you may find it necessary to modify the deployed data model. For instance, you may need to add new attributes. *Connext DDS*'s Extensible Types feature is designed to make your data type flexible and allow it to evolve over time.

1.1 Guide to this Document

This document will guide you through the demonstration and the underlying principles.

- 1.2 Goals of the Demonstration below outlines the concepts and goals of this demonstration.
- Chapter 2 Background Information on page 4 provides an overview of publish-subscribe and other communication paradigms. It also provides an overview of *Connext DDS* and its key concepts.
- Chapter 3 Installing and Using Shapes Demo on page 9 details the features of the demonstration application.
- Chapter 4 Examples on page 26 jumps right into using the application and playing with examples. Feel free to start here if you are familiar with publish-subscribe networking.
- Appendix A Running from the Command Line on page 64 explains how to run from the command line.
- Appendix B Troubleshooting on page 66 contains a few troubleshooting hints.

1.2 Goals of the Demonstration

There is no teacher like experience. Playing with this demonstration will give you a first-hand introduction to key *Connext DDS* concepts. These include:

- Anonymous publish-subscribe

  Applications communicating over publish subscribe networks do not need to know the source or destination of the data. This loosely coupled design simplifies (or eliminates) configuration, eases fault tolerance, and boosts performance.
- Dynamic discovery

  With publish subscribe, applications simply ask for the information they need and provide the information they have. The middleware does the hard task of finding the information and delivering it where it needs to go. There is no (or minimal) configuration; each node can simply join or leave the network at any time.
- Failover

  Connext DDS supports the concept of "ownership"; a publisher can own the responsibility for providing data to the network. Ownership makes failover simple; if the owner fails, a backup owner can instantly take over responsibility
- Failure notification

  Connext DDS is designed for the real world. In the case of failure, e.g., the violation of a deadline or the termination of service, interested applications are immediately notified.

- Extensible Types

  Connext DDS supports the "Extensible and Dynamic Topic Types for DDS" specification from the Object Management Group (OMG)[1]. (See *Connext DDS* documentation for details and limitations.) Using Extensible Types, existing applications that are designed to publish and subscribe data with a particular data model will be able to communicate with newer applications that use an extended/compatible data model—without any changes or recompilation.

- Advanced concepts

  If you are interested in exploring the demo more extensively, this document also briefly illustrates additional use cases such as content-filtered topics, reliability, durability and time-based filtering.

[1] http://www.omg.org/spec/DDS-XTypes/

Chapter 2 Background Information

This section provides an overview of existing middleware communication paradigms, including publish-subscribe, along with basic concepts of *Connext DDS*.

If you are already familiar with this information, you can go directly to Chapter 3 Installing and Using Shapes Demo on page 9.

2.1 Communication Models in Distributed Systems

Software applications are becoming increasingly distributed. A node in a distributed application must find the right data, know where to send it, and deliver it to the right place at the right time. Simplifying access to this data would enable a whole new class of distributed applications. The challenge, especially in embedded and real-time networks, is to quickly find and disseminate information to many nodes.

Three major communication paradigms have emerged to meet this need: client-server, message passing, and publish-subscribe.

Client-server is fundamentally a many-to-one design that works well for systems with centralized information, such as databases, transaction processing systems, and central file servers. However, if multiple nodes generate information, client-server architectures require that all the information be sent to the server for later redistribution to the clients, resulting in inefficient client-to-client communication. The central server is a potential bottleneck and single-point of failure. It also adds an unknown delay (and therefore indeterminism) to the system, because the receiving client does not know when it has a message waiting.

Message-passing architectures work by implementing queues of messages. Processes can create queues, send messages, and service messages that arrive. This extends the many-to-one client-server design to a more distributed topology. Message passing allows direct peer-to-peer connection; it is much easier to exchange information between many nodes in the system with a simple messaging design. However, the message-passing architecture does not support a data-centric model. Applications have to find data indirectly by targeting specific sources (e.g., by process ID or "channel" or queue name) on specific nodes. So, this architecture doesn't address how

applications know where a process/channel is, what happens if that process/channel doesn't exist, etc. The application must determine where to get data, where to send it, and when to perform the transaction. In the message-passing architecture, there is a model of the means to transfer data but no real model of the data itself.

Publish-subscribe adds a data model to messaging. Publish-subscribe nodes simply "publish" information they have and "subscribe" to data they need. Messages logically pass directly between the communicating nodes. The fundamental communications model implies both discovery (i.e. what data should be sent) and delivery (i.e. when and where to send the data). This design mirrors time-critical information delivery systems in everyday life (e.g. television, radio, magazines and newspapers). Publish-subscribe systems are good at distributing large quantities of time-critical information quickly, even in the presence of unreliable delivery mechanisms.

Publish-subscribe architectures map well to the real-time communications challenge. Finding the right data is straight forward; nodes just declare their interest once and the system delivers it. Sending the data at the right time is also natural; publishers send data when the data is available. Publish-subscribe can be efficient because the data flows directly from source to destination without requiring intermediate servers. Multiple sources and destinations are easily defined within the model, making redundancy and fault tolerance natural. Finally, the intent declaration process provides an opportunity to specify per-data-stream Quality of Service (QoS), requirements. Properly implemented, publish-subscribe delivers the right data to the right place at the right time.

In summary, client-server middleware is best for centralized data designs and for systems that are naturally service oriented, such as file servers and transaction systems. Client-server middleware is not the best choice in systems that entail many, often-poorly-defined data paths. Message passing, with "send that there" semantics, map well to systems with clear, simple dataflow needs. Message passing middleware is better than client-server middleware at free-form data sharing, but still require the application to discover where data resides. Publish-subscribe, by providing both discovery and messaging, implements a data centric information distribution system. Nodes communicate simply by sending the data they have and asking for the data they need.

2.2 Connext DDS Overview

Connext DDS presents a publish-subscribe connectivity framework that connects anonymous information producers (publishers) with information consumers (subscribers). The overall distributed application is composed of processes called "participants," each running in a separate address space, possibly on different computers. A participant may simultaneously publish and subscribe to typed data-streams identified by names called "Topics." The Application Programming Interface (API) offered by *Connext DDS* complies with the Object Management Group (OMG) Data Distribution Service (DDS) standard. It is the first comprehensive specification available for "publish-subscribe" data-centric designs.

Connext DDS defines a communications relationship between publishers and subscribers. The communications are decoupled in space (nodes can be anywhere), time (delivery may be immediately after publication or later), and flow (delivery may be reliably made at controlled bandwidth). To increase scalability,

topics may contain multiple independent data channels identified by "keys." This allows nodes to subscribe to many, possibly thousands, of similar data streams with a single subscription. When the data arrives, *Connext DDS* can sort it by the key and deliver it for efficient processing.

Connext DDS is fundamentally designed to work over unreliable transports, such as UDP or wireless networks. No facilities require central servers or special nodes. Efficient, direct, peer-to-peer communications, or even multicasting, can implement every part of the model.

2.2.1 Quality of Service

Fine control over Quality of Service (QoS) is perhaps the most important feature of *Connext DDS*. Each publisher-subscriber pair can establish independent QoS agreements. Thus, *Connext DDS* designs can support extremely complex, flexible data-flow requirements.

QoS parameters control virtually every aspect of the *Connext DDS* model and the underlying communications mechanisms. Many QoS parameters are implemented as "contracts" between publishers and subscribers; publishers offer and subscribers request levels of service. *Connext DDS* is responsible for determining if the offer can satisfy the request, thereby establishing the communication or indicating an incompatibility error. Ensuring that participants meet the level-of-service contracts guarantees predictable operation. More information about some important QoS parameters is presented below.

- **Deadline:** Periodic publishers can indicate the speed at which they can publish by offering guaranteed update deadlines. By setting a deadline, a compliant publisher promises to send a new update at a minimum rate. Subscribers may then request data at that or any slower rate.
- **Reliability:** Publishers may offer levels of reliability, parameterized by the number of past issues they can store for the purpose of retrying transmissions. Subscribers may then request differing levels of reliable delivery, ranging from fast-but-unreliable "best effort" to highly reliable in-order delivery. This provides per-data-stream reliability control.
- **Strength:** *Connext DDS* can automatically arbitrate between multiple publishers of the same topic with a parameter called "strength." Subscribers receive from the strongest active publisher. This provides automatic failover; if a strong publisher fails, all subscribers immediately receive updates from the backup (weaker) publisher.
- **Durability:** Publishers can declare "durability," a parameter that determines how long previously published data is saved. Late-joining subscribers to durable publications can then be updated with past values.

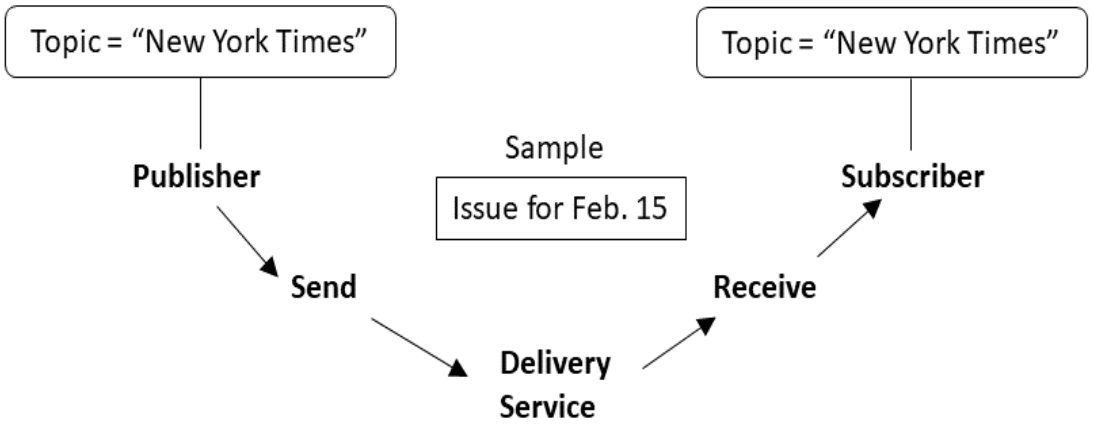
Other QoS parameters control when *Connext DDS* detects nodes that have failed, suggest latency budgets, set delivery order, attach user data, prioritize messages, set resource utilization limits, partition the system into namespaces, and more. *Connext DDS* QoS facilities offer unprecedented flexibility and communications control.

2.3 Publish-Subscribe Simple Analogy

The publish-subscribe communications model is analogous to that of magazine or newspaper publications and subscriptions. Think of a publication as a newspaper such as New York Times®. The Topic is the name of the periodical ("New York Times"). The type specifies the format of the information (weekly printed magazine or daily newspaper). The user data is the contents (text and graphics) of each sample (weekly or daily issues). The middleware is the distribution service (US Postal service or a paper delivery service) that delivers the reading material from where it is created (a printing house) to the individual subscribers (people's homes). This analogy is illustrated in Figure 2.1: Publish-Subscribe Example below.

Note that by subscribing to a publication, subscribers are requesting current and future samples of that publication, so that as new samples are published, they are delivered without having to submit another request for data.

Figure 2.1: Publish-Subscribe Example



The publish-subscribe model is analogous to publishing magazines or newspapers. The Publisher sends samples of a particular Topic to all Subscribers of that Topic. With the New York Times®, the Topic would be "New York Times." The sample consists of the data (articles and pictures) sent to all Subscribers daily or weekly. Connext DDS is the distribution channel: all of the planes, trucks, and people who distribute issues to the Subscribers.

In this example, Quality of Service (QoS) parameters can be linked to delivery requirements; deliver only the Sunday edition, the paper must be delivered by 7:00am; the paper must be in the mailbox or on the porch, etc. QoS parameters specify where, how, and when the data is to be delivered, controlling not only transport-level delivery properties, but also application-level concepts of fault tolerance, ordering, and reliability.

2.4 Publish-Subscribe Complex Analogy

Above, we drew an analogy between publish-subscribe and a newspaper delivery system. That is, of course, an oversimplification. Complex systems have complex data-delivery requirements. *Connext DDS* is perhaps more like a picture-in-picture-in-picture super-television system, with each super-TV set capable

of displaying dozens or even thousands of simultaneous channels. Super-TV sets can optionally be broadcast stations; each can publish hundreds of channels from locally mounted cameras to all other interested sets. Any set can add new pictures by subscribing to any channel at any time.

Each of these sets can also be outfitted with cameras and act as a transmitting station. TV sets publish many channels, and may add new outgoing channels at any time. Each communications channel, indeed each publisher-subscriber pair, can agree on reliability, bandwidth, and history-storage parameters, so the pictures may update at different rates and record outgoing streams to accommodate new subscribers.

These super-TV sets can also join or leave the network, intentionally or not, at any time. If and when they leave or fail, backup TV set-transmitters will take over their picture streams so no channels ever go blank.

That would be quite a system! It is only an analogy, but we hope this gives you some idea of the enormity of the real-time communications challenge. It also outlines the power of publish-subscribe: as you will see, *Connext DDS* provides simple parameters to permit all these scenarios with a remarkably simple and intuitive model.

2.5 Publish-Subscribe Example Application

An air traffic control system provides a more realistic example application. An air traffic control system monitors and directs all flights over an entire continent. The data distributed in such a system is in the form of aircraft tracks, which provides positional information (e.g., course, speed, etc.) about an airplane. Components of an air traffic control system would include radar systems, airplanes and air traffic control centers that provide current flight status information through real-time displays.

Managing correct distribution of data in such a system is complex. Each radar system can track many different airplanes, and each airplane may be tracked by more than one radar system. Real-time access to this information is needed for displays at air-traffic control centers so that air traffic controllers can make informed decisions. Air traffic controllers in the north-east may only want aircraft track information in their area, so only a subset of data needs to be provided to them. Based on current local conditions (e.g. air traffic, weather, etc.) air traffic controllers may issue flight plan updates back to airplanes in order to rout around inclement weather and other airplanes. Though airplanes do not need flight plans from all other air planes, it would be useful to have information about planes in the immediate vicinity.

Defining the air traffic control system in terms of publishers, subscribers and QoS parameters reveals that *Connext DDS* is a natural fit to address this data distribution problem. Each radar system can be thought of as a publisher that publishes the "tracks" topic which describes an airplane's positional information. Each airplane that the radar system is tracking can be thought of as an "instance" of the "track" topic. The real-time controller displays are both subscribers that subscriber to the "tracks" topic and publishers that publish "flight plan" topic updates back to the specific airplane. QoS parameters can be used to manage and control deterministic behaviors and fault tolerance capabilities of the system.

Chapter 3 Installing and Using Shapes Demo

3.1 Installation

Note: If you have *Connext DDS* installed, you may already have *Shapes Demo* installed. In this case, you do not need to install *Shapes Demo* separately.

- On Linux® systems:

  The distribution is packaged in a **.run** file. Run the downloaded file. For example:

  ```
  >  rti_shapes_demo-<version>-<architecture>.run
  ```

- On Windows® systems:

  The distribution is packaged in a .**exe** file. Simply double-click the file to run the installer.

- On Mac® OS X® systems:

  The distribution is packaged in a .**dmg** file. Double-click the file, this will open a folder that contains another file—double-click that file to run the installer.

The resulting installation directory will be named **rti_shapes_demo-<version>.**

3.2 Running Shapes Demo

You can run *Shapes Demo* on a single computer or on multiple workstations connected via Ethernet. Mac, Linux, and Windows operating systems are supported.

You can start multiple copies of the demo on as many computers as you would like (see below). By default, the demo discovers other instances using multicast, loopback, or shared memory. The discovery mechanism is fully configurable.

Note: *Shapes Demo* is not compatible with applications built with *RTI Data Distribution Service* 4.5e and earlier releases when communicating over shared memory. For more information, please see the Transport Compatibility section in the *RTI Connext DDS Core Libraries Release Notes* for release 5.3.1 or earlier.

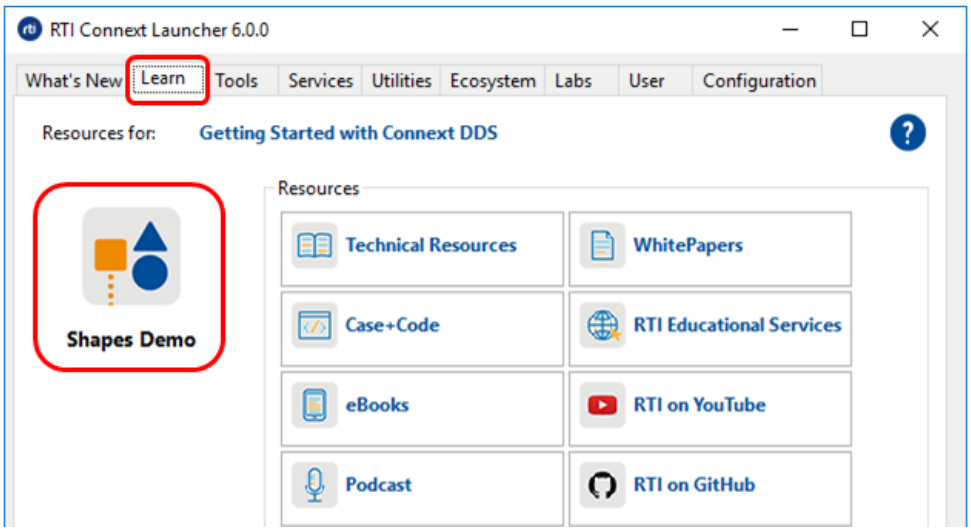
There are two ways to start *Shapes Demo*:

- From a command prompt:

  ```
  <Shapes Demo installation directory>/bin/rtishapesdemo
  ```

  For details on running from the command-line, see Appendix A Running from the Command Line on page 64.

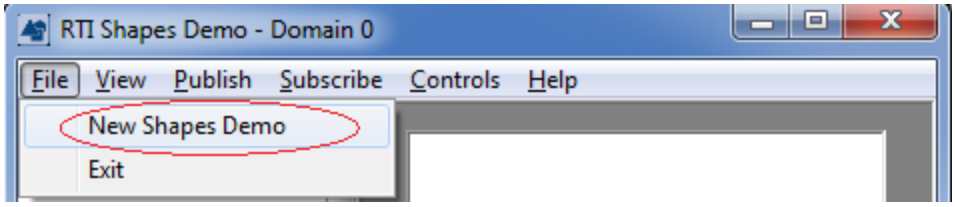
  Note: If *Shapes Demo* was installed as part of a *Connext DDS* bundle: the <Shapes Demo installation directory> mentioned in this document is the same as the <NDDSHOME> location mentioned in the *Connext DDS Core Libraries Getting Started Guide* and *User's Manual* (**rti_connext_dds-<version>**).
- From *RTI Launcher's* **Learn** tab, select **Shapes Demo**. (*RTI Launcher* is provided with *RTI Connext DDS*.)



Running a Second Instance of Shapes Demo

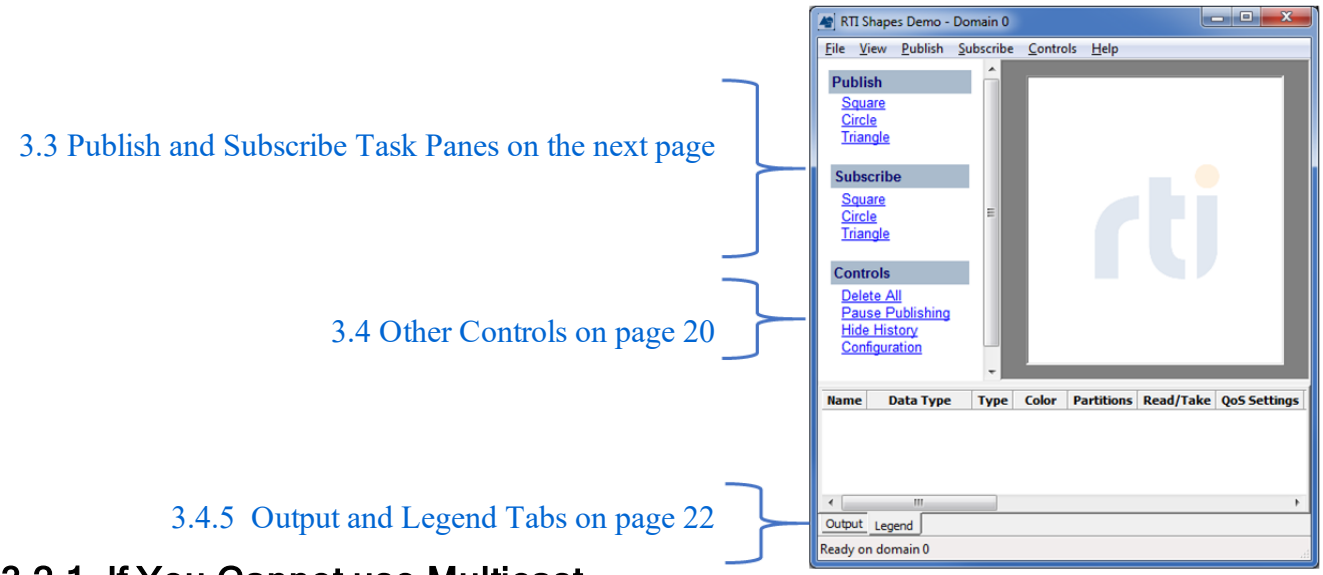
- Once you have an instance of *Shapes Demo* running, you can start additional ones by using the above options, or in *Shapes Demo*, select File, New Shapes Demo. The new instance of *Shapes*

Demo will be completely independent from the first one.



When *Shapes Demo* starts, you will see a window like that in Figure 3.1: Shapes Demo—Initial View below.

Figure 3.1: Shapes Demo–Initial View



3.2.1 If You Cannot use Multicast

If you want to run *Shapes Demo* on multiple computers that do not support multicast, or on a network that doesn't support multicast, you need to explicitly give *Shapes Demo* a list of all of the hosts that need to discover each other; we call this the *discovery peers list*. The list can contain hostnames and/or IP addresses. In its simplest format, each entry should be: **builtin.udpv4://<*hostname*|IP>**. The list can contain multiple, comma-separated entries.

To set your discovery peers list, either:

- Set the **NDDS_DISCOVERY_PEERS** environment variable:
  - **On Windows systems**: For example:

```
set NDDS_DISCOVERY_PEERS=builtin.udpv4://mypeerhost1, \
    builtin.udpv4://mypeerhost2
```

- **On UNIX-based systems**: For example, if you are using csh or tcsh:

```
setenv NDDS_DISCOVERY_PEERS builtin.udpv4://mypeerhost1,\
    builtin.udpv4://mypeerhost2
```

- Edit the profile in **<Shapes Demo installation directory>/resource/xml/RTI_SHAPES_DEMO_QOS_PROFILES.xml** or **<rti_workspace>/USER_RTI_SHAPES_DEMO_QOS_PROFILES.xml**. (See 3.5 Shapes Demo's Workspace on page 23.) For example, if the other machine has an IP address of 10.30.42.8:

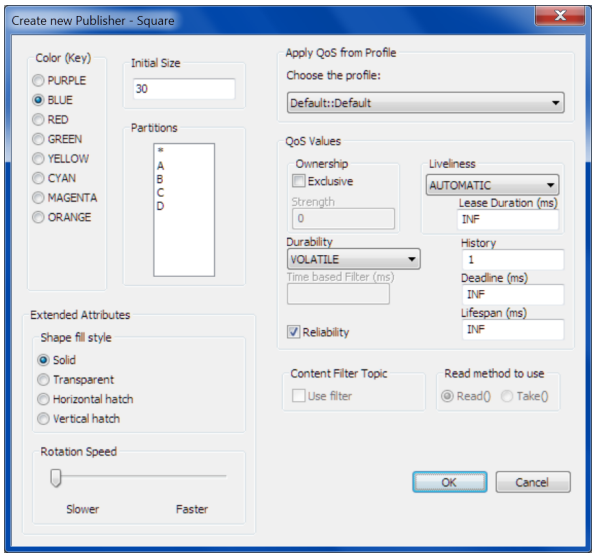
```
<discovery>
    <initial_peers>
        <element>builtin.udpv4://10.30.42.8</element>
    </initial_peers>
    <multicast_receive_addresses/>
</discovery>
```

3.3 Publish and Subscribe Task Panes

Connext DDS applications publish (write) and subscribe to (read) Topics. A Topic has a name and a type; the type defines the structure of the data.

Shapes Demo can publish and subscribe to three Topics: Square, Circle, and Triangle.

Clicking any of these options will open a dialog that allows you to set the QoS for the publisher/subscriber:



3.3.1 Color

Color is selectable only when creating a publisher. You can use color to represent different instances of the same topic (shape).

A shape's color is used as a *key*—simply a way to distinguish between data for multiple instances of the same shape (topic). Data that belongs to the same instance in the topic (shape) will have the same key (color).

The Color (key) area is grayed out for subscribers. The subscriber of a topic will receive all data sent on all instances of the topic.

3.3.2 Initial Size

The “initial size” field allows you to control how big the shape is.

3.3.3 Partitions

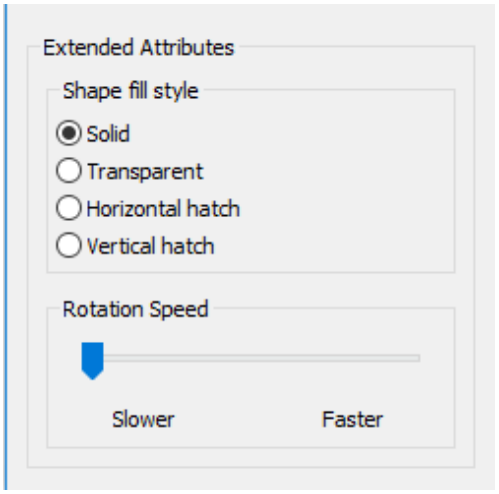
You can use partitions to dynamically isolate and group publishers and subscribers. If a publisher has a partition, then only subscribers with that same partition will receive data from that publisher.

The demo supports four partitions: A, B, C, and D. Partitions support regular expressions, so a publisher with a wildcard (*) partition will match subscribers with partitions A, B, C, and D.

A publisher with no partition (the default case) will not be matched with a subscriber that does have a partition. That is, "no partition" is *not* the same thing as a wild card (*) partition.

3.3.4 Extended Attributes

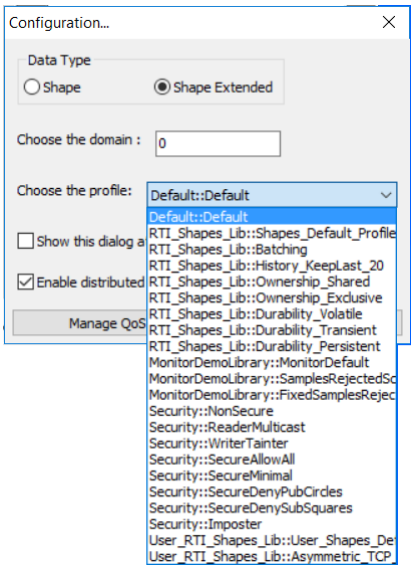
This section is enabled when you are using the “Shape Extended” Data Type in the Configuration dialog (see 3.4.4 Configuration on page 21), this is the default case. The extended attributes only apply to Publishers. (You will see it in the dialog for Subscribers, but it cannot be enabled.) You can choose a fill pattern and rotation speed for the shape.



These attributes illustrate a feature known as Extensible Types, which are described in the *RTI Connext DDS Core Libraries Getting Started Guide Addendum for Extensible Types*.

3.3.5 Applying QoS from a Profile

The drop-down listbox allows you to choose a QoS profile that has been pre-loaded from an XML file.



If the listbox contains only **Default::Default**, it means you haven't specified an XML file via the Configuration dialog (see 3.4.4 Configuration on page 21). In this case, **Default::Default** will result in all default QoS settings, as described in the *Connext DDS* API reference HTML documentation.

A profile contains the QoS values that will be used for the objects created by the demo. All QoS values not specified in the selected profile will use default values noted in the *Connext DDS* API reference HTML documentation. Any QoS settings that you make in the Create New Publisher/Subscriber dialog take precedence over the values in the selected profile. (See 3.3.6 Setting QoS Values on the facing page.)

Shapes Demo includes an XML file, **RTI_SHAPES_DEMO_QOS_PROFILES.xml**, which includes these profiles:

- **Default::Default**—This profile means you want to use whichever profile in the XML file is marked as the default (with <qos_profile name="x" is_default_qos="true">). In **RTI_SHAPES_DEMO_QOS_PROFILE.xml**, the default profile is **RTI_Shapes_Lib::Shapes_Default_Profile**.
- **RTI_Shapes_Lib::Shapes_Default_Profile**—Sets the data writer's **autodispose_unregistered_instances**[1] to false and the data reader's History **depth** to keep the last 6 samples.
- **RTI_Shapes_Lib::Batching**—Enables best-effort communication in the data writer and keeps the last 10 samples. It also enables batching with a maximum flush delay of 1 second and allows an unlimited number of bytes to be batched for up to 10 samples.
- **RTI_Shapes_Lib::History_KeepLast20**—Sets the data reader's History QoS to keep the last 20 samples.
- **RTI_Shapes_Lib::Ownership_Shared**—Sets Ownership to SHARED and Durability to TRANSIENT with direct communication to true for both the data reader and data writer. Both the reader's and writer's Liveliness is set to AUTOMATIC with a lease duration of 1 second. The reader has a History depth is 100 samples and uses RELIABLE reliability.
- **RTI_Shapes_Lib::Ownership_Exclusive**—Sets Ownership to EXCLUSIVE for both the data reader and data writer. The writer's Ownership Strength is set to 4.
- **RTI_Shapes_Lib::Durability_Volatile**—Sets Ownership to VOLATILE and History of 100 samples for both the data reader and data writer. The reader uses RELIABLE Reliability.
- **RTI_Shapes_Lib::Durability_Transient**—Sets Ownership to TRANSIENT for both the data reader and data writer.

[1] See Dispose vs. Unregister: on page 23.

- **RTI_Shapes_Lib::Durability_Persistent**—Sets Ownership to PERSISTENT for both the data reader and data writer.
- **MonitorDemoLibrary::Default**—Enables monitoring. See 3.6 Using Monitoring on page 24.
- **MonitorDemoLibrary::SamplesRejectedScenario**, **MonitorDemoLibrary::FixedSamplesRejectedScenario**—Profiles used in the tutorial for *RTI Monitor* (see 3.6 Using Monitoring on page 24).
- **Security::NonSecure**—Baseline profile of the Security QoS library with security disabled.
- **Security::SecureAllowAll**—Security enabled, subscribing and publishing to all topics is allowed.
- **Security::SecureMinimal**—Minimal security profile: does not protect outgoing data from being tainted, does not check that incoming data hasn't been tainted, and unencrypted topics will be communicated with Multicast (no security) participants.
- **Security::SecureDenyPubCircles**—Circles published by this participant will be accepted by secure participants because they have **enable_write_access_control** set to false for Circle.
- **Security::SecureDenySubSquares**—Prevents this participant from subscribing to Squares.
- **Security::Imposter (adversarial)**—Enables security. Tries to impersonate another peer by using its certificate.
- **Security::WriterTainter (adversarial)**—Simulates a man in the middle tainting data while it's in flight. This profiles applies to a data writer.
- **Security::ReaderMulticast**—Enables multicast in the data reader. This profiles applies to a data reader.

Note: In all the Security profiles, the Triangle and Square topics are encrypted. Circles remain unencrypted.

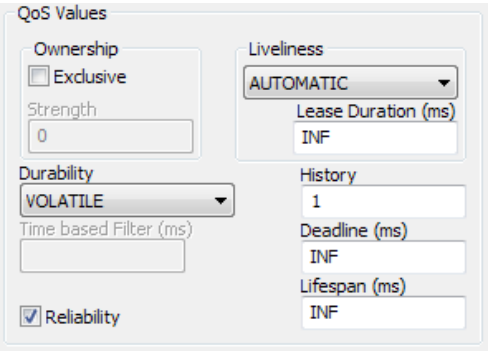
RTI_SHAPES_DEMO_QOS_PROFILES.xml is in **<Shapes Demo installation directory>/resource/xml**. If you open this file, you will see that these profiles have the property **base_name**, which points to another profile. The profile uses all the QoS settings of the profile pointed to by **base_name** plus the QoS settings that are explicitly specified. If a property is specified in both the base profile and the current profile, the property in the current one is used.

In the *Shapes Demo* workspace directory (see 3.5 Shapes Demo's Workspace on page 23) there is a file named **USER_RTI_SHAPES_DEMO_QOS_PROFILES.xml**. You can use this files as a template to create your own QoS profiles. *Shapes Demo* automatically loads the profiles from this file and the profiles in **RTI_SHAPES_DEMO_QOS_PROFILES.xml**.

3.3.6 Setting QoS Values

There are two ways to control the QoS values for the publisher and subscriber:

1. You can modify the QoS values in a profile and apply that profile as described in 3.3.5 Applying QoS from a Profile on page 13.
2. You can explicitly set some QoS values directly in the Create New Publisher/Subscriber dialog, as seen in this screenshot and described below. Values set in the dialog override values in the profile.



3.3.6.1 Exclusive Ownership and Strength

Ownership determines whether or not the instance (specified by color) of the Topic is exclusively owned by one publisher—that is, if multiple publishers of Red Squares can send data to this instance at the same time or not.

If the "Exclusive" check box is selected for a publisher, the Strength box will become available for input. The publisher with the highest Ownership Strength number is the only publisher that can write data to this instance.

If the "Exclusive" check box is selected for a subscriber, it means that the subscriber only wants data from one publisher—the one with the highest ownership strength.

The publisher and subscriber must use the same setting, so either check this box for both, or leave it unchecked for both. Otherwise, their QoS are incompatible and the publisher and subscriber will not communicate.

3.3.6.2 Durability

Durability controls whether the publisher will store the data that it sends, so that it can be sent to new subscribers that join the system later. The possible settings for this QoS are:

- **VOLATILE**(Default) Data samples are not stored.
- **TRANSIENT** *Connext DDS* will attempt to store samples in memory. The data will survive the data writer.
- **TRANSIENT_LOCAL***Connext DDS* will attempt to store samples in memory. The data will not survive the data writer.
- **PERSISTENT** *Connext DDS* will store previously published samples in permanent storage, like a disk. The data will survive the data writer.

Which particular samples are stored depends on other QoS such as History (3.3.6.6 History on page 18) and ResourceLimits.

If Durability is selected for a subscriber, the subscriber will ask the publisher to send all previously written data. All data in the publisher's history queue will be sent to the subscriber. To buffer this temporary high throughput, the subscriber should use a History value comparable to the publisher's.

The publisher and subscriber must use compatible settings, as described in Table 3.1 Valid Combinations of Durability.

Table 3.1 Valid Combinations of Durability

		Subscriber			
		VOLATILE	TRANSIENT_ LOCAL	TRANSIENT	PERSISTENT
Publisher	VOLATILE	yes	incompatible	incompatible	incompatible
	TRANSIENT_ LOCAL	yes	yes	incompatible	incompatible
	TRANSIENT	yes	yes	yes	incompatible
	PERSISTENT	yes	yes	yes	yes

Note: If you select Durability, you must also select Reliability (this applies to the publisher and subscriber).

3.3.6.3 Time-Based Filter

The Time-Based Filter field is only available when creating a subscriber. It is the minimum separation time (in milliseconds) that the subscriber wants between data updates. Any data arriving within this time interval will be discarded. Where possible, the publisher will not "publish" the data. Valid settings range from 0 to 31,536,000,000 ms (1 year).

The Time-Based Filter value must be less than the Deadline value (3.3.6.7 Deadline on page 19).

3.3.6.4 Reliability

The Reliability QoS can be RELIABLE or BEST_EFFORT. Selecting the Reliability check box sets Reliability to RELIABLE. If the check box is not selected, Reliability is set to BEST_EFFORT.

For publishers:

- The default is RELIABLE.
- If Reliability is RELIABLE (check box is selected), the publisher will attempt to deliver all the data that has been sent. If data is not received by the subscriber due to a communication error, *Connext DDS* will retransmit the data.

- If Reliability is BEST_EFFORT (check box is not selected), the publisher will use best-effort communication and will not retransmit any missing data.

For subscribers:

- The default is BEST_EFFORT.
- If Reliability is RELIABLE (check box is selected), the subscriber expects to receive all data updates reliably. The subscriber listens for "heartbeats" from the publisher and responds with either a positive acknowledgement to indicate data receipt or a negative acknowledgement to initiate retransmission of missing data.
- If Reliability is BEST_EFFORT (check box is not selected), the subscriber will not expect lost data to be resent.

The publisher and subscriber must use compatible settings, as described in Table 3.2 Valid Combinations of Reliability.

Table 3.2 Valid Combinations of Reliability

		Subscriber	
		Reliability not selected (default) (BEST_EFFORT)	Reliability selected (RELIABLE)
Publisher	Reliability not selected (default) (BEST_EFFORT)	yes	incompatible
	Reliability selected (RELIABLE)	yes	yes

3.3.6.5 Liveliness and Lease Duration

Liveliness is used to detect the state of the publisher even when it is not actively sending data. For a publisher, the Liveliness value is the maximum time interval within which a publisher will signal that it is active. For a subscriber, the Liveliness value is the maximum time interval within which a subscriber expects to be notified that the publisher is alive.

A subscriber's Liveliness must be greater than or equal to the publisher's Liveliness.Valid settings range from 0 to 31,536,000,000 ms (1 year), or "INF" for infinity (the default).

3.3.6.6 History

History controls the amount of data that is kept in the send queue. This is normally used in connection with Durability and/or Reliability. If Durability is selected, then History determines how much previously sent data is sent to late-joining subscribers. Valid settings range from 0 to 100,000,000. The default is 1.

3.3.6.7 Deadline

For a publisher, the Deadline value is the time interval within which the publisher commits to updating data at least once, if not more frequently.

For a subscriber, the Deadline value is the maximum time interval between data updates that the subscriber expects from the publisher.

If a publisher fails to send a data update within the subscriber's requested Deadline interval, the subscriber will get a "deadline missed" notification.

Valid settings range from 1 ms to 1 year, or "INF" for infinity (the default).

A subscriber's Deadline value must be greater than or equal to the publisher's. A subscriber's deadline must also be >= its minimum separation (see 3.3.6.3 Time-Based Filter on page 17).

3.3.6.8 Lifespan

Lifespan is only available when creating a publisher. The purpose of the Lifespan QoS is to avoid delivering stale data.

Each data sample written has an associated expiration time, beyond which the data should not be delivered. *Connext DDS* attaches timestamps to all data sent and received. The expiration time of each sample is computed by adding the specified Lifespan duration to the destination timestamp. When you specify a finite Lifespan, *Connext DDS* will compare the current time with those timestamps and drop data when the specified Lifespan expires. The default setting is an infinite duration, meaning the data will never 'expire.'

If you have multiple publishers for the same instance, they should all use the same Lifespan value.

Valid settings range from 1 ms to 1 year, or "INF" for infinity (the default).

3.3.7 Using a Content Filtered Topic

The 'Use filter' check box is only available when creating a subscriber. If selected, a filter is created for data updates to a topic based on the content of the data. Only data that satisfies the filter will be made available to the subscriber.

3.3.8 Controlling the Read Method

When creating a subscriber, you can choose whether it will use **read()** or **take()**.

With **read()** (the default), *Connext DDS* will continue to store the data in the data reader's receive queue. The same data may be read again until it is taken in subsequent **take()** calls. Graphically, a "new" sample is shown with a thicker border.

With **take()**, *Connext DDS* will remove the data from the data reader's receive queue. The data returned by *Connext DDS* is no longer stored by *Connext DDS*.

3.4 Other Controls

The Controls sub-panel includes various commands that you can use to control the demo.

3.4.1 Delete All

This command deletes all the publishers and subscribers that have been created in the demo application. All objects moving in the application window will disappear and no data will be sent or received. (NOTE: **Delete All** removes all the entities but it does not destroy the participant. The quick reset is to select **Configuration, Stop, Start**). If you have started multiple copies of *Shapes Demo,* you will need to click **Delete All** in each copy to delete their respective publishers and subscribers.

3.4.2 Pause Publishing

The **Pause Publishing** command is only effective on publishers. It pauses the sending of coordinate data for the shape until you click Resume Publishing. When **Pause Publishing** is clicked, the label changes to **Resume Publishing**.

The **Pause/Resume Publishing** commands are also available when you right-click an entity (if it is a publisher) in the **Legend** tab. In this way you can individually pause each single publisher.

When publishing is paused, you will still see published topics (colored shapes) moving in the publisher demo window, but corresponding topics in a subscriber window will stop moving. That's because what you see in the publisher window is the data being generated (not necessarily sent); what you see in the subscriber window is data being received. When you pause publishing, the subscriber stops receiving updates to the topic (that is, the shape's coordinates).

3.4.3 Show/Hide History

The **Show History** and **Hide History** commands tells the demo to start/stop drawing the shapes from all the packets that are in the subscriber's history queue.

This command has no effect on subscribers that use the **take()** method of accessing data. It is only for subscribers that use **read()**. It also has no effect on publishers.

If you set History greater than 1, by default all the packets in the history queue are displayed, showing the historical path of the shapes on the subscriber's canvas. If History is 1 (the default), no historical samples appear because there is only room for one sample in the queue.

By default, historical samples are shown; that is, **Show History** is the default setting and you will see the **Hide History** command in the Controls panel.

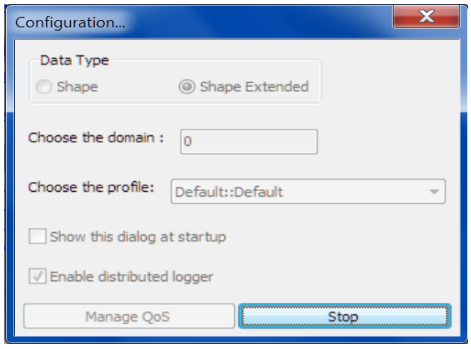
When you select **Show History**, the samples stay in the data reader's queue, so you can see the shadow trail of the historical samples (up to the number set in the History field).

3.4.4 Configuration

Note: To make changes with this dialog, first click **Stop**. Then make the desired changes and click **Start**.

The **Configuration** dialog is where you can change the domain ID, manage QoS profiles, and start/stop. Using the Stop and Start buttons is the equivalent of a Reset button, short of quitting and restarting the application.

The dialog also lets you choose between two data types: **Shape** and **Shape Extended** (the default). Use **Shape Extended** if you want to select the shape's fill pattern or rotation speed when you create a publisher (see 3.3.4 Extended Attributes on page 13).

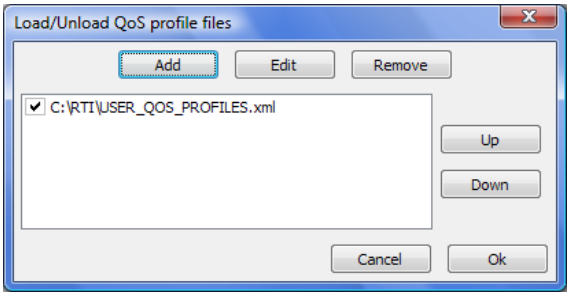


The "Enable distributed Logger" checkbox is described in 3.7 Using RTI Distributed Logger on page 24.

You can use the "Choose the profile" listbox to select a profile from one of the loaded XML files. If this listbox contains only "Default::Default", this means no XML files have been loaded.

To load an XML QoS Profiles file:

1. Click **Stop**. (Any publishers/subscribers will be deleted when you do this.)
2. Click **Manage QoS**.
3. In the resulting dialog box, click **Add**; then browse to select an XML QoS profiles file.



   You can use your own file, or the following files, which are provided with *Shapes Demo*:

   - **RTI_SHAPES_DEMO_QOS_PROFILES.xml**, in **<Shapes Demo installation directory>/resource/xml**. For information on the contents of this file, see 3.3.5 Applying QoS from a Profile on page 13.
   - **USER_SHAPES_DEMO_QOS_PROFILES.xml** in *Shapes Demo's* workspace. You can edit this file to include your own profiles.

   If you specify multiple XML files, the **Up** and **Down** buttons change the order in which they are loaded. If you load files that contain profiles with is_default_qos="true", the last profile loaded is used. This information is saved in the workspace (see 3.5 Shapes Demo's Workspace on page 23).

To unload an XML QoS Profiles file:

- Select **Configuration**, then **Stop**.
- Click **Manage QoS**.
- In the resulting dialog box, clear the check box next to the file, or select the file and click **Remove**.

If the XML QoS Profile file has errors:

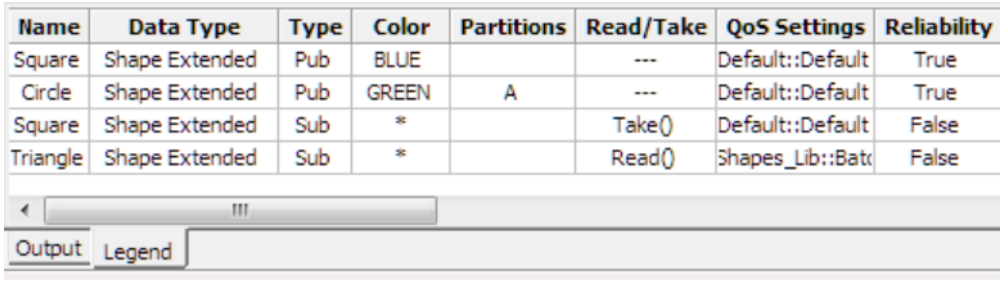
If you add an XML QoS Profile file that has errors and you click Ok, *Shapes Demo* will try detect the error and will show a popup that indicates with file has been detected to be wrong. Once you click **OK**, the **Load/Unload QoS profile files** window will automatically uncheck all the incorrectly formatted files.

At this point you can either press **Ok** and proceed without loading those files or edit them by pressing the **Edit button**: the default XML editor will open, allowing you to correct the file and correct the error.

3.4.5 Output and Legend Tabs

There are two tabs at the bottom of the demo application window.

- The **Legend** tab shows you the publishers and subscribers created for the demo and their QoS settings.



Right-click on a publisher entity in the **Legend** tab to access these commands:

- Pause/resume publishing (see 3.4.2 Pause Publishing on page 20)
- Dispose data and delete the data writer.
- Unregister data and delete the data writer.

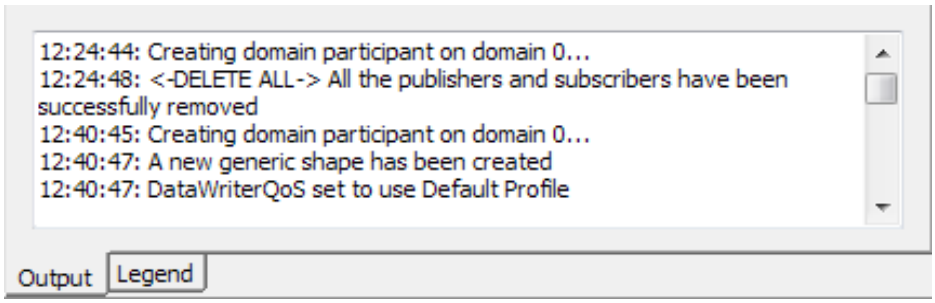
Right-click on a subscriber in the **Legend** tab to access a command to delete the data reader.

Another way to delete a publisher or subscriber is to click on it in the **Legend** tab and press the **Delete** button on your keyboard.[1]

Dispose vs. Unregister:

When data is *disposed*, all data readers are informed that, as far as the data writer knows, the data instance no longer exists and can be considered "not alive." When data is *unregistered*, this only indicates that a particular data writer no longer wants to modify an instance—an important distinction if there are multiple writers for the same instance.

- The **Output** tab shows statuses, events and other information.



3.5 Shapes Demo's Workspace

The workspace directory for *Shapes Demo* is here:

- On Mac systems:
  /Users/*your user name*/rti_workspace/*version*/user_config/shapes_demo
- On Linux systems:
  /home/*your user name*/rti_workspace/*version*/user_config/shapes_demo
- On Windows systems:
  <*your home directory*>\rti_workspace*version*\user_config/shapes_demo

Shapes Demo uses the concept of a workspace, which is an XML file that contains the last settings used by *Shapes Demo*. For example, it contains the list of QoS XML profile files loaded through the **Load/Unload QoS profile files** window and whether or not the files should be loaded. Another useful piece of information saved in the workspace is the last domain ID specified by the Configuration window. This allows you to start *Shapes Demo* with well-known settings each time. (If you start *Shapes Demo* with the **-domainId** option, that domain ID setting is not saved in the workspace.)

[1] When you press **Delete**, the current setting for the WriterDataLifecycle QoS policy's **autodispose_unregistered_instances** field determines if the writer's data is disposed before it is unregistered. If **autodispose_unregistered_instances** has not been changed via a QoS profile, the default setting will cause the data to be disposed and unregistered.

If the workspace directory contains a file named **RTI_SHAPES_DEMO.xml**, this file is used as the workspace file. You can specify a different workspace file by starting *Shapes Demo* with the **-workspaceFile** <*filename*> command-line option. If the file specified with this option cannot be found, it will be created.

If you do not use the **-workspaceFile** <*filename*> option and **RTI_SHAPES_DEMO.xml** is not in the workspace directory, *Shapes Demo* will automatically create **RTI_SHAPES_DEMO.xml** in the workspace directory.

3.6 Using Monitoring

This section is only useful if you have *RTI Monitor,* a graphical tool that displays monitoring data from *Connext DDS* applications in which monitoring is enabled.

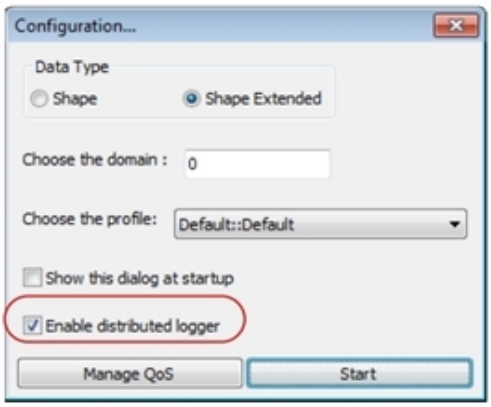
To enable monitoring in *Shapes Demo*, select the **MonitorDemoLibrary::Default** QoS profile described in 3.3.5 Applying QoS from a Profile on page 13. For more information on monitoring, please see the *RTI Monitor Getting Started Guide* and *RTI Monitor User's Manual*.

3.7 Using RTI Distributed Logger

Shapes Demo provides integrated support for RTI Distributed Logger and is enabled by default.

When you enable Distributed Logger, *Shapes Demo* will publish its log messages to *Connext DDS* in the same domain that *Shapes Demo* is using. Then you can use *RTI Monitor* or *RTI Admin Console*[1] to see the log message data. Since the data is provided in a DDS Topic, you can also use *rtiddsspy*[2] or even write your own visualization tool.

To disable/enable *Distributed Logger*, use the checkbox provided in the Configuration dialog under the Controls menu.



[1] *RTI Monitor* and *RTI Admin Console* are separate tools that can run on the same host as your application or on a different host.

[2] *rtiddsspy* is provided with *Connext DDS*.

3.8 Using Security

Shapes Demo supports the *RTI Security Plugins*, which provide security features such as authentication, access control, and encryption. These features can be tested by selecting the corresponding QoS profile within the Security QoS library as described in 3.3.5 Applying QoS from a Profile on page 13 and following the example in 4.7 Security Examples on page 41. For more information on security, please see the RTI Security Plugins Release Notes and RTI Security Plugins Getting Started Guide

Chapter 4 Examples

Important: Unless otherwise noted, these examples assume you are using the default *Shapes Demo* settings—meaning the **RTI_SHAPES_DEMO_QOS_PROFILES.xml** file is loaded. This file tells *Connext DDS* to load the profile called **Shapes_Default_Profile** from the **RTI_Shapes_Lib** library and use it as the default settings. For more information about profiles, see 3.3.5 Applying QoS from a Profile on page 13.

4.1 Publish-Subscribe Example

This example showcases the publish-subscribe concept. It uses best-effort communication and shows the decoupling between the publisher and the subscriber; i.e., the publisher can send data without knowing where/what the subscriber(s) are, and the subscriber can receive data without knowing where/what the publisher(s) are. In this example, you will be asked to start two copies of *Shapes Demo*. There is no need to configure a discovery service or provide any *a priori* information about where the demo applications are being run.

1. Create a red circle publisher:

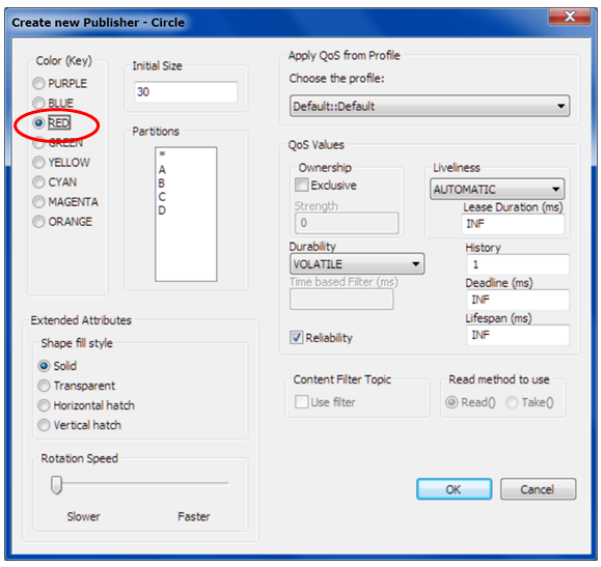
   a. Start *Shapes Demo*. We will refer to this instance of the application as Publisher1.

   b. Under **Publish**, click on **Circle**.

   c. In the Create New Publisher window:

      - Select **RED** for Color.
      - Click **OK**.

      You will see a red circle moving on the Publisher canvas. If there were



any subscribers, the publisher would start sending data (the coordinates of the red circle).

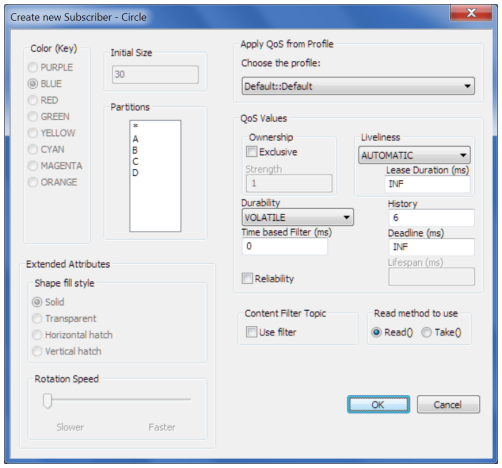
2. Create a subscriber for circles:

   a. Start a second *Shapes Demo*. We will refer to this instance of the application as Subscriber1.

   b. Under **Subscribe**, click on **Circle**.

   c. In the Create New Subscriber window:

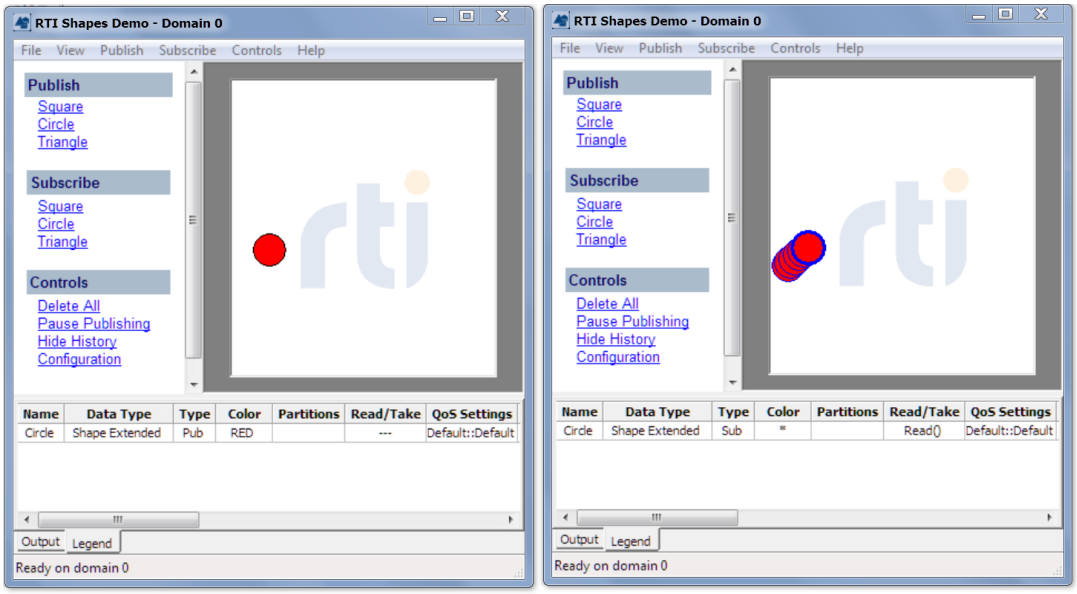
      - Click **OK**. (Use all the defaults.)



   You will see 6 red circles with blue borders on the Subscriber canvas, mirroring the movements of the circle in the Publisher canvas. The leading circle indicates the current position of the published circle. The other circles are the historical samples kept by *Connext DDS*. You can see the difference between historical data and new data looking at the thickness of the border. (You can also hide historical data by selecting **Hide History** from the Controls menu.)

Your windows should look similar to Figure 4.1: Publisher and Subscriber Displays below.

Figure 4.1: Publisher and Subscriber Displays



3. Test real-time data delivery:

   To show that the subscriber is receiving real-time data, move the cursor over the Publisher's red circle and click the mouse button. This will stop the red circle in the publisher canvas. Drag the

cursor and move it around while holding down the mouse button. The red circles on the subscriber canvas should exactly mirror your mouse movements.

Congratulations, you have just finished the first exercise, which illustrates basic publish-subscribe functionality!

If you plan to continue with the next exercise, **leave the two demo windows running**. The next exercise will use the red circles.

4.2 Multiple Instances Example

Instances are useful when you are dealing with data that is unpredictable in terms of its creation and deletion—e.g., aircraft/airplane flight tracks and shipment tracking. Flights and shipments can come and go. The application has no way of knowing when or how many flights/shipments show up. *Connext DDS* provides rich semantics that can be used to track, monitor, and check the state (new, deleted, no writers, etc.) of individual instances. Some of the possible notifications are displayed in the Output tab.

Publishers and subscribers are associated with a topic. If you create a new topic every time a new flight is detected, you would need to create a matching subscriber and publisher pair. This is obviously not scalable, since you can have many different aircraft flight plans.

Instances give you the ability to scale a topic. Unique instances of a topic are defined by unique key values. A subscriber of a topic will get all the data sent on all the instances of this topic. Take the example of flight track data: the key could be the flight ID, pilot name or mission code. Regardless of how many new flights there are, you would only need one subscriber to get the data, because the topic is the same.

In this example, the topic is the shape of the object (Square, Circle or Triangle) and the key is its color. So different colors of an object give you different instances of the topic. For example, a red circle is a different instance from a green circle, yet they are all instances of the Circle topic.

At this point, you should have two copies of *Shapes Demo* running, which will be referred to as Publisher1 and Subscriber1. In this example, you will be asked to start additional copies of *Shapes Demo*.

Tip: If you make a mistake during the following steps and need to delete a single publisher or subscriber, select the item in the Legend tab and press the **Delete** key on your keyboard.

1. This exercise picks up where the previous one left off. So you should have two demo windows running: one is publishing red circles (Publisher1) and the other is subscribing to circles (Subscriber1).
2. In Subscriber1, choose **Delete All** from the Controls Menu.
3. Create a circle subscriber with History = 1:
    a. In Subscriber1, under **Subscribe**, click on **Circle**.
    b. In the Create New Subscriber window:

- Change the History field from 6 to 1.
- Click **OK**.

You should now see one red circle moving in each instance of *Shapes Demo*.

4. Create a green circle publisher:

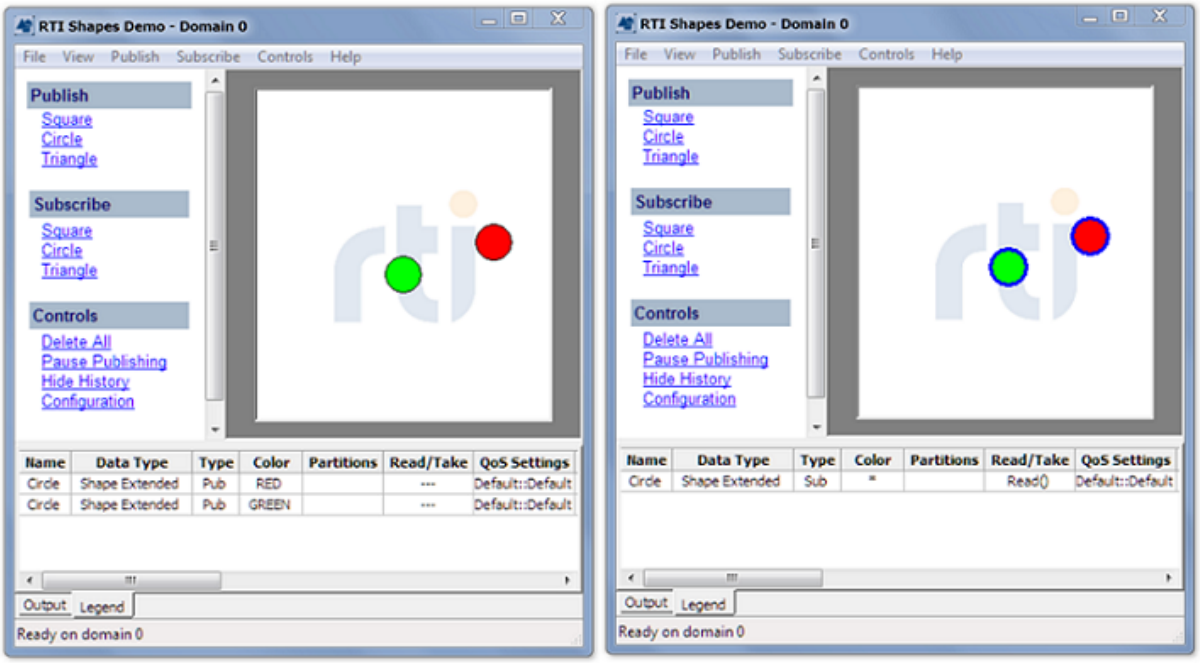
   a. In Publisher1 under **Publish**, click on Circle.

   b. In the Create New Publisher window:

      - Select **GREEN** for Color.
      - Click **OK**.

   You should see two circles moving on each canvas—one red and one green.

   Figure 4.2: Publisher and Subscriber Displays for Multiple Instances



Publisher1 Subscriber1

Notice that we did not have to do anything in Subscriber1 to start receiving the green circle's data. That's because the subscriber of a topic (Circle, in this case) gets all data sent for all instances of the topic. The green circle was just another instance of the topic Circle, so the subscriber received this new data automatically.

5. Create another red circle publisher in a new window:

   a. Start a third *Shapes Demo*. We will refer to this copy of the application as Publisher2.

b. Under **Publish**, click on **Circle**.

c. In the Create New Publisher window:

- Select **RED** for Color
- Click **OK**.

We now have multiple publishers updating the same instance (Red) of the topic Circle, as in Click OK. above. You'll see that the red circle in Subscriber1's canvas flickers between different locations. This happens because the subscriber is receiving position data from both of the publishers and is trying to display them at the same time. Details on how to handle such a situation will be discussed in the next section.

a. Start a third *Shapes Demo*. We will refer to this copy of the application as Publisher2.

b. Under **Publish**, click on **Circle**.

c. In the Create New Publisher window:

- Select **RED** for Color
- Click **OK**.

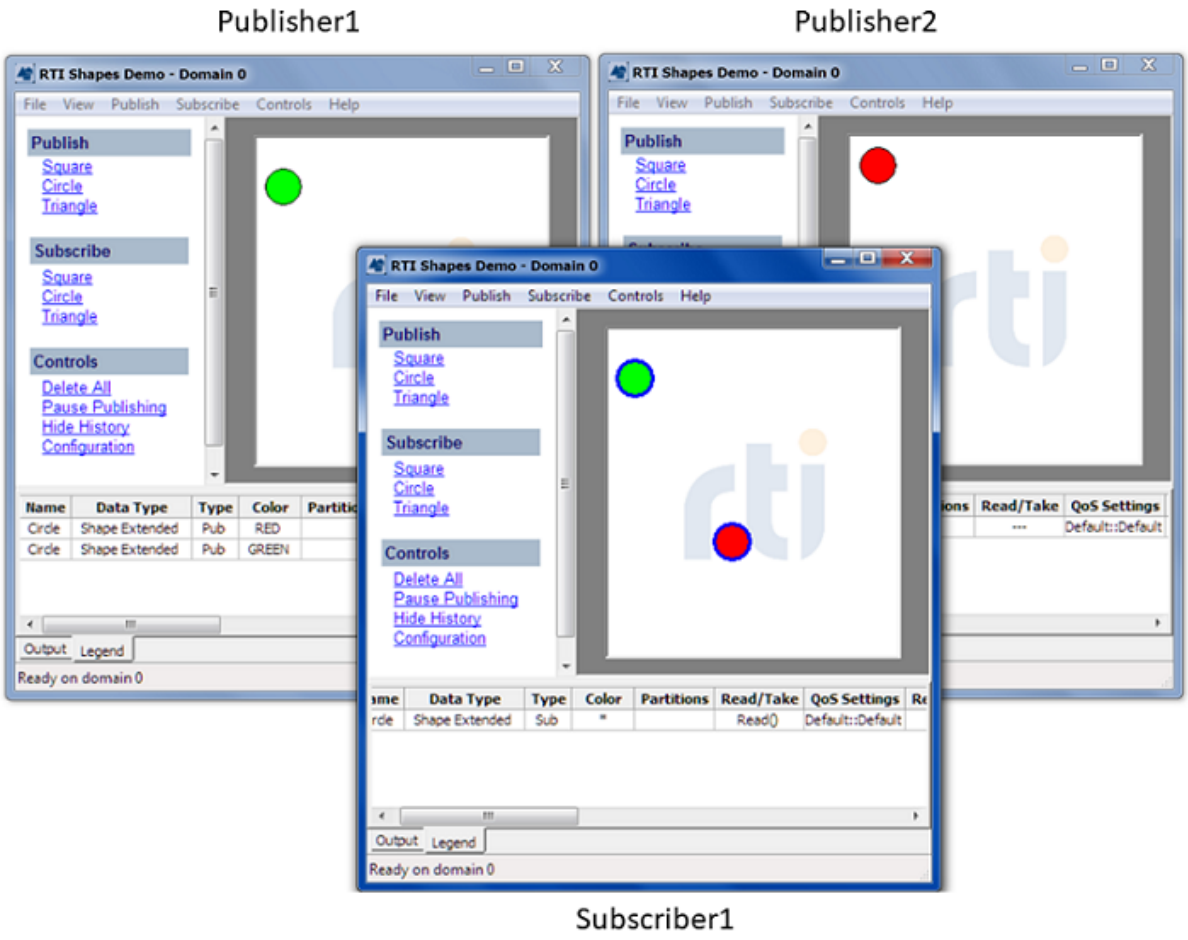
We now have multiple publishers updating the same instance (Red) of the topic Circle, as in Figure 4.3: Two Publishers and One Subscriber on the next page. You'll see that the red circle in Subscriber1's canvas flickers between different locations. This happens because the subscriber is receiving position data from both of the publishers and is trying to display them at the same time. Details on how to handle such a situation will be discussed in the next section.

6. Click **Delete All** in the **Controls** sub-panel of each of the three demo windows.

Note: A Subscriber shape may appear with an **X** or a **?** symbol on it:

- **X** means the instance has been disposed by the *DataWriter* (DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE).
- **?** means none of the *DataWriters* that are currently alive are writing the instance (DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE).
- For more information on these states, please see the *RTI Connext DDS Core Libraries User's Manual* or API reference HTML documentation.

Figure 4.3: Two Publishers and One Subscriber



4.3 Ownership Example

As you saw in the previous example, it's possible for multiple publishers to simultaneously send data to the same instance of a topic. You may or may not want this behavior. For certain types of data such as commands, you may want to receive updates from just one publisher at a time in order to be ensure consistency. Exclusive ownership is a way to ensure that only one publisher's data for a specific instance can get through to a subscriber. With multiple publishers, the one with the highest ownership strength wins.

At this point, you should have three copies of *Shapes Demo* running, which will be referred to as Publisher1, Publisher2 and Subscriber1. If you have not already done so, click **Delete All** on each one, so they are not publishing or subscribing to any shapes.

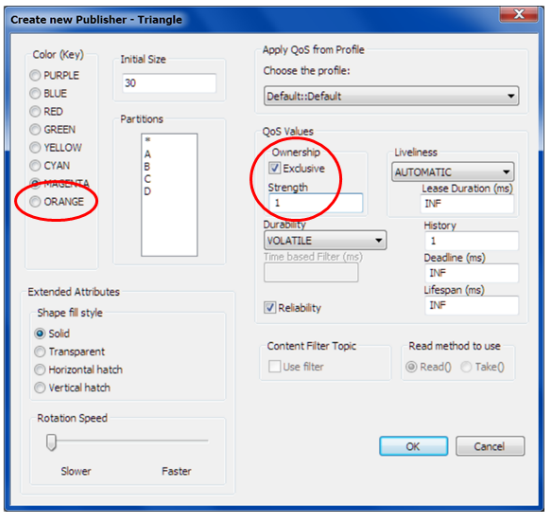
Tip: If you make a mistake during the following steps and need to delete a single publisher or subscriber, select the item in the Legend tab and press the **Delete** key on your keyboard.

1. In Publisher1, create an orange triangle publisher, with **Exclusive** ownership, **Strength** = 1:

   a. Under **Publish**, click on **Triangle**.

   b. In the Create New Publisher window:

   - Select **ORANGE** for Color.
   - Check **Exclusive**.
   - Set **Strength** to 1.
   - Click O**K**.



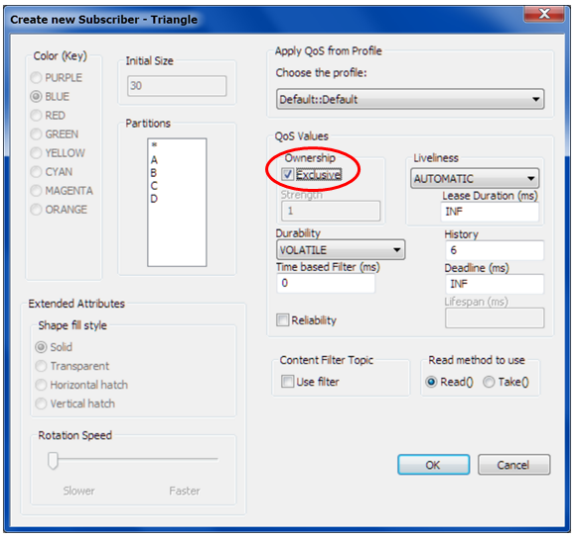
   You should see a floating orange triangle on the canvas. We created a publisher with exclusive ownership and a strength of 1.

2. In one of the other *Shapes Demo* windows, create a triangle subscriber with **Exclusive** ownership. We will call this window Subscriber1.

   a. Under **Subscribe**, click on **Triangle**.

   b. In the Create New Subscriber window:

   - Check **Exclusive**.
   - Click **OK**.



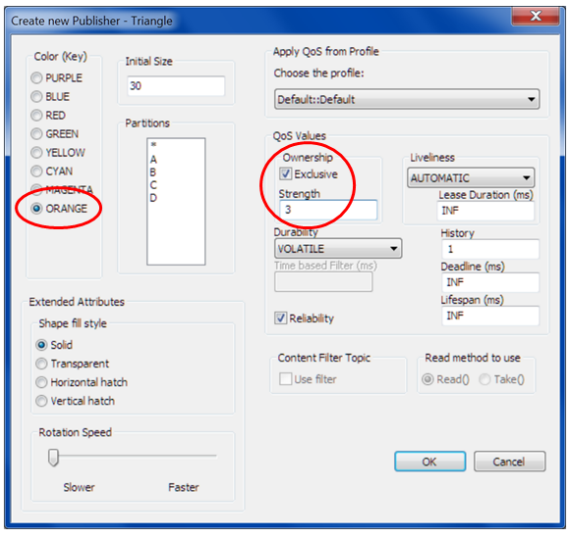
   You should see 6 orange triangles with blue borders moving around in the Subscriber1 canvas. So far, this is similar to the publisher-subscriber exercise.

3. In the third window, create an orange triangle publisher with Exclusive ownership and **Strength** = 3. We will call this window Publisher2.

a. Under **Publish**, click on **Triangle**.

b. In the Create New Publisher window:

- Select **ORANGE** for Color.
- Check **Exclusive** and set **Strength** to 3.
- Click **OK**.

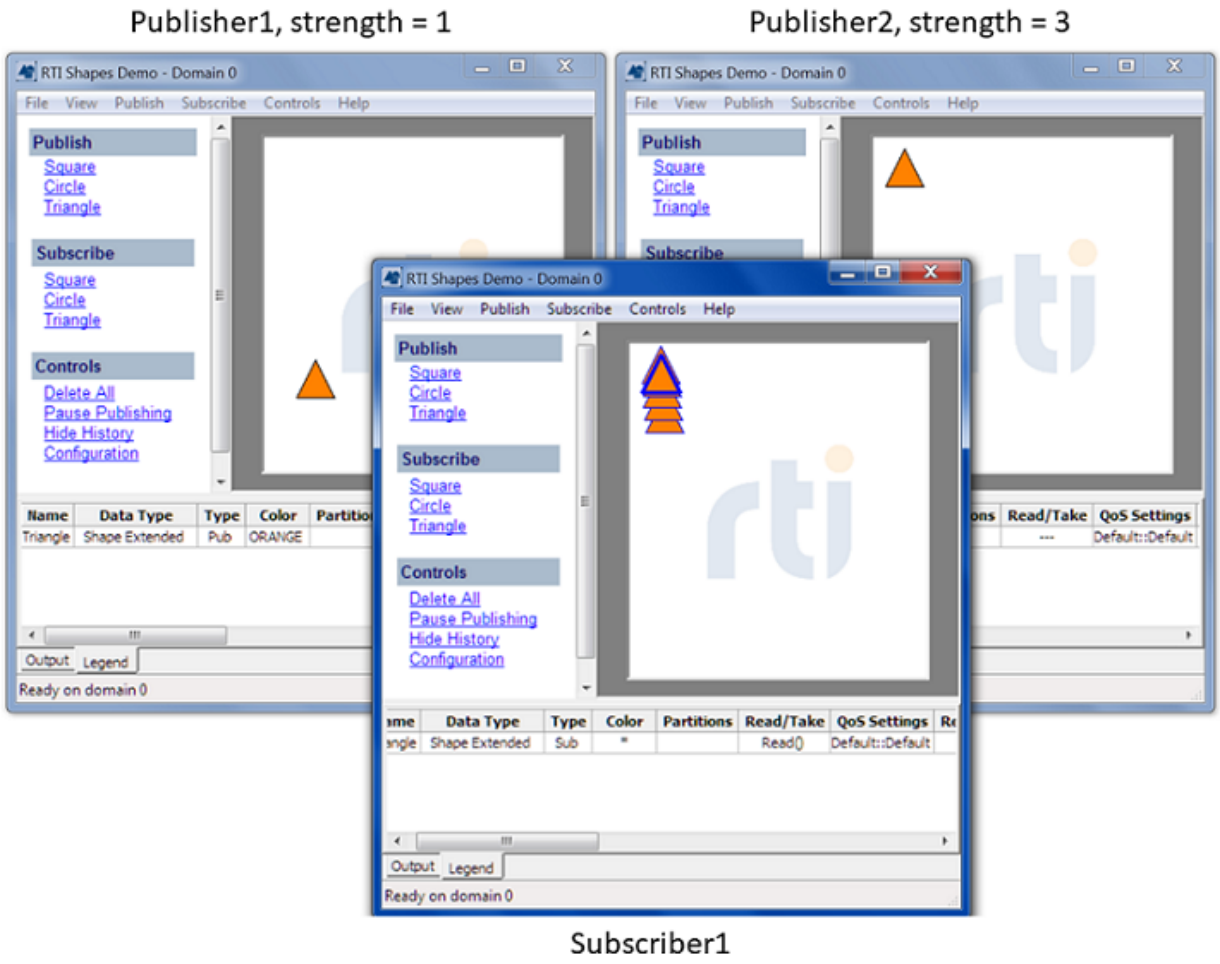


You should see an orange triangle in Publisher2's canvas, as in Figure 4.4: Different Ownership Strengths on the facing page.

4. Use your mouse in Publisher2 to drag the triangle around the canvas. The triangle in Subscriber1 should exactly mirror your mouse movements, because Publisher2 has a higher strength than Publisher1.

5. Click **Delete All** in the **Controls** sub-panel of each of the three demo windows.

Figure 4.4: Different Ownership Strengths



4.4 Failure Detection Example

You may want to detect when the publisher or the network is behaving abnormally and the subscriber hasn't seen updates for an instance within a specified period of time. The Deadline QoS offers a way to do this.

Deadline is a contract between the publisher and the subscriber based on the data rate. The publisher offers to send data at least once in its specified deadline period and the subscriber requests to receive data within its deadline period. If either the subscriber or the publisher misses their deadline, an event callback for "deadline missed" occurs.

At this point, you should have three copies of *Shapes Demo* running, though you will only use two of them for this example. The two copies will be referred to as Publisher1 and Subscriber1.

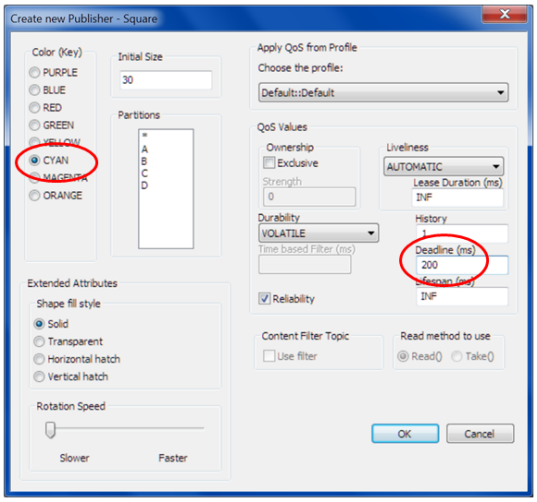
Tip: If you make a mistake during the following steps and need to delete a single publisher or subscriber, select the item in the Legend tab and press the **Delete** key on your keyboard.

1. In Publisher1, create a cyan square publisher, **Deadline** = 200 ms.:

   a. Under **Publish**, click on **Square**.

   b. In the Create New Publisher window:

   - Select **CYAN** for Color.
   - Set **Deadline** to 200 ms.
   - Click **OK**.

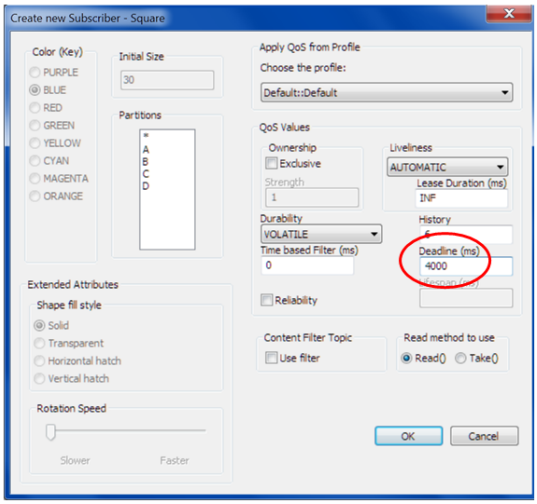


2. Create a square subscriber in Subscriber1 with Deadline = 4000 ms:

   a. Under **Subscribe**, click on **Square**.

   b. In the Create New Subscriber window:

   - Set **Deadline** to 4000 ms.
   - Click **OK**.



   You'll see six cyan squares moving around Subscriber1's canvas. This set of squares mirrors the movement of the cyan square in Publisher1's canvas, along with 5 historical samples.

   Note: The subscriber's deadline must be greater than or equal to the publisher's deadline. If not, an "Incompatible QoS (Deadline) on Square" error message will be displayed in the Output tab of the Subscriber demo application.

3. In Publisher1's **Controls** sub-panel, click **Pause Publishing**.

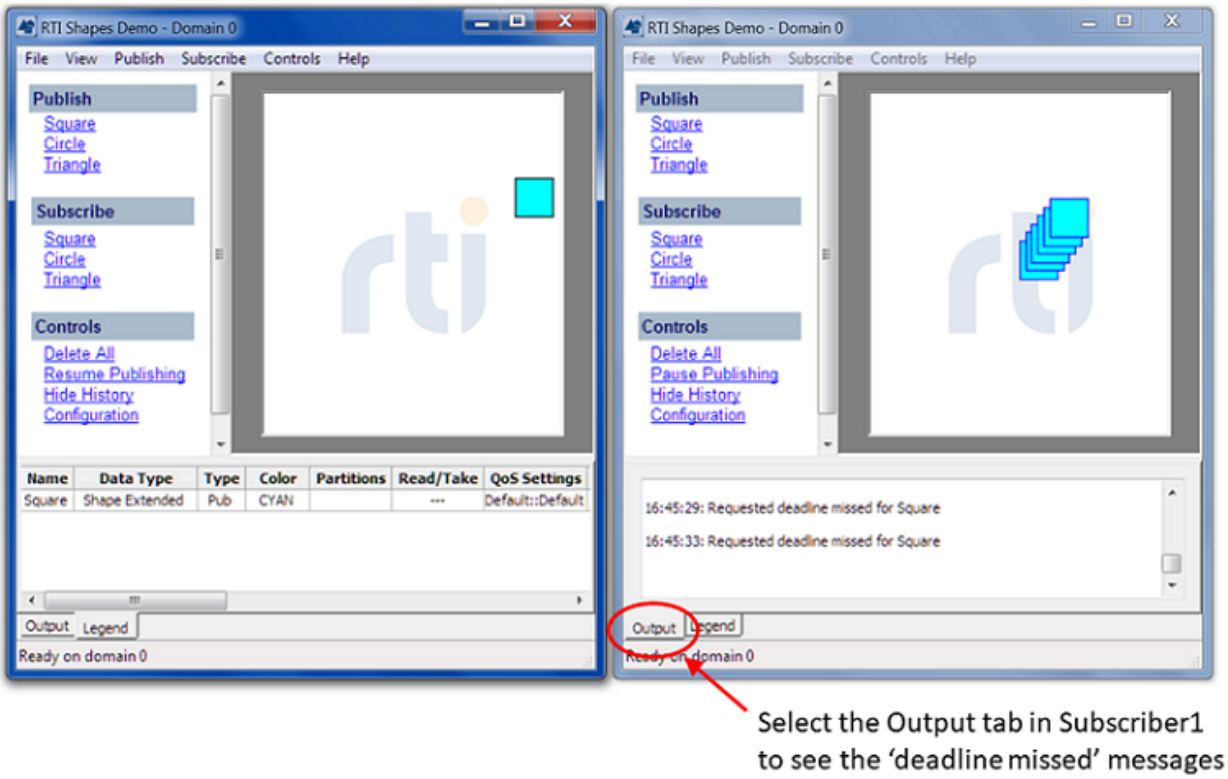
   The cyan square in Subscriber1's canvas should freeze. Note that now all the samples' borders have the same thickness: this indicates that all of them are historical data. In Subscriber1, select the **Output** tab to see messages notifying the application that the promised deadline of 4000 ms has been missed, as seen in Figure 4.5: Missed Deadline on the facing page.

4. Click **Resume Publishing**.

   The cyan squares in Subscriber1's canvas will start moving again, mirroring the movement in Publisher1's canvas.

5. Click **Delete All** in the **Controls** sub-panel of each demo window.

Figure 4.5: Missed Deadline



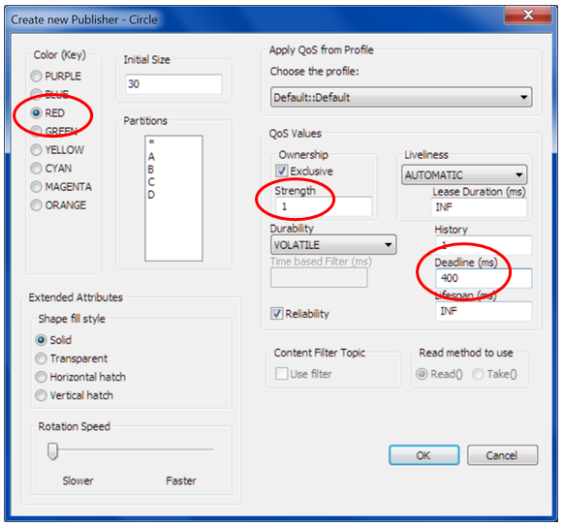
4.5 Failover Example

In most mission-critical systems, there are failover mechanisms to handle unexpected behaviors. In this exercise, we combine the previous two exercises to illustrate hot-failover behavior where the "primary" publisher goes down and the subscriber immediately detects the loss and starts taking data from the "secondary" publisher.

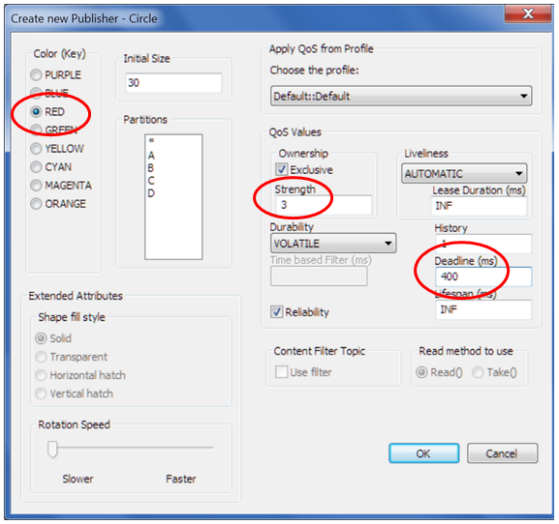
At this point, you should have three copies of *Shapes Demo* running, referred to as Publisher1, Publisher2 and Subscriber1.

Tip: If you make a mistake during the following steps and need to delete a single publisher or subscriber, select the item in the Legend tab and press the **Delete** key on your keyboard.

1. In Publisher1, create a red circle publisher with **Exclusive** Ownership, **Strength** = 1, **Deadline** = 400 ms:
    a. In Publisher1, under **Publish**, click on **Circle**.
    b. In the Create New Publisher window:
        - Select **RED** for Color.
        - Check **Exclusive**.
        - Set **Strength** to 1.
        - Set **Deadline** to 400 ms.
        - Click **OK**.



2. In Publisher2, create a red circle publisher with **Exclusive** Ownership, **Strength** = 3, **Deadline** = 400 ms:
    a. Under **Publish**, click on **Circle**.
    b. In the Create New Publisher window:
        - Select **RED** for Color.
        - Check **Exclusive**.
        - Set **Strength** to 3.
        - Set **Deadline** to 400 ms.
        - Click **OK**.

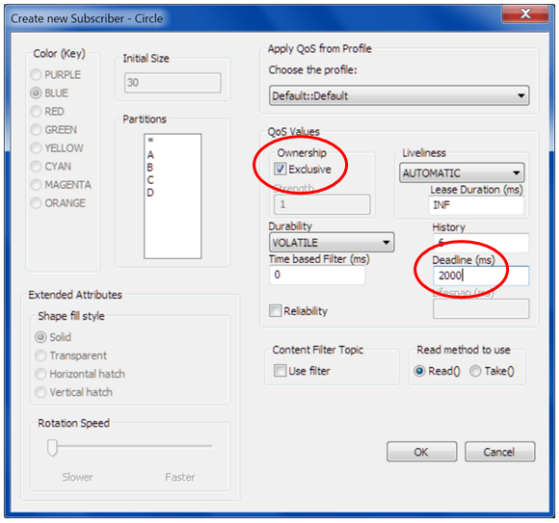


3. In Subscriber1, create a circle subscriber, **Exclusive** selected, **Deadline** = 2000 ms.

   a. Under **Subscribe**, click on **Circle.**

   b. In the Create New Subscriber window:

      - Check **Exclusive**.
      - Set **Deadline** to 2000 ms.
      - Click **OK**.

   In the subscriber canvas, you should see red circles that mirror the movement of the one in Publisher2. This happens because Publisher2's circle has a higher strength that Publisher1's. The deadline setting for the subscriber is the time at which the subscriber application will "fail-over" to the lower strength publisher application.

4. In Publisher2, click on **Pause Publishing**.

   After 2000 ms, Subscriber1 will show a "requested deadline missed" message in its Output tab and at the same time, fail over to display the movements of the red circle in Publisher1.

   Publisher2 initially had exclusive ownership of the red circle instance because it had a higher strength. However, this ownership was lost to the lower-strength Publisher1 when the subscriber missed a deadline. This is especially useful if a publisher is unable to gracefully shutdown and relinquish its ownership.

5. In Publisher2, click on **Resume Publishing**.

   Subscriber1's red circle should immediately switch to tracking the movements of Publisher2.

6. Click **Delete All** in the **Controls** sub-panel of each demo window.

4.6 Extensible Types Examples

Data models often need to evolve. In a deployed system, you might want to deploy new applications that can handle additional attributes in the data model, yet maintain compatibility with already deployed applications—without making any changes. The Extensible Types feature is designed to handle these situations: applications using different but compatible data-types can still communicate. The *Shapes Demo* application uses two different data types to demonstrate this scenario. *Shapes Demo* can publish and subscribe to either a “Shapes Extended” data type (the default) or a more basic “Shape” data type. The difference between these types is that the Shapes Extended type includes two more pieces of information: a fill-pattern and a rotation speed.

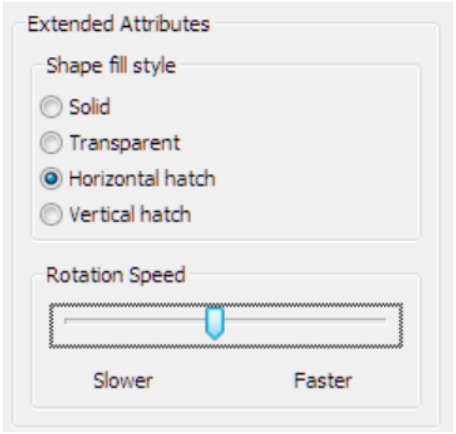
In addition to the QoS settings that you will experiment with in these exercises, there is another QoS specific to Extensible Types (TypeConsistencyEnforcementQosPolicy) that can further customize the

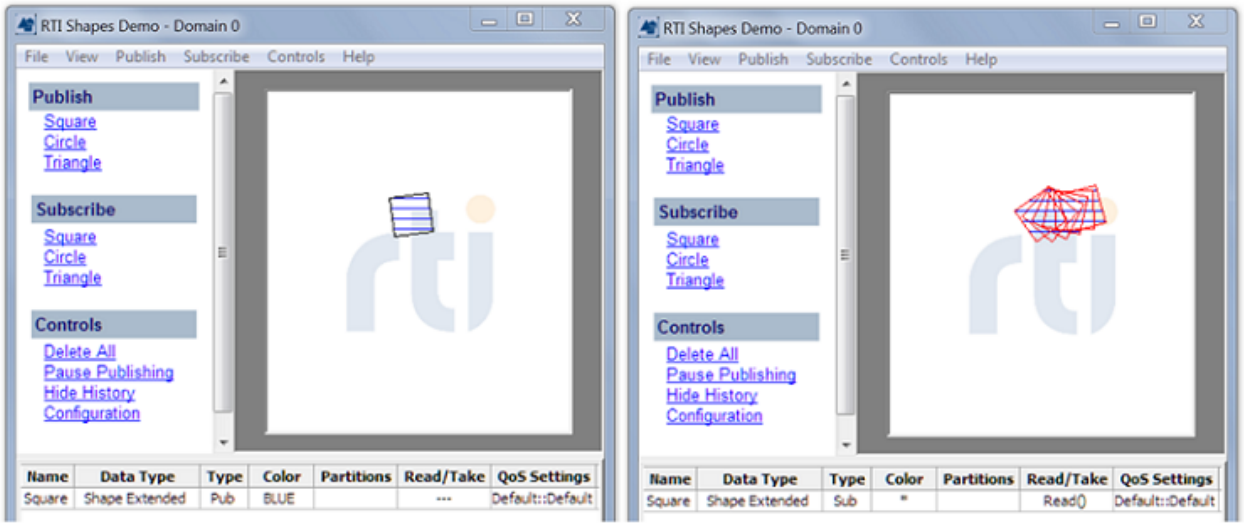
behavior of applications when using Extensible Types. For details, see the *RTI Connext DDS Core Libraries Getting Started Guide Addendum for Extensible Types*.

At this point, you should have three copies of *Shapes Demo* running, referred to as Publisher1, Publisher2 and Subscriber1. All are using the Shape Extended data type by default.

4.6.1 Introduction to the Shape Extended Type

1. Publish a Square in Publisher1. In the publish screen, choose the horizontal hatch pattern and set the rotation speed to middle setting.
2. Subscribe to Squares in Subscriber1. In the subscriber, you will see the shape with the selected pattern, rotating at the selected speed.





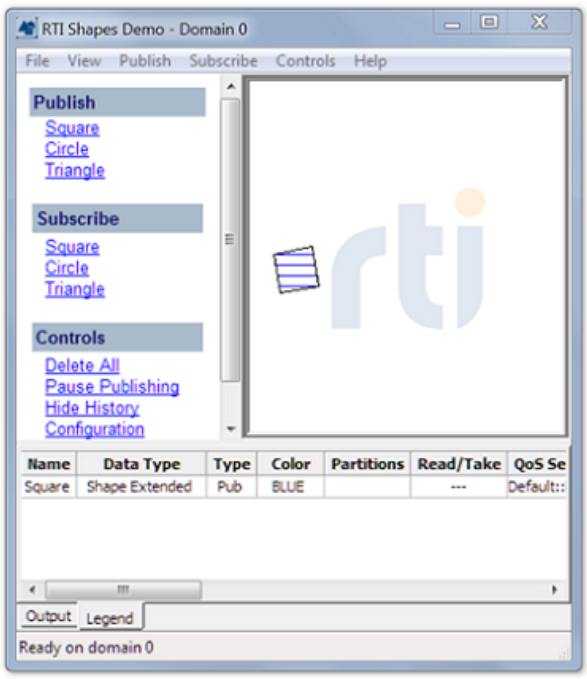
Publisher1 Subscriber1

3. Feel free to repeat with other shapes, fill patterns, and speeds.
4. Select **Delete All** in each instance of *Shapes Demo*.

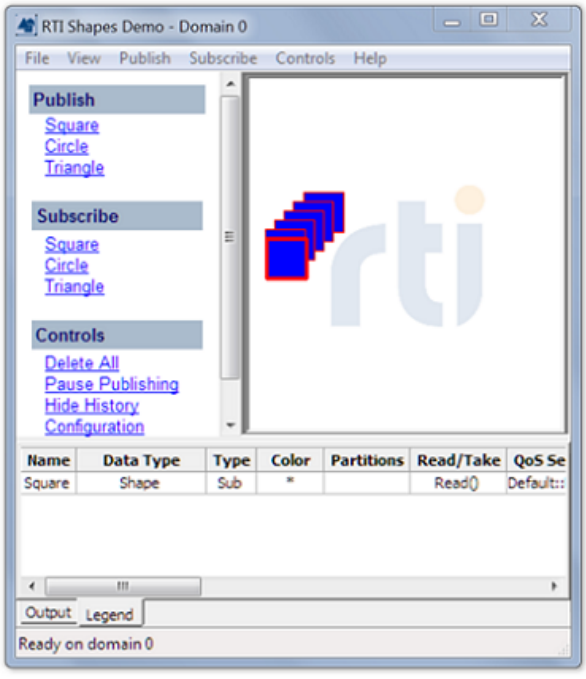
4.6.2 Publishing Extended Type, Subscribing to Basic Type

This scenario simulates the situation where new applications are publishing data with extra information using an extended data model, but there are existing applications that only need to subscribe to the original, basic data model (and in fact, don't even have the logic to deal with extra attributes in the newer, extended model).

1. In Publisher 1 (which is using the Shape Extended type by default), publish a blue square. Select the horizontal hatch fill-pattern and a medium rotation speed.
2. In Subscriber1's Configuration dialog, press **Stop**, select the "Shape" data type, press **Start**.
3. In Subscriber1, subscribe to squares.
4. In Publisher 1, you should see a square with the selected pattern, rotating at the selected speed. In Subscriber 1, you should see a blue square that does *not* have the pattern and is *not* rotating.



Publishing "Shape Extended"



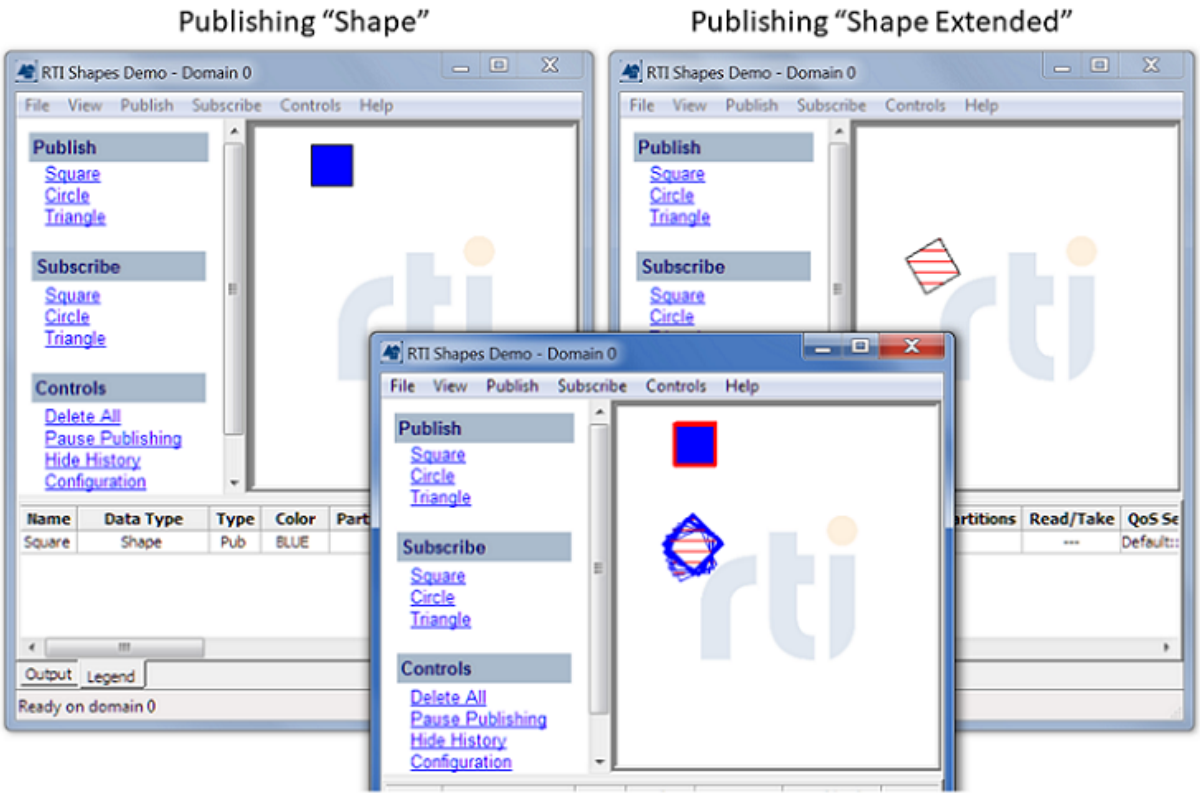
Subscribing to Squares,
configured to use "Shape" data type

5. Select **Delete All** in both instances of **Shapes Demo**.

4.6.3 Publishing Original and Extended Types, Subscribing to Extended Type

This scenario simulates the situation where deployed applications are publishing data using the old model and new applications are receiving data of both the original and extended data types.

1. In Publisher1's Configuration dialog, press **Stop**, select the "Shape" data type, press **Start**.
2. In Publisher 1, publish a blue square.
3. In Publisher2 (using the Shape Extended type by default), publish a red square with the horizontal hatch fill-pattern and a medium rotation speed.
4. In Subscriber1 (using the Shape Extended type by default), subscribe to squares. You should see that Subscriber1 is receiving both types of squares, as seen below.



4.7 Security Examples

*RTI Security Plugin*s introduce a robust set of security capabilities, including authentication, encryption, access control and logging. Secure multicast support enables efficient and scalable distribution of data to many subscribers. Performance is also optimized by fine-grain control over the level of security applied to each data flow, such as whether data confidentiality, or just data integrity is required.

RTI Shapes Demo is configured to run Security examples out of the box, with no configuration from your side other than selecting the correct profile.

<Shapes Demo installation directory>/resource/xml includes the following files, which are used by *Shapes Demo* to configure/enable security:

- **RTI_SHAPES_DEMO_GOVERNANCE_MAX.xml**
- **RTI_SHAPES_DEMO_GOVERNANCE_MIN.xml**
- **RTI_SHAPES_DEMO_PERMISSIONS.xml**
- **signed/RTI_SHAPES_DEMO_GOVERNANCE_MAX.p7s**
- **signed/RTI_SHAPES_DEMO_GOVERNANCE_MIN.p7s**
- **signed/RTI_SHAPES_DEMO_PERMISSIONS.p7s**

Note: Make sure to run every instance of *Shapes Demo* using domain ID 0, since that's what the governance files use:

```
<domain_access_rules>
    <domain_rule>
        <domains>
            <id_range>
                <min>0</min>
            </id_range>
        </domains>
    </domain_rule>
    ...
```

The following sections refer to these profiles (which are in **RTI_SHAPES_DEMO_QOS_PROFILES.xml**):

- **Security::NonSecure**—Baseline profile of the Security QoS library with security disabled.
- **Security::SecureAllowAll**—Security enabled, subscribing and publishing to all topics is allowed.
- **Security::SecureMinimal**—Minimal security profile: does not protect outgoing data from being tainted, does not check that incoming data hasn't been tainted, and unencrypted topics will be communicated with Multicast (no security) participants.
- **Security::SecureDenyPubCircles**—Circles published by this participant will be accepted by secure participants because they have enable_write_access_control set to false for Circle.
- **Security::SecureDenySubSquares**—Prevents this participant from subscribing to Squares.
- **Security::Imposter** (adversarial)—Enables security. Tries to impersonate another peer by using its certificate.
- **Security::WriterTainter** (adversarial)—Simulates a man in the middle tainting data while it's in flight. This profiles applies to a data writer.

- **Security::ReaderMulticast**—Enables multicast in the data reader. This profiles applies to a data reader.

Note: In all the Security profiles, the Triangle and Square topics are encrypted. Circles remain unencrypted.

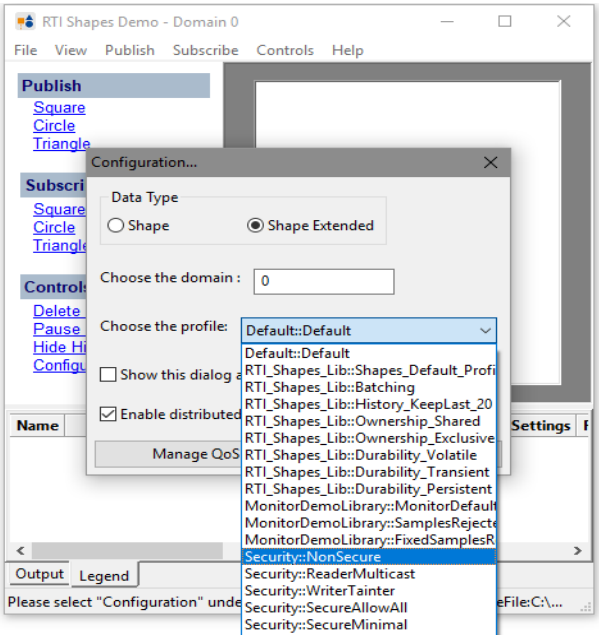
4.7.1 Subscribing to Data from an Application not using Security Plugins

This scenario simulates the situation where we have an application using the security plugins that wants to subscribe to a topic published by an application that is not using the security plugins.

Notes:

- Make sure to run every instance of *Shapes Demo* using domain 0.
- Use the default profile when publishing/subscribing unless otherwise noted.

1. Start *Shapes Demo* and choose the **Security:NonSecure** profile as explained in 3.3.5 Applying QoS from a Profile on page 13. We'll call this instance Publisher1.



Publish blue circles and squares.

Publisher1
NonSecure

2. Start a second instance of *Shapes Demo* using the **Security:SecureAllowAll** profile. We'll call this instance Participant1.

   Tip: To start another instance of *Shapes Demo*, select **File, New Shapes Demo**.

   Subscribe to circles and squares.

   Notice that no shapes appear in Participant1:

Publisher1
NonSecure

Participant1
SecureAllowAll

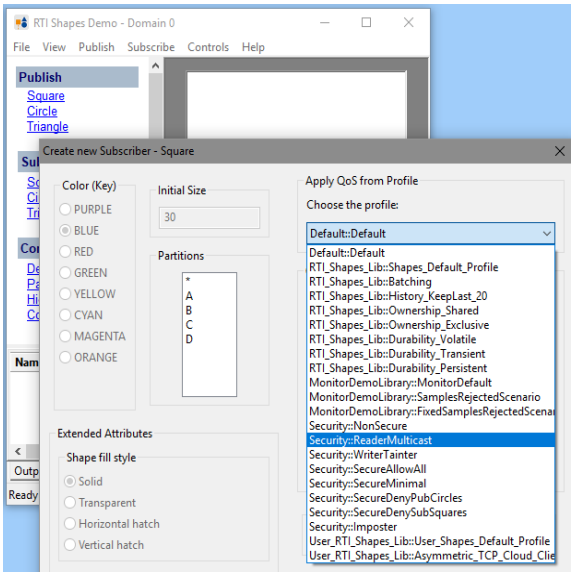
Let's see a snippet of the **RTI_SHAPES_DEMO_GOVERNANCE_MAX.xml** file used by the **Security:SecureAllowAll** profile to see why this is happening:

```
<domain_access_rules>
    <domain_rule>
        ...
        <allow_unauthenticated_participants>false</allow_unauthenticated_participants>
        <enable_join_access_control>true</enable_join_access_control>
        <discovery_protection_kind>ENCRYPT</discovery_protection_kind>
        <liveliness_protection_kind>ENCRYPT</liveliness_protection_kind>
        <rtps_protection_kind>SIGN</rtps_protection_kind>
        ...
    </domain_rule>
</domain_access_rules>
```

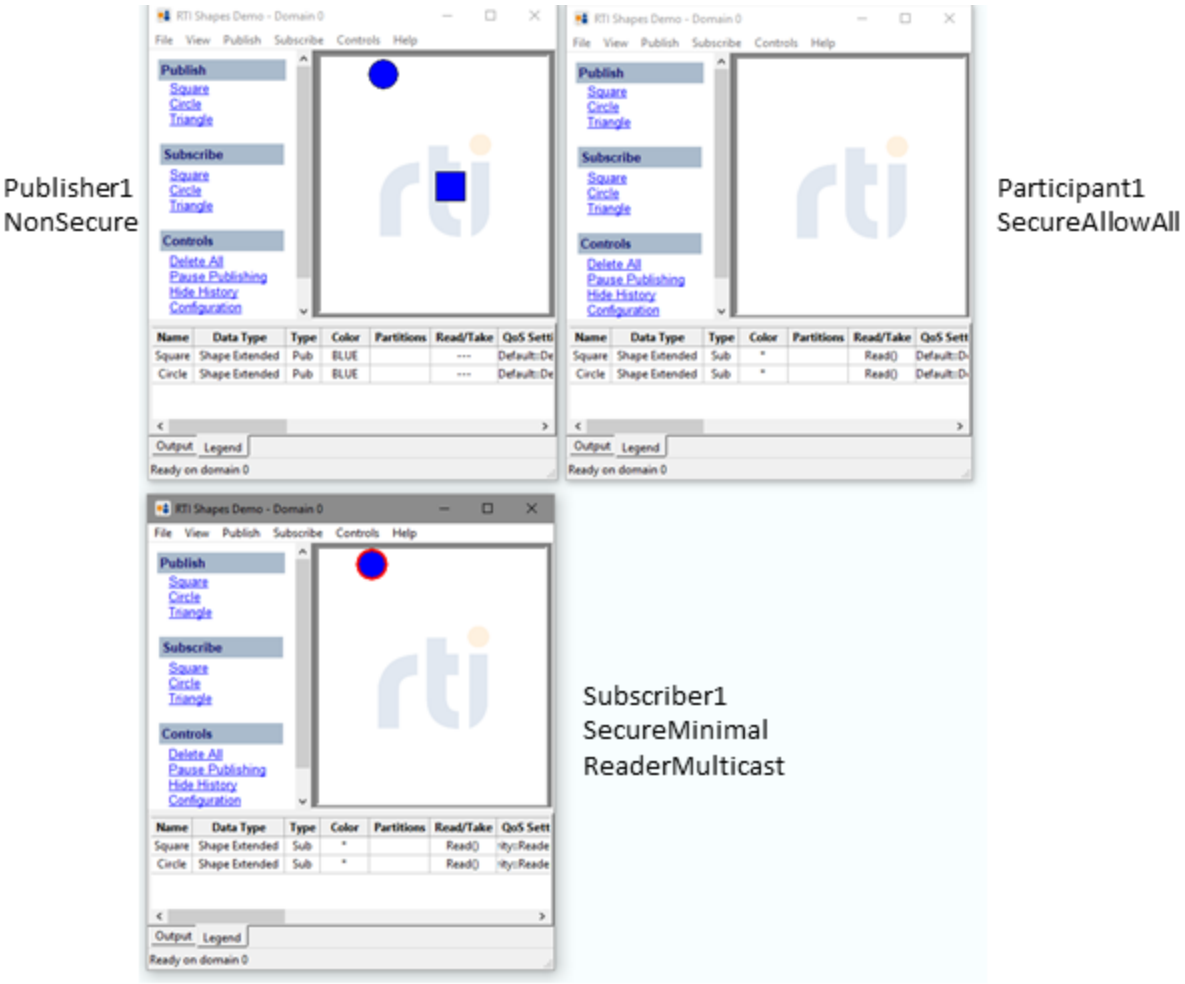
Discovery is configured to be encrypted in the **Security:SecureAllowAll** profile used by Participant1 (the subscriber), but not in the **Security:NonSecure** profile used by Publisher1. These two profiles are not compatible with each other and therefore will not interoperate.

3. Start a third instance of *Shapes Demo* using the **Security:SecureMinimal** profile. We'll call this instance Subscriber1.

   Subscribe to circles and squares using the **Security:ReaderMulticast** profile.



Notice that only blue circles appear in Subscriber1.



Let's take a look at a snippet of **RTI_SHAPES_DEMO_GOVERNANCE_MIN.p7s**, which is used by the **Security:SecureMinimal profile**, to see why only circles are being displayed:

```
<domain_access_rules>
    <domain_rule>
        ...
        <allow_unauthenticated_participants>true</allow_unauthenticated_participants>
        <enable_join_access_control>false</enable_join_access_control>
        <discovery_protection_kind>NONE</discovery_protection_kind>
        <liveliness_protection_kind>NONE</liveliness_protection_kind>
        <rtps_protection_kind>NONE</rtps_protection_kind>
        <topic_access_rules>
            <topic_rule>
                <topic_expression>Circle*</topic_expression>
                <enable_discovery_protection>false</enable_discovery_protection>
                <enable_liveliness_protection>false</enable_liveliness_protection>
                <enable_read_access_control>false</enable_read_access_control>
                <enable_write_access_control>false</enable_write_access_control>
```

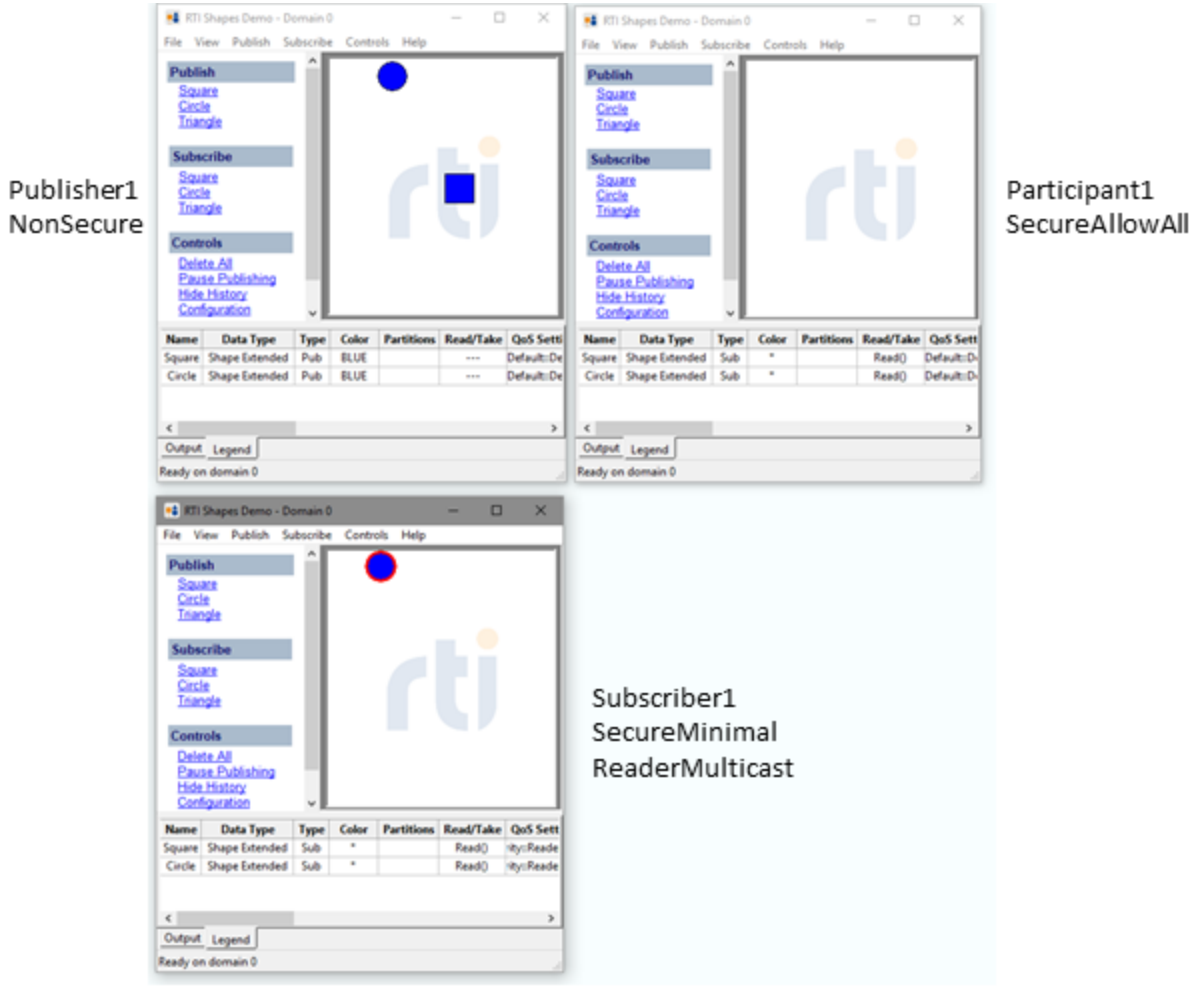
```
                <metadata_protection_kind>NONE</metadata_protection_kind>
                <data_protection_kind>NONE</data_protection_kind>
            </topic_rule>
            <topic_rule>
                <topic_expression>*</topic_expression>
                <enable_discovery_protection>true</enable_discovery_protection>
                <enable_liveliness_protection>true</enable_liveliness_protection>
                <enable_read_access_control>true</enable_read_access_control>
                <enable_write_access_control>true</enable_write_access_control>
                <metadata_protection_kind>ENCRYPT</metadata_protection_kind>
                <data_protection_kind>ENCRYPT</data_protection_kind>
            </topic_rule>
        </topic_access_rules>
    </domain_rule>
</domain_access_rules>
```

The reader cannot process squares because both the data and metadata for the "Squares" topic are configured to be encrypted. This is accomplished by setting **metadata_protection_kind = ENCRYPT** and **data_protection_kind = ENCRYPT** in the Governance file seen above.

When defining a topic_rule, order matters. The first rule will be used to try to match with the DataWriter/DataReader topic that has been published/subscribed to. If the topic doesn't match the first rule, the second one will be used, etc. This process continues until it finds a match.

4. Using Participant1, publish green circles and squares.

   Subscriber1 won't display either shape from Participant1 because the **rtps_protection_kind** settings in **Security:SecureAllowAll** and **Security:SecureMinimal** are incompatible.



Participant1 (using **Security:SecureAllowAll)** rejects messages from Subscriber1 (using **Security:SecureMinimal**) because it is configured to reject RTPS messages that have not been signed (**rtps_protection_kind** = SIGN). This prevents discovery from succeeding at the participant level. Let's see their **rtps_protection_kind** settings in these snippets:

RTI_SHAPES_DEMO_GOVERNANCE_MIN.xml (used by **Security:SecureMinimal**, Subscriber1):

```
...
<rtps_protection_kind>NONE</rtps_protection_kind>
...
```

RTI_SHAPES_DEMO_GOVERNANCE_MAX.xml (used by **Security:SecureAllowAll**, Participant1):

```
...
<rtps_protection_kind>SIGN</rtps_protection_kind>
...
```

5. Start a fourth instance of *Shapes Demo* using the **Security:SecureAllowAll** profile. We'll call this instance Subscriber2.

   Subscribe to circles and squares using the **Security:ReaderMulticast** profile.

   In Subscriber2, you will see the green circle and square published by Participant1, because these Participants use compatible security profiles.

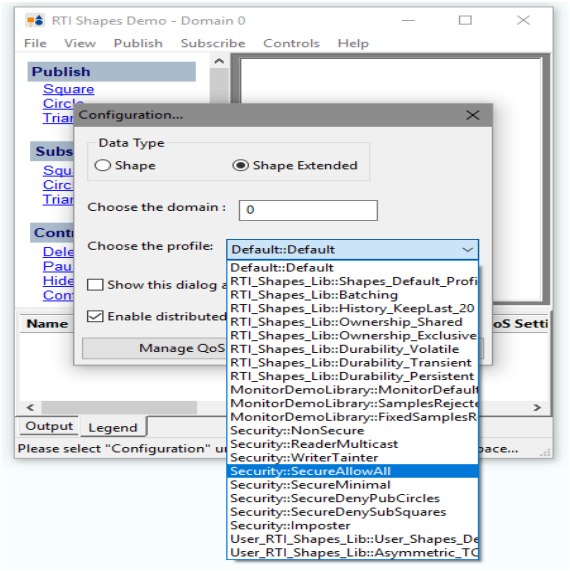
4.7.2 Access Control and Authentication

This scenario simulates the situation where we have various applications using the security plugins and we want to prevent receiving certain topics while using authentication.

Notes:

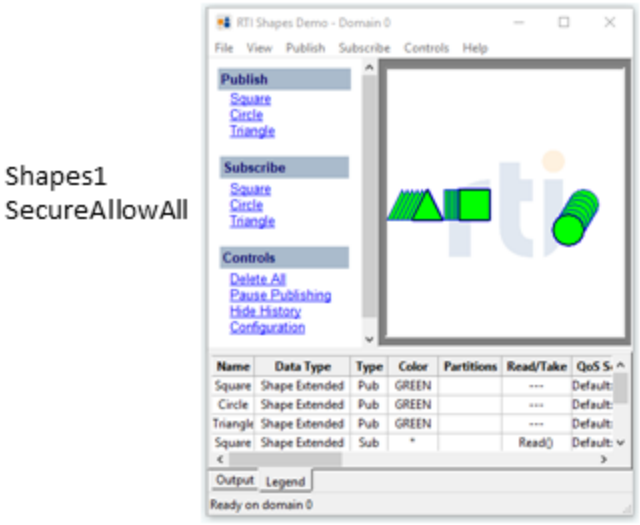
- Make sure to run every instance of *Shapes Demo* using domain 0.
- Use the default profile when publishing/subscribing unless otherwise noted.

1. Start an instance of *Shapes Demo* using the **Security::SecureAllowAll** profile as explained in 3.3.5 Applying QoS from a Profile on page 13. We'll call this instance Shapes1.



   Publish green squares, circles, and triangles.

   Subscribe to squares, circles, and triangles.

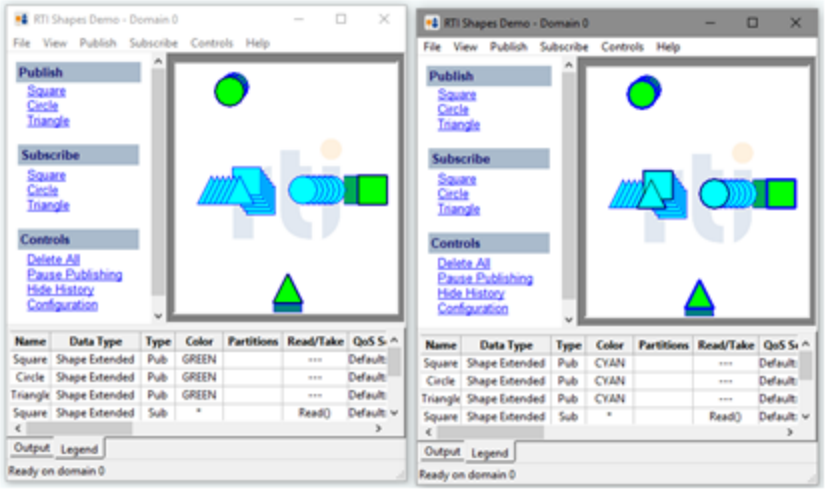


2. Start a second instance of *Shapes Demo* using the **Security::SecureDenyPubCircles** profile. We'll call this instance Shapes2.

   Tip: To start another instance of *Shapes Demo*, select **File, New Shapes Demo**.

   Publish cyan squares, circles, and triangles.

   Subscribe to squares, circles, and triangles.

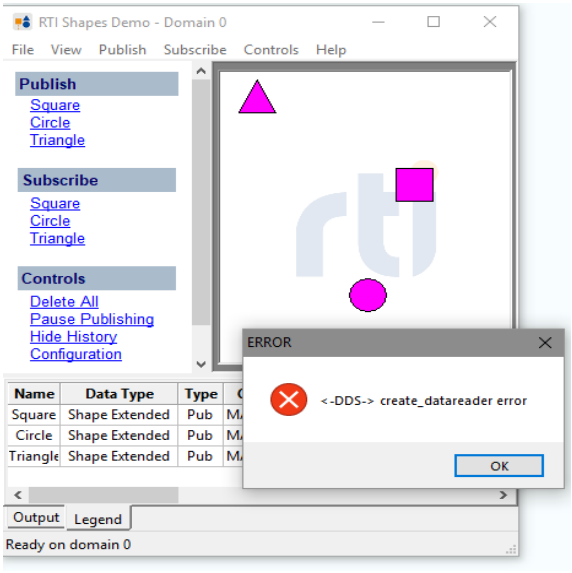


3. Start a third instance of *Shapes Demo* using the **Security::SecureDenySubSquares** profile. We'll call this instance Shapes3.

   Publish magenta squares, circles, and triangles.

Subscribe to squares, circles, and triangles.

Notice that subscribing to squares causes an error:

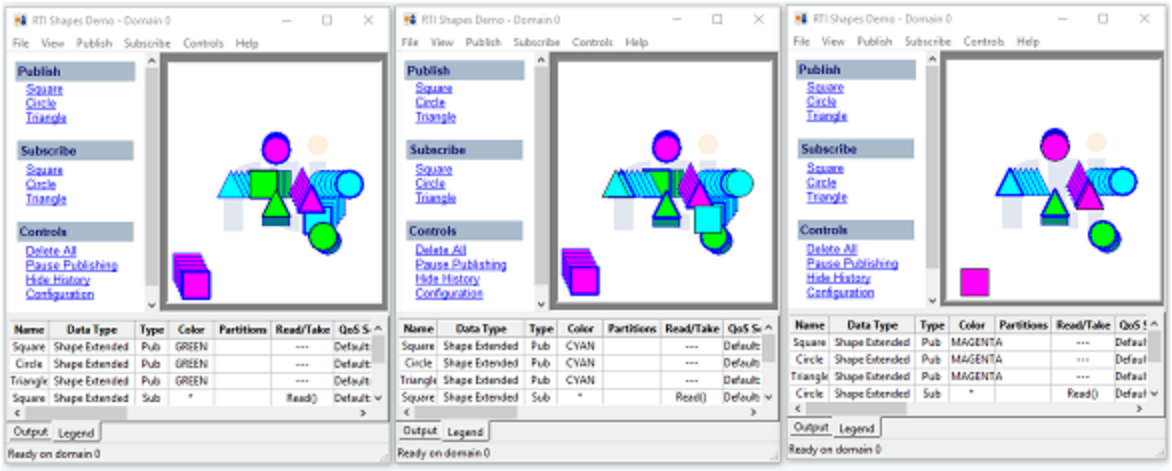


This error is because the profile will not allow you to create DataReaders for the topic 'Square'. We can see this in the <deny_rule> section of the profile:

```
<deny_rule>
    <domains>
        <id>0</id>
    </domains>
    <subscribe>
        <topics>
            <topic>Square*</topic>
        </topics>
    </subscribe>
</deny_rule>
```

Click **OK** on the error and continue to subscribe to circles and triangles.

Now the first three instances should look like this:



Shapes1
SecureAllowAll

Shapes2
SecureDenyPubCircles

Shapes3
SecureDenySubSquares

To recap:

- Shapes1 displays all the shapes that are being published, because **Security::SecureAllowAll** is configured to allow subscribing to all Topics.
- Shapes2 also displays all the shapes, because **Security::SecureDenyPubCircles** is configured to allow subscribing to all Topics. Circles published will be accepted by secure participants because **enable_write_access_control** is set to false for circles.
- Shapes3 displays all the shapes except squares, because **Security::SecureDenySubSquares** prevents subscribing to squares.

4. Start a fourth instance of *Shapes Demo* using the **Security::SecureMinimal** profile. We'll call this instance Subscriber1.

   Subscribe to squares, circles, and triangles.

   Subscriber1 won't display any shapes, because **Security::SecureMinimal** is not configured to use authentication.
5. Start a fifth instance of *Shapes Demo* using the **Security::Imposter** profile. We'll call this instance ShapesImposter.

   Publish orange squares, circles, and triangles.

   Subscribe to squares, circles, and triangles.

ShapesImposter
Imposter

This instance will only receive its own shapes. The other instances will not communicate with it. Authentication is failing because ShapesImposter is using (impersonating) a wrong combination of private key and public certificates.

Let's take a look at the **Security::Imposter** profile in the **RTI_SHAPES_DEMO_QOS_PROFILES.xml** file to see why:

```
<qos_profile name="Imposter" base_name="SecureAllowAll">
    <participant_qos>
        <property>
            <value>
                <element>
                    <name>dds.sec.auth.identity_certificate</name>
                    <value>
                        file:$(NDDSHOME)/resource/cert/RTI_SHAPES_DEMO_PEER_2_CERT.pem
                    </value>
                </element>
                <element>
                    <name>dds.sec.auth.private_key</name>
                    <value>
                        file:$(NDDSHOME)/resource/cert/RTI_SHAPES_DEMO_PEER_3_KEY.pem
                    </value>
                </element>
                <element>
                    <name>dds.sec.auth.password</name>
                    <value>SnVsaWFuRTEx</value>
                </element>
            </value>
        </property>
    </participant_qos>
```

```
</qos_profile>
```

4.7.3 Data Integrity

This scenario simulates the situation where there is a "man in the middle" tainting data while it's in flight.

Notes:

- Make sure to run every instance of *Shapes Demo* using domain 0.
- Use the default profile when publishing/subscribing unless otherwise noted.

1. Start an instance of *Shapes Demo* using the **Security:NonSecure** profile as explained in 3.3.5 Applying QoS from a Profile on page 13. We'll call this instance Publisher1.

   Publish a blue square.

   Publish a blue triangle using the **Security:WriterTainter** profile.

Publisher1
NonSecure
Squares: Default
Triangles: WriterTainter

2. Start a second instance of *Shapes Demo* using the **Security::NonSecure** profile. We'll call this instance Subscriber1.

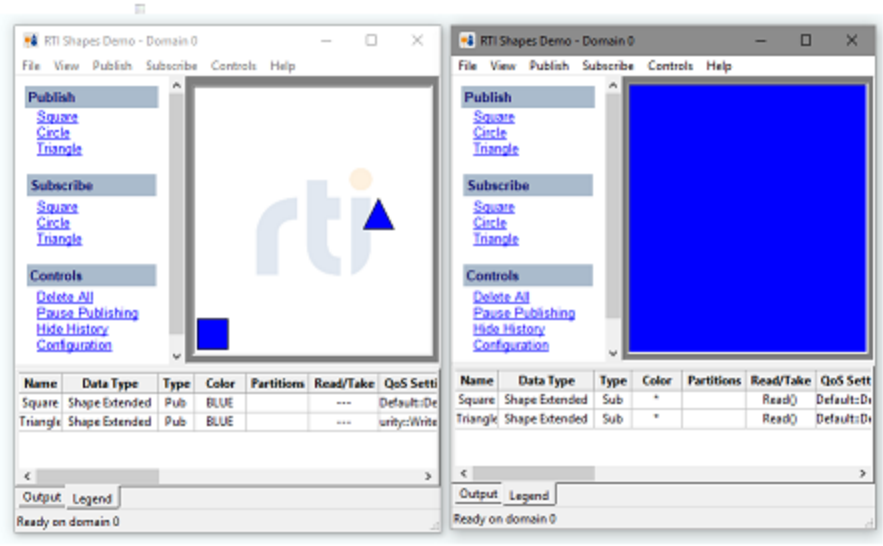
   Tip: To start another instance of *Shapes Demo*, select **File, New Shapes Demo**.

   Subscribe to squares.

   You will see the blue squares.

   Now subscribe to triangles.

You will see that the blue triangles are corrupted. The shape does not match the triangle that is published in Publisher1. In fact, the shape may be so big that it takes up the whole area, as seen here:



Publisher1
NonSecure
Squares: Default
Triangles: WriterTainter

Subscriber1
NonSecure

Let's take a look at the **WriterTainter** profile to see why the triangle is being corrupted:

```
<qos_profile name="WriterTainter">
    <datawriter_qos>
        <property>
            <value>
                <element>
                    <name>dds.data_writer.cryptography.taint_data</name>
                    <value>true</value>
                </element>
            </value>
        </property>
    </datawriter_qos>
</qos_profile>
```

The DataWriter is using a property **dds.data_writer.cryptography.taint_data**, which is used to simulate tainted data. This means the data will be modified after *Security Plugins* generates the GMAC for the data.

3. Start a third instance of *Shapes Demo* using the **Security:SecureMinimal** profile. We'll call this instance Publisher2.

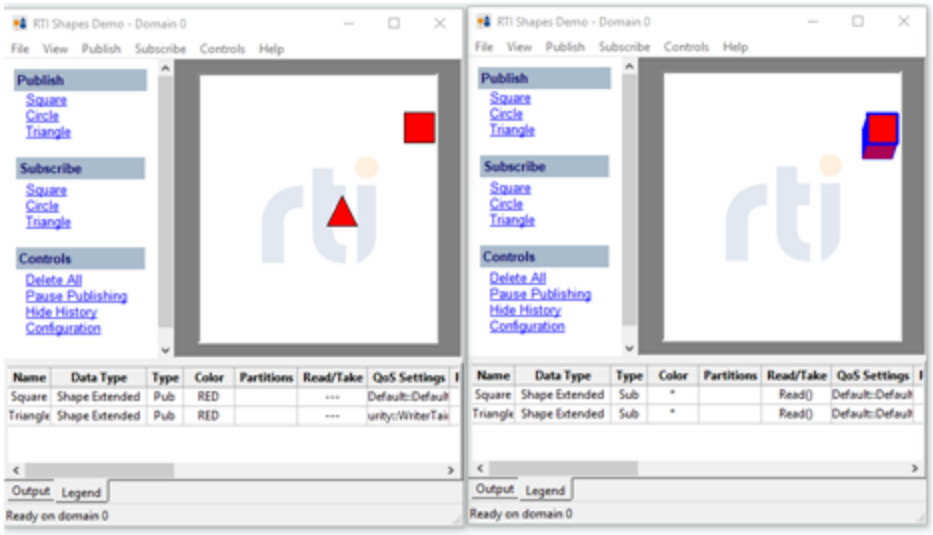
Publish red squares.

Publish red triangles using the **Security::WriterTainter** profile.

4. Start a fourth instance of *Shapes Demo* using the **Security::SecureMinimal** profile. We'll call this instance Subscriber2.

   Subscribe to squares and triangles.

   Only the red square will appear because the triangle has been tainted. Subscriber2 will detect that the data has been modified after the data was signed on origin, so it will drop it.



Publisher2
SecureMinimal
Squares: Default
Triangles: WriterTainter

Subscriber2
SecureMinimal

Chapter 5 More Experiments

Please feel free to experiment and run tests using the other QoS options in the Create New Subscriber and Create New Publisher windows. Described below are a few other interesting behaviors to test.

5.1 Content-Filtered Topics Example

A content-filtered topic is a very useful feature if you want to filter data received by the Subscriber. It also helps to control network and CPU usage on the subscriber side because only data that is of interest to the subscriber is sent.

For example, assume your application is a radar monitor that draws flights detected within a 20-mile radius. The application can subscribe to the track data with a content filtered topic for a 20-mile radius on the coordinates of all flights. With the filter, only coordinates that are within the 20-mile radius will be sent to the application.

1. Start two copies of *Shapes Demo,* which we will call Publisher1 and Subscriber1. If you are reusing demo windows from a previous section, delete any existing publishers and subscribers (under Controls, click **Delete All**.)
2. In Publisher1, create a circle publisher (any color):
    a. Under **Publish**, click on **Circle**.
    b. In the Create New Publisher window, click **OK**.

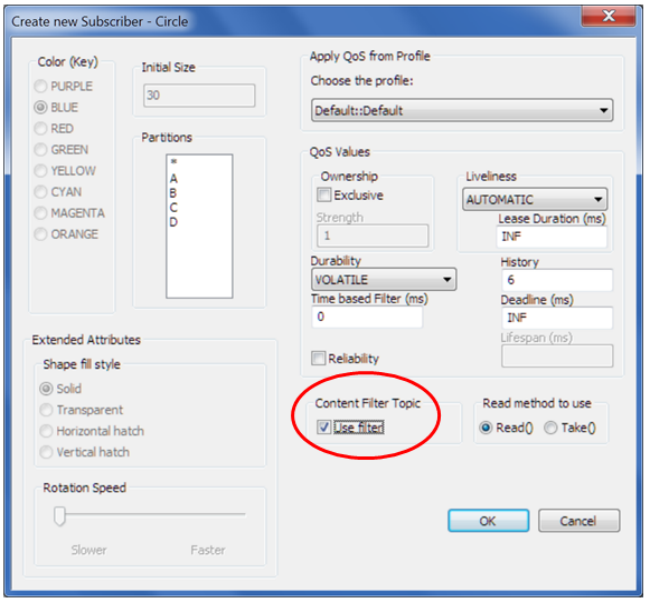
3. In Subscriber1, create a circle subscriber with a content filtered topic:

   a. Under **Subscribe**, click on **Circle**.

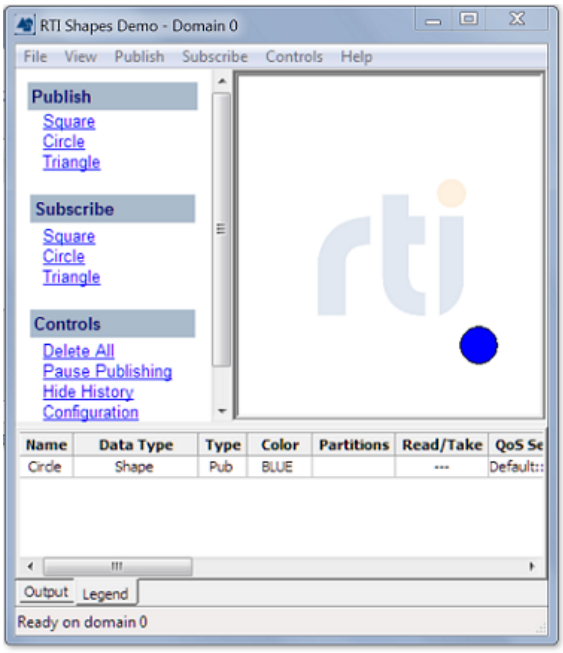
   b. In the Create New Subscriber window:

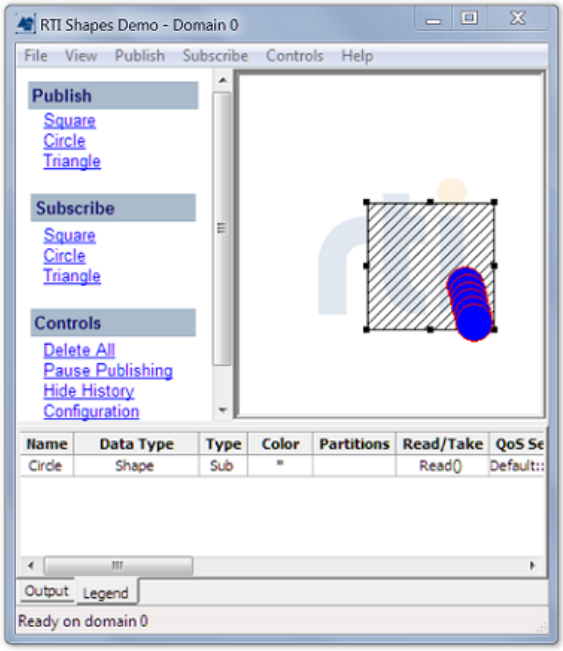
      - Check **Use filter**.
      - Click **OK**.

   You will see a shaded rectangle appear in the subscriber canvas. This is the filter for the coordinates of the Circle topic. The subscriber will receive position data for the Circle only when it is with the area defined by the content filter.



4. To see the effect of dynamic filters, use your mouse to move and resize the shaded area in Subscriber1.



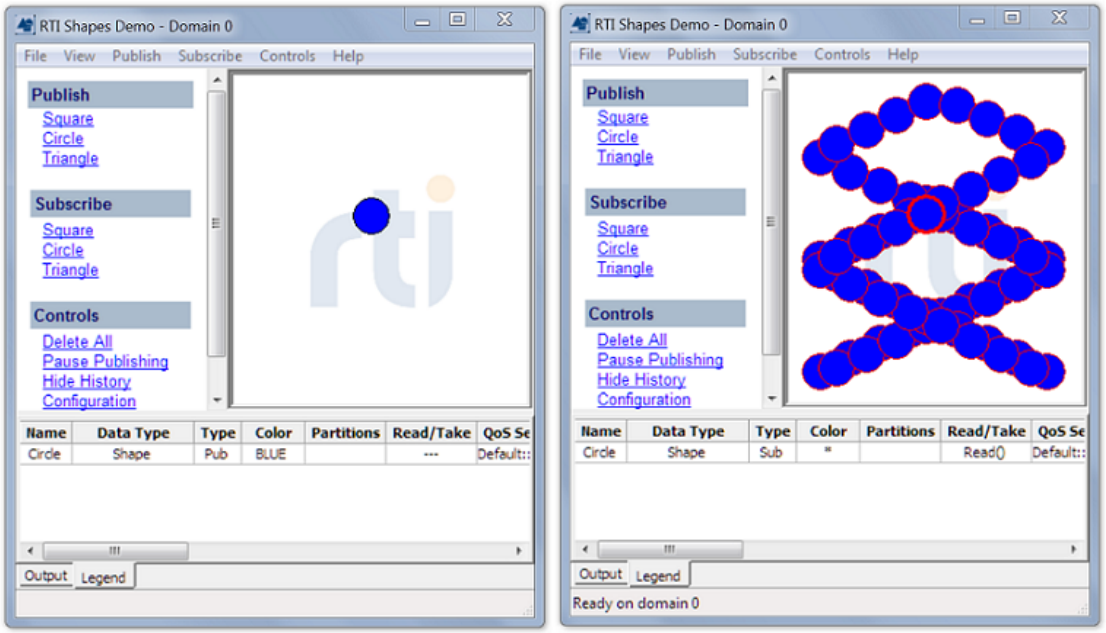


5.2 Lifespan Example

The Lifespan QoS controls how long data samples are considered valid. You can use it to prevent sending data that is considered too old to be valid. The default setting is an infinite duration, meaning the data will never 'expire.'

1. Start two copies of *Shapes Demo,* which we will call Publisher1 and Subscriber1. If you are reusing demo windows from a previous section, delete any existing publishers and subscribers (under Controls, click **Delete All**.)
2. In Publisher1, create a circle publisher (any color) with **History** = 100, **Lifespan** = 1000 ms.:
   a. Under **Publish**, click on **Circle**.
   b. Set **History** to 100 and **Lifespan** to 1000 ms.
   c. Click **OK**.
3. In Subscriber1, create a circle subscriber with History = 100:
   a. Under **Subscribe**, click on **Circle**.
   b. Set **History** to 100.
   c. Click **OK**.
4. Drag the shape around on Publisher1's canvas.

   On Subscriber1's canvas, you will see a "shadow" of objects printed out in a continuous pattern. The shadow is caused by the subscriber showing the last 100 data samples from the publisher's history queue.
5. In Publisher1, click **Pause Publishing**.
6. In Subscriber1, notice that the samples disappear as they time out. Experiment by increasing the Lifespan setting for the publisher. The longer the Lifespan, the longer it takes for the samples to disappear when you pause publishing.

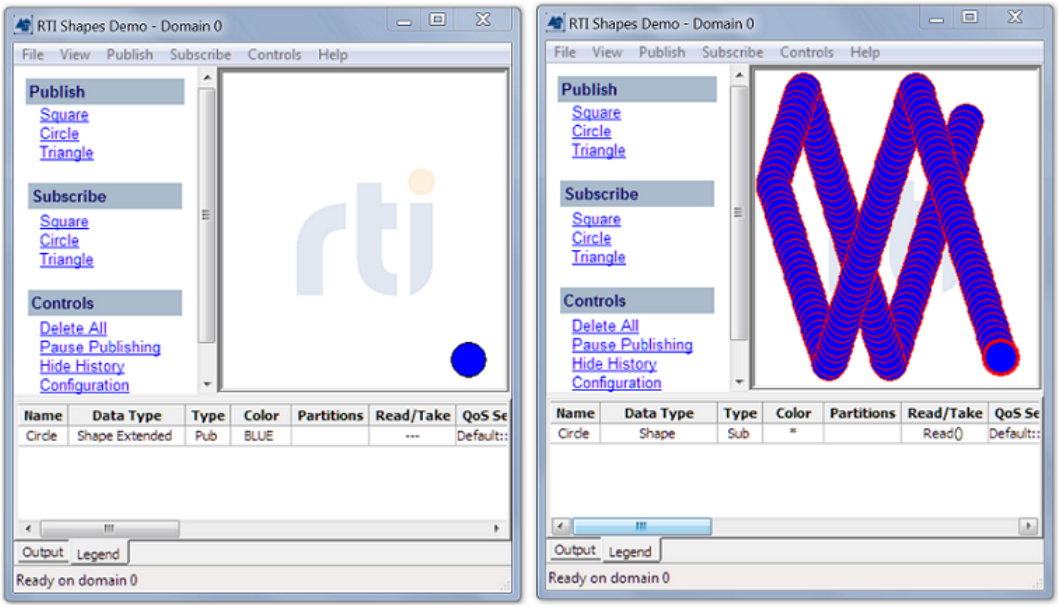


5.3 Reliability and Durability Example

In a dynamic system, you may want late-joining nodes to get the data that was sent before the nodes connected to the network. For example, suppose you need to initialize the state of these late-joining nodes and don't want to be continually sending the state just in case some node joins late. The Durability QoS provides late-joining nodes with the ability to get previously sent data.

1. Start two copies of *Shapes Demo,* which we will call Publisher1 and Subscriber1. If you are reusing demo windows from a previous section, delete any existing publishers and subscribers (under Controls, click **Delete All**.)
2. In Publisher 1, create a circle publisher (any color) with Transient Local Durability, Reliability, and History = 200.
    a. Under **Publish**, click on **Circle**.
    b. In the Create New Publisher window:
        - Use the drop-down list box to change Durability to **Transient Local**.
        - Set **History** to 200.
        - Click **OK**.
3. Wait for a bit.
4. In Subscriber1, create a circle subscriber with Transient-Local Durability, Reliability and History = 200.
    a. Under **Subscribe**, click on **Circle**.
    b. In the Create New Subscriber window:
        - Use the drop-down list box to change Durability to **Transient Local**.
        - Check **Reliability**.
        - Set **History** to 200.
        - Click **OK**.
5. Watch the Subscriber canvas. You will see a "shadow" of objects printed out in a continuous pattern. The shadow results from the subscriber showing the last 200 samples from the publisher's history queue.

6. To stop showing the shadow trail of samples in Subscriber1, click on **Hide History**.



5.4 Time-based Filtering Example

Sometimes subscribers are located on slower or more remote systems that cannot handle the amount of data that the publisher is capable of sending. For example, consider a system where a central command center is publishing high-resolution aerial photos of a geographic area once every 30 seconds and a soldier with a handheld computer is trying to subscribe to the data. In this case, the handheld computer does not have the bandwidth to handle the command center's send rate. With time-based filtering, the handheld computer can "throttle" the data so that it only receives data once every 5 minutes.

1. Start two copies of *Shapes Demo,* which we will call Publisher1 and Subscriber1. If you are reusing demo windows from the previous section, delete any existing publishers and subscribers (under Controls, click **Delete All**.)
2. In Publisher1, create a circle publisher (any color).
    a. Under **Publish**, click on **Circle**.
    b. In the Create New Publisher window, click **OK**.
3. In Subscriber1, create a circle subscriber, History = 1, Time Based Filter = 1000 ms.
    a. Under **Subscribe**, click on **Circle**.
    b. In the Create New Subscriber window:
        - Set **History** to 1
        - Set **Time Based Filter** to 1000.

- Click **OK**.

You will see the circle jump once every second, instead of a fluid movement. In this case, the publisher is only sending data to the subscriber once a second, according to the subscriber's time-based filtering.

Appendix A Running from the Command Line

In some cases you may want to run *Shapes Demo* from the command line.

Open a command prompt and navigate to the folder where *Shapes Demo* is installed.

Enter the following command:

```
> bin/rtishapesdemo <command-line options>
```

Table A.1 Command-line Options describes the command-line options. These options take precedence over conflicting settings in the configuration file (if any). (For example, if the configuration file specifies domain ID 1 and you enter **-domainId 2**, then domain ID 2 will be used.)

Table A.1 Command-line Options

Option	Description
-compact	Starts *Shapes Demo* using a compact view
-configure	Opens the configuration dialog at start up, even if **-dataType** is set.
-dataType <Shape|ShapeExtended>	Sets the default value for the type.
-domainId <ID>	For different copies of *Shapes Demo* to communicate with each other, they must use the same domain ID. The default domain ID is 0; if you need to use a different domain ID, you must use the same value for all copies of *Shapes Demo* that need to communicate with each other. The ID is an integer value, 0 or higher.
-help	Lists the command-line options.
-posX <integer> -posY <integer>	Sets the X and Y positions where the *Shapes Demo* window will be displayed on your screen. The valid range for <integer> depends on your screen's resolution. Using (-1, -1) for the X and Y positions results in a default position chosen by either the windowing system or wxWidgets, depending on platform.

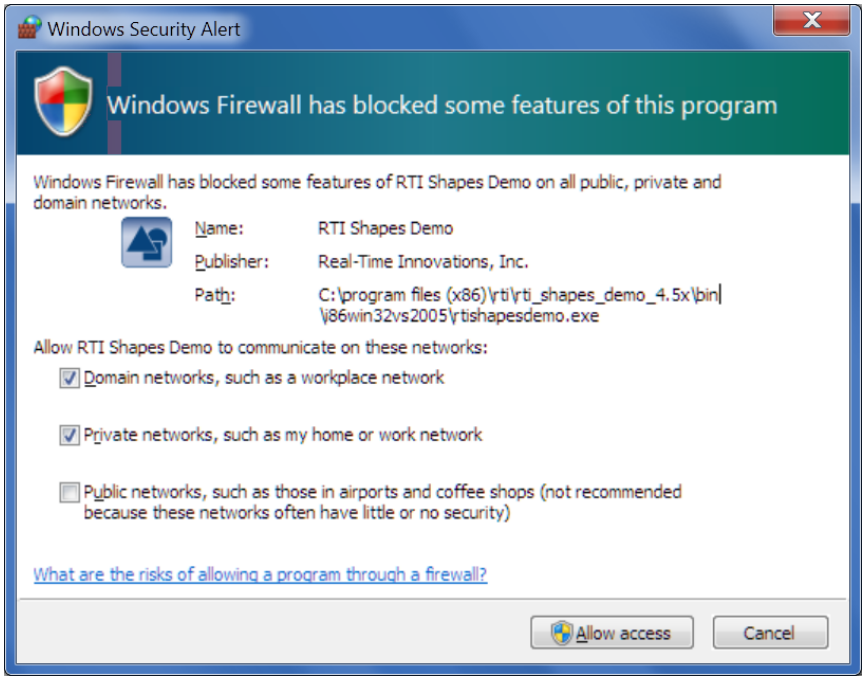
Table A.1 Command-line Options

Option	Description
-pubInterval <integer>	Specifies how often the publisher should send data (in ms). Default: 50 ms
-subInterval <integer>	Specifies how often the subscriber should look for data (in ms). Default: 50 ms
-verbosity <0..5>	Controls the verbosity of messages from *Shapes Demo*. 0 = SILENT No further output will be logged. (Default) 1 = ERROR Only error messages will be logged. 2 = WARNING Both errors and warnings will be logged. 3 = LOCAL Errors, warnings, and verbose information about the lifecycles of local *Connext DDS* objects will be logged. 4 = REMOTE Errors, warnings, and verbose information about the lifecycles of remote *Connext DDS* objects will be logged. 5 = ALL Errors, warnings, verbose information about the lifecycles of local and remote *Connext DDS* objects, and periodic information about *Connext DDS* threads will be logged.
-workspaceFile <file>	Specifies an XML configuration file. Default: See 3.5 Shapes Demo's Workspace on page 23.

Appendix B Troubleshooting

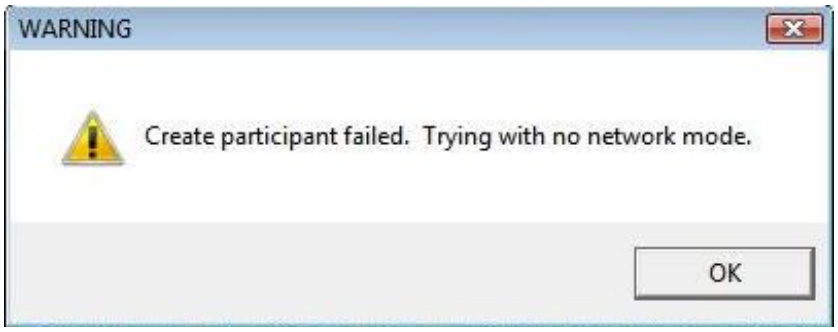
B.1 Windows Security Alert

When you run the demo, you may encounter a "Windows Security Alert" dialog. Simply click **Allow Access**.



B.2 Running without an Active Network Interface

If you run *Shapes Demo* on a system that does not have an active network interface, you may see this warning:



Participant creation failed because, by default, *Shapes Demo* uses UDPv4, which is not available if there is no active network interface.

After you select **OK**, *Shapes Demo* will create a participant using shared memory instead of UDPv4.

Appendix C Known Issues

C.1 Shapes Demo not Scaled Properly in Some Cases

Shapes Demo is not scaled properly when using a scaling factor other than the default (100%), on high-DPI displays on Windows 7, 8, and 8.1.

[RTI Issue ID SHAPES-173]