RTI Data Distribution Service

The Real-Time Publish-Subscribe Middleware

3.x to 4.0 Transition Guide

Version 4.0d

Printed in U.S.A. Second printing.
July 2008.

Trademarks

Copy and Use Restrictions

Technical Support

Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089
Phone: (617) 623-2570 x208
Fax: (617) 623-2574
Email: support@rti.com
Website: http://www.rti.com

Contents

Chapter 1

Overview

What is RTI Data Distribution Service?

RTI Data Distribution Service is a software infrastructure that provides transparent network connectivity that enables the efficient design of highly complex distributed applications. *RTI Data Distribution Service* implements a publish-subscribe communications model and allows distributed processes to share data without concern for the actual physical location and architecture of their peers.

The *RTI Data Distribution Service* 4.0 architecture implements the Data-Centric Publish-Subscribe (DCPS) layer of the OMG Data Distribution Service (DDS) for Real-Time Systems. The goal of DCPS is to provide efficient data movement within a distributed real-time system. With *RTI Data Distribution Service*, system architects and programmers can create fault-tolerant and flexible communication systems that will function over a wide variety of computer hardware, operating systems, languages, and networking transport protocols.

Purpose of this Document

This document has been written to facilitate the understanding and use of the 4.0 (DDS) API for those familiar with the 3.0 API (formerly known as NDDS). After reading this document, a 3.x user should be able to port a 3.x-based application to *RTI Data Distribution Service* 4.0.

The object models of both products are quite similar; both address the real-time autonomous publish-subscribe communication paradigm. While several *RTI Data Distribution Service* 3.x configuration and tuning parameters map directly to QoS policies supported in *RTI Data Distribution Service* 4.0, several new policies are introduced as well. The API

has changed between the products as a result of the introduction of the DDS Specification. Appendix A provides a list of available source code examples that represent use cases using both the 3.x and 4.0 APIs. These buildable projects are available from the same URL used to download this document. For information on which platforms are supported in *RTI Data Distribution Service* 4.0, contact your local RTI Sales Team or see the *RTI Data Distribution Service* 4.0 Release Notes (a separate document that is part of the *RTI Data Distribution Service* 4.0 distribution).

Chapter 2

Object Model Comparative Analysis

The object model of both products is similar in that an application instantiates a Domain or DomainParticipant, and proceeds to create objects that allow the sending and/or receiving of Topic data. The "discovery" information is fully distributed (without centralized name servers), resulting in a highly robust, fault tolerant middleware architecture with no single point of failure.

RTI Data Distribution Service 3.x supports the concept of Subscriptions and Publications within a domain. These are the objects that allow the application the ability to actually 'publish' or 'subscribe' to the data Topic of choice. Publishers and Subscribers allow publications and subscriptions to be managed and are optional. This chapter describes the similarities and differences between the two products.

This chapter includes the following sections:

2.1 Basic Models

For an illustration of the basic object model for *RTI Data Distribution Service* 3.x, see Figure 2.1.

RTI Data Distribution Service 4.0 supports the concepts of *DataWriters* and *DataReaders* within a *DomainParticipant*. These are the objects that allow the application the ability to actually 'publish' or 'subscribe' to the data *Topic* of choice. Publishers and Subscribers are directly associated with DataWriters and DataReaders, respectively. *Publishers* and *Subscribers* are required in *RTI Data Distribution Service* 4.0. For an illustration of the basic object model for *RTI Data Distribution Service* 4.0, see Figure 2.2.

2.2 Domains and DomainParticipants

A *domain* is a distributed concept that links all applications that are able to communicate with each other. A domain represents a communication plane. Domains partition applications into logical units or separate name spaces: only the Publishers and the Subscribers attached to the same domain may interact. The object representing an application's presence in a domain (called a *domain participant* in *RTI Data Distribution Service* 4.0 and simply a Domain in 3.x) is a manager (or factory) for every other DDS object. Domains are global in nature; configuring or changing a Domain's default behavior effects all of the objects created within the Domain. It is important to point out that *RTI Data Distribution Service* 4.0 offers additional flexibility with respect to participating in multiple domains. A single application can participant in multiple domains or can even create multiple DomainParticipants within the same domain.

Figure 2.1 **RTI Data Distribution Service 3.x Object Model**

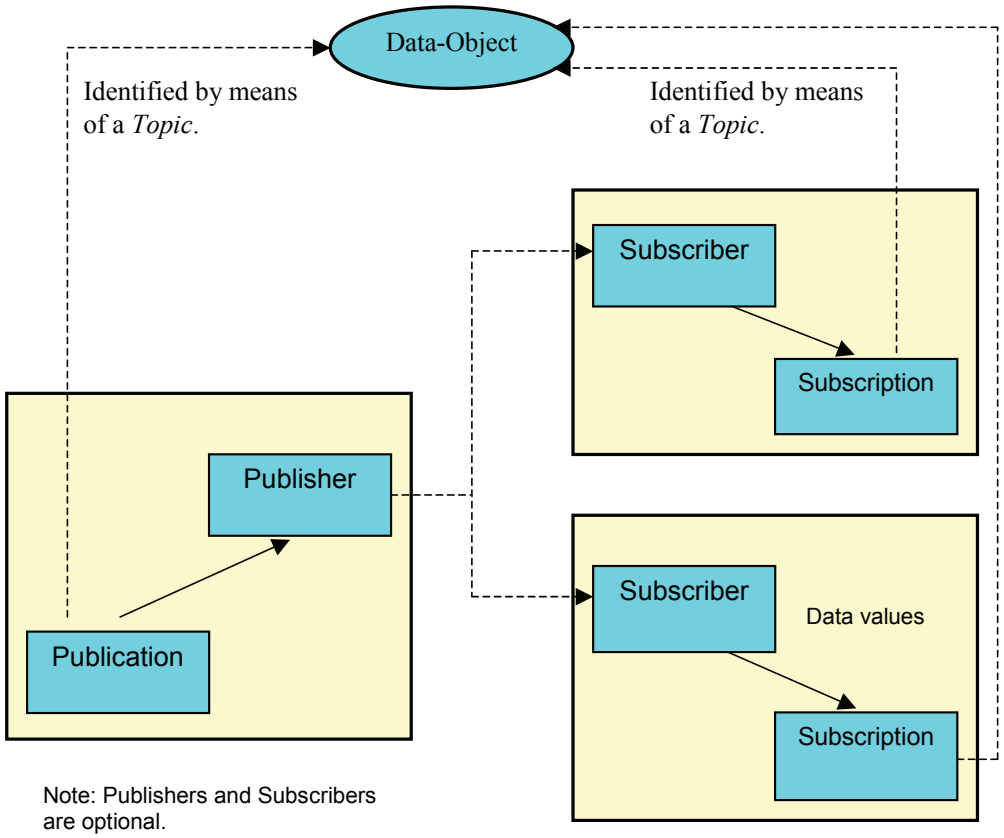
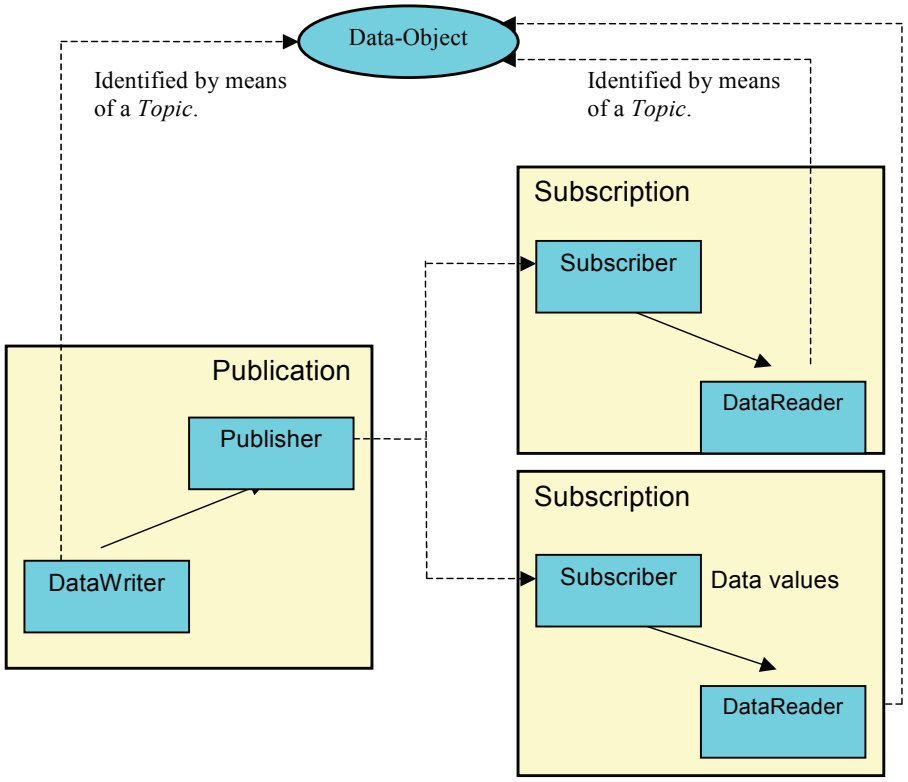


Figure 2.2 **RTI Data Distribution Service 4.0/DDS Object Model**



2.3 Publications and DataWriters

In *RTI Data Distribution Service* 3.x, the Publication is used by an application to write instances of data for publication and has parameters and properties associated with it that dictate behavior. A Publication is created for each Topic to be published and can stand alone or be added to a Publisher.

In *RTI Data Distribution Service* 4.0, a DataWriter is very similar to a Publication in *RTI Data Distribution Service* 3.x from a functional perspective and acts as a liaison between an application and a Publisher, with which it is always associated. The DataWriter

informs the Publisher about the Topic (name, data type, existence) and provides the Publisher with each data sample. The Publisher is responsible for sending the data.

The application writes data using the 'write' operation of the DataWriter, which informs *RTI Data Distribution Service* that there is a new value for the data object. It does not necessarily cause any immediate network communications. The actual generation of messages is controlled by the Publisher and the QoS.

The association of a DataWriter to a Publisher is called a Publication. A Publication indicates that the application intends to publish the data described by the DataWriter by using the Publisher. What is referred to as a "Publication" in *RTI Data Distribution Service* 4.0 is unlike the Publication object in *RTI Data Distribution Service* 3.x. An *RTI Data Distribution Service* 3.x Publication is an actual object with which the user interacts. In *RTI Data Distribution Service* 4.0, a "publication" is a logical concept only; there is no object by that name.

2.4 Subscriptions and DataReaders

In *RTI Data Distribution Service* 3.x, a Subscription is used by an application to subscribe to an *RTI Data Distribution Service* Topic and thus declare the data it wishes to receive. In essence, it provides an interface with *RTI Data Distribution Service* in order to receive incoming published data. A Subscription can stand alone and refers to exactly one Topic that identifies the data to be read. An application may also manage a group of Subscriptions with a Subscriber.

In *RTI Data Distribution Service* 4.0, a DataReader acts as a liaison between an application and a Subscriber and is analogous to a Subscription in *RTI Data Distribution Service* 3.x. The Subscriber is responsible for receiving the data and making it accessible to the DataReader. The DataReader allows the application to declare the Topic it wishes to receive and to access the data received by the attached Subscriber. The application must then use the DataReader's 'read' or 'take' operations to gain access to the received data. A DataReader and Subscriber are inseparable: each DataReader must belong to exactly one Subscriber. The association of a DataReader to a Subscriber in *RTI Data Distribution Service* 4.0 is called a Subscription. A Subscription indicates that the application wants to receive the data described by the DataReader by using the Subscriber. It is important to note that what is referred to as a "Subscription" in *RTI Data Distribution Service* 4.0 is unlike the Subscription object in *RTI Data Distribution Service* 3.x.

To access incoming data, you use a DataReader's Listener. Listeners are a type of callback mechanism for asynchronous notification of data arrival (and other important

events). Any type of DCPS entity can have an associated Listener. When specific events occur (such as the arrival of data, QoS changes, or changes in status), *RTI Data Distribution Service* invokes the appropriate object's Listener method. Each object has just one Listener, which is used to process all incoming events. (Of course, that Listener can be designed to have its own set of callback routines, to further refine event processing.) Similar listener functionality was supported in *RTI Data Distribution Service* 3.x.

The data is accessed by invoking the appropriate operation, such as 'read' or 'take' on the related DataReader. The 'read' method leaves the data in place, whereas 'take' removes the issue from *RTI Data Distribution Service*'s queue.

A future version of the product will implement Conditions and Wait-sets, which will provide a way for an application to block until specific events occur. This mechanism will allow the application to process events synchronously, within its own execution thread.

2.5 Publishers

In *RTI Data Distribution Service* 3.x, an application may create several Publications. Managing each Publication separately can be cumbersome and inefficient. Publishers are optional, but allow applications to manage several Publications, and provide additional modes of configuration in order to support the real-time requirements of multi-threaded applications (i.e. Signaled and Asynchronous publishing). Once a Publisher is created and Publications are added, they are managed as a group and the published messages are coalesced when disseminated for network bandwidth efficiency.

In *RTI Data Distribution Service* 4.0, a Publisher is an object responsible for sending data. You can use the same Publisher to handle the publishing of multiple topics of different data types. A Publisher is a mandatory object responsible for data distribution and allows applications to manage DataWriters as a group. DataWriters are automatically associated with the Publisher when the DataWriter is created. The Publisher acts on behalf of one or several DataWriter objects. When it is informed of a change to the data associated with one of its DataWriter objects, it is responsible for determining when to send, and actually sending the data. This behavior is driven by the attached QoS policies. Note that DataWriters, unlike Publications in *RTI Data Distribution Service* 3.x, cannot stand alone and must be associated with a Publisher. In summary, the concept of Publisher in 3.x and 4.0 is very similar. Both play the same role.

2.6 Subscribers

In *RTI Data Distribution Service* 3.x, an application may create several Subscriptions. Managing each Subscription individually can be cumbersome. A Subscriber allows applications to manage several Subscriptions, and can also support pattern Subscriptions, which allow an application to subscribe to a set of Topics. *RTI Data Distribution Service* uses regular expression patterns and pattern matching rules by default. The pattern matching behavior is consistent with the filename expansion rules used by most UNIX shells.

In *RTI Data Distribution Service* 4.0, a Subscriber is an object responsible for receiving published data and making it available (according to the Subscriber's QoS policies) to the receiving application and may receive and dispatch data of different specified types. To access the received data, the application must use a typed DataReader attached to the Subscriber. A Subscriber is associated with zero or more DataReaders. In summary, the concept of a Subscriber in 3.x and 4.0 is very similar as they both play the same role.

2.7 Topics

In *RTI Data Distribution Service* 3.x, Topics identify the publications in the distributed system and represent the information that other nodes subscribe to. Topics are important in that they are the primary means of connecting the information flow within your application. Topics are user-provided strings that identify both Publications and Subscriptions.

In *RTI Data Distribution Service* 4.0, the Topic is an actual entity and not simply a string. Topics allow Publications to be known in such a way that Subscriptions can refer to them unambiguously. A Topic associates a name (unique in the Domain), a data-type, and QoS related to the data itself. The Topic entity provides support for keys, QoS policies, and Listeners. In *RTI Data Distribution Service* 3.x, a Topic/type instance was singular. In *RTI Data Distribution Service* 4.0, a Topic/type can have multiple instances, distinguishable by a user supplied key. In summary, the concept of a Topic in *RTI Data Distribution Service* 3.x and *RTI Data Distribution Service* 4.0 is very similar and is identified by a string (name) and its associated type.

The main difference is that in *RTI Data Distribution Service* 4.0 there is an explicit Topic object that represents the association of the name and the type and can also have QoS and listeners associated with it.

2.8 DataTypes

The concept of a DataType in *RTI Data Distribution Service* 3.x and 4.0 is very similar. In both cases it represents the format of the data-structure that the application wants to publish/subscribe. The format is represented by a name and described to the middleware by means of a set of functions that encode how to marshal/unmarshal the type. These functions are auto-generated by the bundled utility "nddsgen" from a text document that describes the type in a platform-neutral language.

2.9 Properties, Parameters, and QoS

In *RTI Data Distribution Service* 3.x, we use the terms Properties and Parameters, in *RTI Data Distribution Service* 4.0, we refer to Quality of Service (QoS) and QoS policies. The overall QoS of the system is comprised of individual QoS policies for each object. QoS policies allow developers to configure *RTI Data Distribution Service* to exhibit extremely flexible behavior. QoS policies control many aspects of how and when data is distributed between applications. See Chapter 3: Configuration Parameters and Chapter 4: Object Properties for more details.

2.10 Listeners

Listeners provide a mechanism for *RTI Data Distribution Service* to asynchronously alert the application of the occurrence of relevant events, such as arrival of data corresponding to a Subscription, or a remote application coming online. Listeners are interfaces that the application implements. Each object Listener class provides several virtual methods that correspond to relevant events that the application may wish to respond to. It is the application's responsibility to implement the Listener methods' functionality.

In *RTI Data Distribution Service* 3.x, Listeners can be associated with Publications, Subscriptions, and Domains. Each Listener class provides specific virtual methods appropriate for the object of interest.

RTI Data Distribution Service 4.0 also provides the listener mechanism as a means to notify the application of relevant events. Listeners can be associated with Topics, DataWriters, DataReaders, Publishers, Subscribers, and DomainParticipants. As men-

tioned above, Topics now support their own Listener as well. Each listener class provides specific methods appropriate for the object of interest. (A future version of *RTI Data Distribution Service* will include an additional mechanism for receiving data which will allow the application to block/wait for the events of interest.)

In *RTI Data Distribution Service* 4.0, Listeners are organized in a hierarchical manner where the SubscriberListener generalizes the DataReaderListener, the PublisherListener generalizes the DataWriterListener, and the DomainParticipantListener generalizes all other listeners. The purpose of this hierarchical organization is to allow a more general listener to provide a "default" action in case the more specific Listener does not handle the event. In this manner, the DomainParticipantListener becomes the Listener of last resort that is notified of all status changes not captured by more specific listeners attached to the specific Domain Entity objects. As an example, when a relevant status change occurs associated with a DataReader, *RTI Data Distribution Service* will first attempt to notify the Listener attached to the concerned DataReader if one is installed. Otherwise, *RTI Data Distribution Service* will notify the Listener attached to the Subscriber, or lastly, the DomainParticipant.

2.11 The 'Manager'

In *RTI Data Distribution Service* 3.x, the manager process (called the NDDS Manager) is responsible for discovering remote applications. Each time an application is created or destroyed, the manager process propagates the event information to all other remote managers within the Domain. The manager is a separate process (on operating systems with a process-model such as UNIX® and Windows® NT, Windows 2000, and Windows XP) or a task (VxWorks®). Under normal circumstances, *RTI Data Distribution Service* applications automatically start the manager. Each Domain will have a separate manager process.

In *RTI Data Distribution Service* 4.0, there is no separate manager process. The responsibilities previously performed by the manager process are automatically managed by the *RTI Data Distribution Service* libraries linked to each application.

2.12 Client-Server

RTI Data Distribution Service 4.0 does not provide Client-Server functionality; it is not specified within the DDS Specification. Client-Server functionality will be introduced in a future release of the product.

2.13 Applications and Threads

2.13.1 Threads in *RTI Data Distribution Service* 3.x

In *RTI Data Distribution Service* 3.x, an application is comprised of user threads as well as several *RTI Data Distribution Service* specific threads. There can be several applications active on a single computing node, as well as distributed throughout the network. Each *RTI Data Distribution Service* application will consist of the following threads:

Alarm Thread (AT) — The Alarm Thread (AT) is responsible for waking up other threads at periodic rates. For example, a subscription's deadline is a periodic event that can occur every 2 seconds. There is only one AT thread per application per Domain. As an example, if one application instantiated three Domains, three AT's would be created. Conversely, if two applications participated within one Domain, two AT's would be created. This thread never enters the user's context and is created during initialization.

Receive Thread (RT) — The Receive Thread (RT) receives all user data sent by publications, subscriptions, clients and servers. In the case of reliable subscriptions, the RT will also be responsible for sending acknowledgements. In the case of immediate subscriptions, this is the thread that calls the user's issue listener. There may be more than one RT, depending on whether or not multicast or shared memory is enabled. This thread is created during initialization.

Send Thread (ST) — The Send Thread (ST) is responsible for sending user data. An application can have several STs depending on the number of Publishers and their modes. Each signaled Publisher creates a separate ST. An additional single ST is created for all asynchronous publishers. This thread is created with the first asynchronous publisher. Note that Publications, Clients and Servers and synchronous Publishers do not need an ST. In these cases the data is sent in the user thread.

Database Thread (DT) — The Database Thread (DT) propagates and manages *RTI Data Distribution Service*'s internal database. Each node maintains a database that reflects the entire distributed system. This thread is responsible for ensuring that the internal database is consistent with its peers. This thread is created during initialization.

User Thread (UT) — These represent the application threads used by the user-applications. These threads may be used to send data in the case where the application sends information by means of a Publication, or else if the application used a Publisher that has been configured as a "SYNCHRONOUS" Publisher.

In addition to these *RTI Data Distribution Service*-specific threads, each domain requires that an *RTI Data Distribution Service* manager process be spawned. For operating systems that do not support the process model, such as the VxWorks 5.x operating environment, each thread is spawned as a task—including the *RTI Data Distribution Service* manager.

2.13.2 Threads in RTI Data Distribution Service 4.0

In *RTI Data Distribution Service* 4.0, an application on a specific computer may instantiate one or more DomainParticipant objects within the same or in different domains. DomainParticipants are associated with a specific domainId, have a participantIndex, and cannot contain other DomainParticipants. Several DomainParticipants can co-exist on a single node (computer) within a single Domain (domainId) as long as their participantIndex is unique. DomainParticipants belonging to the same domainId on different computers can use the same participantIndex. A DomainParticipant spawns several *RTI Data Distribution Service*-specific threads. The following threads will exist per DomainParticipant:

Event Thread (ET) — one ET is created per Application. The ET performs several duties, all related to periodic activity within the application. The ET duties include but are not limited to the following:

- Check for DataReader/Writer Deadline expiration, and call Listener when appropriate.
- Sending data issues in response to a NACK from a DataReader. (Note: sending data issues in response to a NACK may be done by a Receive Thread if the extended QoS specifying the time to delay a response to a NACK is zero)
- Check for Liveliness lapses.

The ET handles duties previously performed by the Alarm Thread, NDDS Manager, and the Database Thread in *RTI Data Distribution Service* 3.x.

Database Thread (DT) — is used solely for the purposes of cleaning up the database. This thread no longer disseminates declaration data as it did in *RTI Data Distribution Service* 3.x.

Receive Threads (RT) — there are several receive threads created per application. The number of receive threads needed depends on the number of "Receive Resources" required. These resources are dependent on the transports installed and, depending on the kind of transport, the entry-ports through which reception is expected. For example, the “intra” transport requires only a single receive resource regardless of how many entryports there are; therefore, a single RT is needed to handle all intra communications. For the shared memory transport, a receive resource is needed for each unique "port", so there may be multiple receive threads for shared memory. For IP and IP multicast, a receive resource is needed for each unique port and transport level QoS, and thus we see multiple receive threads for IP and IP multicast (e.g. user-data and meta-traffic) as these are on different ports. We have the following for the default IP transport:

- Intra-transport read thread - handles same CPU pub/sub communications.
- Incoming IP meta-traffic (unicast) - handles unicast receive activity previously handled by the Manager and Database Thread in *RTI Data Distribution Service* 3.x.
- Incoming IP user-data (unicast) - handles receive activity previously handled by the Receive Thread in *RTI Data Distribution Service* 3.x.

All outgoing (non-application-specific data) traffic is handled by the Event Thread, e.g. the responses to NACKs. This also means some outgoing traffic may be handled by a Receive Thread, depending on the extended QoS settings (as mentioned above). Outgoing application-specific user-data will be sent by the user's thread as a result of a write() call; however, it is important to note that outgoing meta-traffic is also sent by the user's thread as a result of creating a new local entity.

There is one additional point that should be considered. We only mentioned the outgoing traffic as it pertains to actual application data (RTPS ISSUEs and VARs). There is also additional outgoing support traffic (RTPS ACKs, HBs, and GAPs) that applies for both metatraffic and user-data, although sometimes in different cases, and are being sent by the Event Thread and Receive Threads.

User Threads (UT) — These represent the threads that are used by the application. *RTI Data Distribution Service* 4.0 will use these threads when the application calls "write" on a DataWriter.

Chapter 3

Configuration Parameters

Both *RTI Data Distribution Service* 3.x and 4.0 allow you to tailor the middleware's behavior to your application's specific requirements. In *RTI Data Distribution Service* 3.x, there are two places where you can specify how *RTI Data Distribution Service* behaves. First, when you create various objects like subscriptions and publications, you pass important parameters like strength, persistence, and deadline to the API that creates the object. Second, you may alter other aspects of these objects directly in their property structures. In *RTI Data Distribution Service* 4.0, all object behavior is controlled through Quality of Service (QoS) policies.

This chapter discusses the *RTI Data Distribution Service* 4.0 equivalents of the parameters explicitly passed when creating the objects. The fields set in 3.x property structures are discussed in Chapter 4.

This chapter includes the following sections:

3.1 Object Parameters in RTI Data Distribution Service 3.x

When you create a Publication in *RTI Data Distribution Service* 3.x, you set its Strength and Persistence; when you create a Subscription, you set its Minimum Separation and

Deadline, and specify whether unicast or multicast will be used. These parameters define the behavior, or QoS, for the object. To use reliable communication, you use a separate 'reliable' API to create the Subscription, still setting the same parameters as mentioned above. The Subscription dictates whether the communication is best-effort or reliable, unicast or multicast.

3.2 Object Parameters in RTI Data Distribution Service 4.0

In *RTI Data Distribution Service* 4.0, you control the behavior of an object by setting QoS policies. QoS policies are set on all entities (Topics, DataWriters, DataReaders, Publishers, Subscribers, and DomainParticipants).

In certain cases, for communications to occur properly, the QoS policy of the Publisher must be compatible with the corresponding policy of the Subscriber. For example, if a Subscriber requests to receive data reliably while the corresponding Publisher only offers best-effort, communication will not be established. To address this issue and maintain the desirable decoupling of Publication and Subscription, the specification for QoS policy follows the Subscriber-requested, Publisher-offered pattern. In this pattern, the Subscriber can specify an ordered list of "requested" values for a particular QoS policy in decreasing order of preference. The Publisher specifies a set of "offered" values for that QoS policy. *RTI Data Distribution Service* selects the most-preferred value requested by the Subscriber that is offered by the Publisher, or it may reject the establishment of communications between the two objects if the QoS requested and the QoS offered cannot be reconciled. This is a new feature inherent within DDS.

On the publishing side, the QoS of each Topic, DataWriter, and Publisher all play a part in controlling how and when data samples are sent to *RTI Data Distribution Service*. Similarly, on the subscribing side, behavior is controlled by the QoS of the Topic, DataReader, and Subscriber. These QoS policies control a variety of behavior, such as how often a DataReader expects to see samples, how to arbitrate when multiple DataWriters send updates for the same Topic and whether a Publisher should save samples in case new Subscriptions later join the network.

The QoS policies in *RTI Data Distribution Service* 4.0 provide a super-set of the control features offered by the object properties in *RTI Data Distribution Service* 3.x. As seen in Table 3.1, some QoS policies map directly from *RTI Data Distribution Service* 3.x parameters to *RTI Data Distribution Service* 4.0 QoS policies. You will note that *RTI Data Distribution Service* 4.0 has several more QoS policies that do not map directly to anything in *RTI Data Distribution Service* 3.x. For more information on these new features, see the *RTI Data Distribution Service* 4.0 User's Manual or online documentation.

Table 3.1 **Controlling Service Behavior**

3.x Parameter	4.0 QoS Policy
deadline	DEADLINE *RTI Data Distribution Service* 4.0 supports the notion of the Publisher having a 'deadline' QoS policy along with the Subscriber. In *RTI Data Distribution Service* 3.x, the 'deadline' QoS was a policy associated only with the Subscription.
minimum separation	TIME_BASED_FILTER *RTI Data Distribution Service* 4.0 also supports the TIME_BASED_FILTER QoS policy for reliable communications. In *RTI Data Distribution Service* 3.x, the minimum separation QoS policy only applied to best-effort communication.
persistence	LIVELINESS (LEASE_DURATION)
reliability (via subscription api call)	RELIABILITY
strength	OWNERSHIP_STRENGTH OWNERSHIP can be either EXCLUSIVE or SHARED. The application can achieve *RTI Data Distribution Service* 3.x publication strength behavior by using OWNERSHIP_STRENGTH and OWNERSHIP_EXCLUSIVE policies on DataWriters. If the application selects OWNERSHIP_SHARED, then multiple DataWriters may alter the Topic issue, and the DataReader will receive all instances of the Topic regardless of ownership strength.
sendqueuesize, timetokeepperiod	DURABILITY, HISTORY, and LIFESPAN *RTI Data Distribution Service* 4.0 QoS are more general and accommodate additional use-cases.

Table 3.1 **Controlling Service Behavior**

3.x Parameter	4.0 QoS Policy
No Direct Mapping	DATA_READER_PROTOCOL
	DATA_READER_RESOURCE_LIMITS
	DATA_WRITER_PROTOCOL
	DATA_WRITER_RESOURCE_LIMITS
	DATABASE
	DESTINATIONORDER
	DISCOVERY_CONFIG
	DISCOVERY
	DOMAIN_PARTICIPANT_RESOURCE_LIMITS
	ENTITYFACTORY
	EVENT
	EXCLUSIVE_AREA
	GROUPDATA
	HISTORY
	LATENCYBUDGET
	LIFESPAN
	PARTITION
	PRESENTATION
	READERDATALIFECYCLE
	RECEIVER_POOL
	RESOURCELIMITS
	SYSTEM_RESOURCE_LIMITS
	TOPICDATA
	TRANSPORT
	TRANSPORTPRIORITY
	USERDATA
	WIRE_PROTOCOL
	WRITERDATALIFECYCLE

Chapter 4

Object Properties

In addition to the parameters explicitly set when creating a Publisher or Subscriber (like the strength, persistence, and deadline parameters discussed in Chapter 3), *RTI Data Distribution Service* 3.x also allows you to modify properties that control the infrastructure's internal behavior. As a result, more performance can be 'tuned' into the system depending on the topology of the application's specific requirements. These object properties are set when the object is created.

This chapter lists the properties you can set for each object in *RTI Data Distribution Service* 3.x, and how they map to associated *RTI Data Distribution Service* 4.0 QoS policies.

This chapter includes the following sections:

4.1 Domain and Infrastructure Properties

4.1.1 NDDSDomainProperties

Table 4.1 provides information on how to map the fields in an NDDSDomainProperties structure.

4.1.2 NDDSTasksProperties

Table 4.2 provides information on how to map the fields in an NDDSTasksProperties structure.

Table 4.1 **Mapping NDDSDomainProperties**

3.x Field Name	Description	4.0 Equivalent
tasks	Stack size and priorities of the tasks automatically started by *NDDS*	See Section 4.1.2
multicast	Properties controlling the use of multicasting	See Section 4.1.3
object	Whether or not clients/servers can create additional clients/servers	Not supported. *RTI Data Distribution Service* 4.0 does not support a Client/Server API. This feature set will be introduced in a future version of the product.
client	Default properties for all clients the application creates	
server	Default properties for all servers the application creates	
publication	Default properties for all publications the application creates	See Section 4.2.1
subscription	Default properties for all subscriptions the application creates	See Section 4.2.2
dgram	Socket properties	See Section 4.1.4
maxSizeSerialize	Maximum size of any *NDDS* issue the application may send or receive. This parameter is typically increased along with sendBufferSize and recvBufferSize when sending large packets greater than 8k bytes.	Not required; specific transports can be configured to handle messages of different sizes.
decl	Controls declarations	See Section 4.1.5
nackRetryMin	Minimum time to wait before resending a NACK for a missing issue in reliable communication mode	DDS_DataWriterQos.protocolX.rtps.reliable_writer.min_nack_response_delay
nicIPAddressCount	Number of entries in nicProperties	Transport configuration API
nicProperties	IP addresses for all NICS on this node	See Section 4.1.6

Table 4.1 **Mapping NDDSDomainProperties**

3.x Field Name	Description	4.0 Equivalent
nddsTopicLength	Maximum length of an *NDDS topic*	Not supported
nddsTypeLength	Maximum length of an *NDDS type*.	
nddsServiceName-Length	Maximum length of an *NDDS Service Name*	
sharedMemory-Communication-Enabled	Whether or not to use shared memory or the network stack for inter-process communication	Transport configuration API.
intraProcessCom-municationEnabled	Whether or not to use intra-process communication	
hostList	Controls where *RTI Data Distribution Service* looks to determine the list of hosts	DDS_DomainParticipantQos.discoveryX.initial_peer_locators[]
domainBase	Control *RTI Data Distribution Service*'s database properties	See Section 4.1.7
internalMulti-castAddress	Multicast address reserved for internal use	Not required
wire	Properties to change RTPS protocol 1.0 functionality	See Section 4.1.8
version	Provides a way to retrieve the product version number	Not supported
lowerCpuUsage-Favored	Controls the function path in NDDSdBase::IssueListAdd. If RTI_FALSE, "ForAllMatches" will be used to achieve low latency. Otherwise (the default), "Find" will be used.	Not required

4.1.3 NDDSMulticastProperties

Multicast support is not available in *RTI Data Distribution Service* 4.0.

Table 4.2 **Mapping NDDSTaskProperties**

3.x Field Name	Description	4.0 Equivalent
realTimeEnabled	Enables support for real-time threading	Not required
atStackSize	Alarm Thread's stack size	DDS_DomainParticipantQoS.eventX.thread.stack_size
atPriority	Alarm Thread's priority	DDS_DomainParticipantQoS.eventX.thread.priority
rtStackSize	Receive Thread's stack size	DDS_DomainParticipantQoS.receiver_poolX.thread.stack_size
rtPriority	Receive Thread's priority	DDS_DomainParticipantQoS.receiver_poolX.thread.priority
stStackSize	Send Thread's stack size	Not supported; Sending Threads are no longer created
stPriority	Send Thread's priority	
dtStackSize	Database Thread's stack size	DDS_DomainParticipantQoS.databaseX.thread.stack_size
dtPriority	Database Thread's priority	DDS_DomainParticipantQoS.databaseX.thread.priority
asyncStackSize	Asynchronous Publisher's stack size	Not available; Publishers do not support asynchronous mode.

4.1.4 NDDSDGramProperties

The datagram properties allow the size of the OS socket send and receive buffers to be altered. The application can reduce this number to conserve memory or increase to handle large messages coming in at high repetition rate. Note: increase this value if the application is dropping declarations during a declaration storm. The dgram field also specifies the size of the message queue used for communication over shared memory message queues (used between applications running on the same node). See Table 4.3.

Table 4.3 **Mapping NDDSDGramProperties**

3.x Field Name	Description	4.0 Equivalent
recvBufferSize	Size of the receive buffer associated with the operating systems stack queue	DDS_DomainParticipantQoS.receiver_poolX.buffer_size
sendBufferSize	Size of the send buffer associated with the operating systems stack queue	Transport Config API

4.1.5 NDDSDeclProperties

Table 4.4 provides information on how to map the fields in an NDDSDeclProperties structure.

Table 4.4 **Mapping NDDSDeclProperties**

3.x Field Name	Description	4.0 Equivalent
bytesPerPacket	Maximum bytes of declaration data to send in a single message	Not supported
bytesPerFastPeriod	Number of bytes to send out per application per fastPeriod	Not required
enabled	Internal parameter for *RTI Data Distribution Service*	
metatrafficPort	Port that the Database Thread binds to in order to send metatraffic data	
push	Whether or not declaration data are immediately distributed when they are created	DDS_DataWriterQos.protocolX.push_on_write
ackSuppression-Delta	Time within which an ACK received is identical to a previous ACK	Not supported

4.1.6 NDDSNICProperties

Table 4.5 provides information on how to map the fields in an NDDSNICProperties structure.

Table 4.5 **Mapping NDDSNICProperties**

3.x Field Name	Description	4.0 Equivalent
ifFlags	Network interface card flag settings	Configured via the TransportConfig API
ipAddress	Network interface card IP address	Configured via the TransportConfig API DDS_DomainParticipantQoS.discoveryX.initial_peer_locators.address can be used to specify the available network interfaces on the computing node

4.1.7 NDDSDomainBaseProperties

Table 4.6 provides information on how to map the fields in an NDDSDomainBaseProperties structure.

Table 4.6 **Mapping NDDSDomainBaseProperties**

3.x Field Name	Description	4.0 Equivalent
slowPeriod	Period at which the Database Thread periodically parses the internal database looking for stale entries	DDS_DomainParticipantQos.databaseX.cleanup_period
fastPeriod	Fastest rate at which the Database Thread will verify the internal database state	Not required
refreshPeriod	How often to refresh the *RTI Data Distribution Service* manager's presence	
expirationTime	How long the application should be considered valid	DDS_DomainParticipantQos.discovery_configX.participant_liveliness_lease_duration
appAnnounceRetries	Maximum retries allowed to contact the application manager	Not required
appAnnouncePeriod	Period at which to contact the application manager	
publisherAsyncPeriod	Period at which the *RTI Data Distribution Service* asynchronous Publisher thread checks for new issues to publish	
rtt	Parameters related to the database's round trip calculations	Not supported
spawnedManager	See Section 4.1.9	See Section 4.1.9

4.1.8 RTPSWireProtocolProperties

Table 4.7 provides information on how to map the fields in an RTPSWireProtocolProperties structure.

Table 4.7 **Mapping RTPSWireProtocolProperties**

3.x Field Name	Description	4.0 Equivalent
appId	Manually assigned application ID	DDS_DomainParticipantQoS.wire_protocolX.rtps_app_id
metaMulticastIP	Manually assigns the metatraffic multicast IP address	DDS_DomainParticipantQoS.discoveryX.multicast_groups[]
ackMulticastApplicationEnabled	Whether or not RTPS_ACK message should contain a multicast IP address	Not required
ackMulticastManagerEnabled	Whether or not RTPS_ACK message should contain a multicast IP address	
multicastThreshold	How ACKs, HBs, and VARs are sent	Not supported
appGroupRemoteProperties	RTPS protocol functionality	
mgrGroupRemoteProperties	RTPS protocol functionality	

4.1.9 NDDSAppManagerProperties

Table 4.8 provides information on how to map the fields in an NDDSAppManagerProperties structure.

Table 4.8 **Mapping NDDSAppManagerProperties**

3.x Field Name	Description	4.0 Equivalent
stackSize	NDDS manager's stack size	Not required; there is no *RTI Data Distribution Service* manager process
priority	Priority of the NDDS manager's thread	
terminate	Whether or not the NDDS manager spawned by the local application will terminate	
sendBufferSize	Size of received messages that the NDDS manager can receive	
recvBufferSize	Size of messages to be disseminated by the NDDS manager	
multicast	Whether or not the manager should send and received using multicast addressing, and if so, with what properties	
purgePeriod	Rate at which the NDDS manager will clean up stale objects within the internal database at this specified rate	
refreshPeriod	Rate at which the NDDS manager's presence is updated	
expirationTime	NDDS manager's expiration time	
pushDelayMax	Maximum delay the NDDS manager will wait prior to pushing manager changes to fellow managers	
pushAttempMax	Number of times to push changes in the NDDS manager to fellow managers	
wire	RTPS wire protocol properties. See Section 4.1.8	See Section 4.1.8

Table 4.8 **Mapping NDDSAppManagerProperties**

3.x Field Name	Description	4.0 Equivalent
hostList	Fellow managers that should be communicated with for inter-manager protocol communication	Not required; there is no *RTI Data Distribution Service* manager process
initialAnnounceEnabled	Enables or disables firing off the announcement of the local application's existence to all the other managers prior to executing the local application	
numNics	Number of valid IP addresses in the nicIPAddress field	DDS_DomainParticipantQoS.discoveryX.multicast_groups_count DDS_DomainParticipantQoS.discoveryX.initial_peer_locators_count
nic[]	Valid IP addresses the NDDS manager should use	Not required; there is no manager process. As mentioned in Section 4.1.6, DDS_DomainParticipantQoS.discoveryX.initial_peer_locators can be used to specify the available network interfaces on the computing node.

4.2 Publication and Subscription Properties

4.2.1 NDDSPublicationProperties

Table 4.9 provides information on how to map the fields in an NDDSPublicationProperties structure.

Table 4.9 **Mapping NDDSPublicationProperties**

3.x Field Name	Description	4.0 Equivalent
timeToKeepPeriod	How long a Publication should keep issues it has already published	Similar functionality is available in the DURABILITY TRANSIENT LOCAL and LIFESPAN QoS policies
persistence	Expiration time. Before this time expires, a Subscription ignores issues from redundant Publications of lesser strength. After this time, a Subscription will accept the next lower-strength Publication as the valid data source.	DDS_DataWriterQos.liveliness. lease_duration
strength	Controls arbitration among multiple Publishers of the same *RTI Data Distribution Service* Topic. Publication strength allows the Subscriber the ability to arbitrate among multiple Publications.	DDS_DataWriterQos. ownership_strength.value
serializeOption	Allows custom information to be passed to the serialization routine; it is not interpreted by *RTI Data Distribution Service*.	Not supported
sendQueueSize	Number of previous issues to save	The ability to set the publishing queue size is facilitated by setting DDS_DataWriterQos. resource_limits.max_samples. Note: the HISTORY and RESOURCE_LIMITS QoS policies must also be considered when setting up instance sizing.

Table 4.9 **Mapping NDDSPublicationProperties**

3.x Field Name	Description	4.0 Equivalent
threadId	Unique thread ID for each publication thread within a Publisher; this guarantees thread-safe operation.	Not required when using multiple DataWriters in a single Publisher
sendSocketHandle	Socket handle to be used by *RTI Data Distribution Service*	Not supported
lowWaterMark	Point at which *RTI Data Distribution Service* should indicate that the reliable publication's send queue's low water mark has been reached	
highWaterMark	Point at which *RTI Data Distribution Service* should indicate that the reliable publication's send queue's high-water mark has been reached	
heartBeatTimeout	How often to send a heart beat message to the reliable Subscription at the normal rate when the send queue size is below the highWaterMark	DDS_DataWriterQos. protocolX.heartbeat_period
heartBeatFastTimeout	How often to send a heart beat message to the reliable Subscription at the fast rate when the send queue size is above the highWaterMark	Not supported
sendMaxWait	Maximum time that a Publication can be blocked while sending data	
heartBeatRetries	Number of times to send a heart beat without receiving a response from the Subscription	DDS_DataWriterQos.protocolX. max_heartbeat_retries
heartBeatPerSendQueue	Number of heartbeats to insert in between issues each time 'sendQueueSize' issues are published	DDS_DataWriterQos.protocolX. heartbeats_per_queue
nackReplySuppressionTime	Amount of time for which *RTI Data Distribution Service* will suppress replying to a heartbeat already received	DDS_DataWriterQos.protocolX. min_nack_response_delay and DDS_DataWriterQos.protocolX. max_nack_response_delay

4.2.2 NDDSSubscriptionProperties

Table 4.10 provides information on how to map the fields in an NDDSSubscriptionProperties structure.

Table 4.10 **Mapping NDDSSubscriptionProperties**

3.x Field Name	Description	4.0 Equivalent
mode	Subscription's mode of operation (either IMMEDIATE or POLLED). Determines when *RTI Data Distribution Service* invokes the Subscription Listener. For an IMMEDIATE Subscription, *RTI Data Distribution Service* 3.x invokes the Listener when the issue is received. The Listener is invoked in the context of an *RTI Data Distribution Service* internal thread. For a POLLED Subscription, the Listener is only invoked when the application polls the Subscription.	The immediate mode is supported by using the listener. The polling mode is not supported in *RTI Data Distribution Service* 4.0. (In a future version, polling will be implemented by using the DDS concept of Conditions and Wait-sets).
deadline	Guaranteed time by which the provided callback will be invoked	The deadline parameter is not specified when creating the Subscription (or DataReader). Deadline is a QoS policy and can be established on a per DataWriter/DataReader basis by setting DDS_DataWriterQos.deadline.period and DDS_DataReaderQos.deadline.period
minimumSeparation	Minimum amount of time between invocations of a Subscription's callback routine	Similar to the deadline property, in *RTI Data Distribution Service* 4.0, the minimumSeparation is not specified when creating the Subscription (or DataReader). The time_based_filter QoS policy can be set for a DataReader (DDS_DataReaderQos.time_based_filter.minimum_separation)

Table 4.10 **Mapping NDDSSubscriptionProperties**

3.x Field Name	Description	4.0 Equivalent
receiveQueue-Size	Maximum number of issues the Subscription should store	DDS_DataReaderQos.resource_limits. max_samples, in conjunction with the HISTORY and RESOURCE_LIMITS QoS policies
enabled	Whether or not a Subscription is enabled	Domain's enable() operation

4.3 Client and Server Properties

RTI Data Distribution Service 4.0 does not support a Client/Server API. This feature set will be introduced in a future version of the product.

Chapter 5

Comparing the C APIs

This chapter describes how the *RTI Data Distribution Service* 3.x C API maps to the *RTI Data Distribution Service* 4.0 API. It focuses on how to port an *RTI Data Distribution Service* 3.x C application to *RTI Data Distribution Service* 4.0. This chapter addresses each *RTI Data Distribution Service* 3.x object and its supported routines and attempts to map the functionality to equivalent *RTI Data Distribution Service* 4.0 routines and/or functionality. Where direct or indirect mappings do not exist, we'll recommend alternate approaches for you to consider. Example source code (using both versions) will be used and will assume best-effort QoS and unicast network addressing. For additional examples of reliable communications, see Appendix A, which lists the buildable example source code that is available online.

This chapter includes the following sections:

5.1 Examples

We'll start with examples of how to instantiate a domain, and send and receive data.

5.1.1 Domain Instantiation

RTI Data Distribution Service 3.x:

```
int             nddsDomain = NDDS_DOMAIN_DEFAULT;
int             nddsVerbosity = NDDS_VERBOSITY_DEFAULT;
NDDSDomain      domain;

NddsVerbositySet(nddsVerbosity);
domain = NddsInit(nddsDomain, NULL, NULL);
```

RTI Data Distribution Service 4.0:

```
DDS_DomainParticipantFactory          *factory = NULL;
struct DDS_DomainParticipantQos       participant_qos;
DDS_DomainParticipant          *participant = NULL;
if(!(factory=DDS_DomainParticipantFactory_get_instance())) {
    return RTI_FALSE;
}
DDS_DomainParticipantFactory_get_default_participant_qos(factory,
        &participant_qos);
participant_qos.discoveryX.participant_index = participantIndex;
if (peerHost != NULL) {

if (!RTINetioAddress_getIpv4AddressByName((RTINetioAddress *)
                &participant_qos.discoveryX.initial_peer_locators[0].address,
                peerHost)) {
            return RTI_FALSE;
    }
participant_qos.discoveryX.initial_peer_locators[0].participant_index_limit =
        peerMaxIndex;
participant_qos.discoveryX.initial_peer_locators_count = 1;

}

if(!(participant = DDS_DomainParticipantFactory_create_participant(factory,
        nddsDomain,
        &participant_qos, NULL))) {
    return RTI_FALSE;
};
```

Notice that there are a few more 'infrastructure' calls that must be used in *RTI Data Distribution Service* 4.0 prior to the 'create_participant' routine that instantiates the actual DomainParticipant. The *RTI Data Distribution Service* 3.x and 4.0 routines to instantiate a domain are shown in Table 5.1 for comparison purposes.

Table 5.1 **C Routines for Creating a Domain**

3.x	4.0
NDDSInit	DDS_DomainParticipantFactory_get_instance
	DDS_DomainParticipantFactory_get_default_participant_qos
	DDS_DomainParticipantFactory_create_participant

Only a few routines are required to actually create the Domain itself. Once the Domain exists, the application is provided a rich set of routines that can be used to support application functionality. In the following sections, you'll find example source code of how an application publishes and subscribes data using both *RTI Data Distribution Service* 3.x and 4.0.

5.1.2 Publishing Data

In *RTI Data Distribution Service* 3.x, a Publication can stand alone. It can also be added to a Publisher if so desired. As you'll see in *RTI Data Distribution Service* 4.0 source code, the Publisher is instantiated first, followed by the DataWriter.

RTI Data Distribution Service 3.x:

```
int publisherMain(int nddsDomain, int nddsVerbosity)
{
   Int                 count = 0;
   RTINtpTime          send_period_sec = {0,0};
   RTINtpTime          persistence = {0,0};
   Int                 strength = 1;
   NDDSPublication     publication;
   NDDSPublisher       publisher;
   HelloMsg            *instance = NULL;
   NDDSDomain          domain;

   RtiNtpTimePackFromNanosec(send_period_sec, 4, 0);    /* 4 seconds */
   RtiNtpTimePackFromNanosec(persistence, 15, 0);       /* 15 seconds */

   NddsVerbositySet(nddsVerbosity);
   domain = NddsInit(nddsDomain, NULL, NULL);

   HelloMsgNddsRegister();
```

```
    printf("Allocate HelloMsg type.\n");
    instance = HelloMsgAllocate();

    publisher = NddsPublisherCreate(domain, NDDS_PUBLISHER_SIGNALLED);
    publication = NddsPublicationCreate(domain, "Example HelloMsg",
                        HelloMsgNDDSType, instance, persistence, strength);

    NddsPublisherPublicationAdd(publisher, publication);

    for (count=0;;count++) {
       printf("Sampling publication, count %d\n", count);

       /* Modify data to be published */
       sprintf(instance->msg, "Hello Universe! (%d)", count);

       NddsPublisherSend(publisher);
       NddsUtilitySleep(send_period_sec);
    }
    return RTI_TRUE;
};
```

RTI Data Distribution Service 4.0:

```
int publisherMain(int nddsDomain, int participantIndex, const char *peerHost,
        int peerMaxIndex)
{
    DDS_DomainParticipantFactory             *factory = NULL;
    struct DDS_DomainParticipantQos  participant_qos;
    DDS_DomainParticipant            *participant = NULL;
    DDS_Publisher                    *publisher = NULL;
    DDS_Topic                        *topic = NULL;
    HelloMsgDataWriter               *writer = NULL;
    HelloMsg                         *instance = NULL;
    DDS_ReturnCode_t                 retcode;
    DDS_InstanceHandle_t             instance_handle = DDS_HANDLE_NIL;
    int                              count = 0;
    RTINtpTime                       send_period_sec = {0,0};


    RtiNtpTimePackFromNanosec(send_period_sec, 4, 0); /* 4 seconds */

    if(!(factory=DDS_DomainParticipantFactory_get_instance())) {
               return RTI_FALSE;
    }

    DDS_DomainParticipantFactory_get_default_participant_qos(factory,
                                      &participant_qos);
    participant_qos.discoveryX.participant_index = participantIndex;
    if (peerHost != NULL) {
```

```
    if (!RTINetioAddress_getIpv4AddressByName((RTINetioAddress *)
            &participant_qos.discoveryX.initial_peer_locators[0].address,
            peerHost)) {
                return RTI_FALSE;
    }
participant_qos.discoveryX.initial_peer_locators[0].participant_index_limit
    = peerMaxIndex;
    participant_qos.discoveryX.initial_peer_locators_count = 1;
}

if(!(participant = DDS_DomainParticipantFactory_create_participant(factory,
    nddsDomain, &participant_qos, NULL))) {
    return RTI_FALSE;
};

if(!(publisher = DDS_DomainParticipant_create_publisher(participant,
    DDS_PUBLISHER_QOS_DEFAULT, NULL))) {
    return RTI_FALSE;
};

retcode = HelloMsgTypeSupport_register_type(participant, HelloMsgTYPENAME);
if (retcode != DDS_RETCODE_OK) {
    return RTI_FALSE;
}

if(!(topic = DDS_DomainParticipant_create_topic(participant, "Example Hel-
    loMsg", HelloMsgTYPENAME, DDS_TOPIC_QOS_DEFAULT, NULL))) {
    return RTI_FALSE;
}

if(!(writer = (HelloMsgDataWriter *)DDS_Publisher_create_datawriter(pub-
    lisher, topic, DDS_DATAWRITER_QOS_DEFAULT, NULL))) {
    return RTI_FALSE;
}

if(!(instance = HelloMsgTypeSupport_createX())) {
    return RTI_FALSE;
}

for (count=0;count>=0;count++) {
    RtiDebugPrint(
    "C API: Publishing best-effort/unicast example, count %d\n", count);

   /* Modify data to be published */
    sprintf(instance->msg,
    "C API: Publishing best-effort/unicast example, count %d", count);
```

```
            retcode = HelloMsgDataWriter_write(writer, instance, &instance_handle);
            if (retcode != DDS_RETCODE_OK) {
                return RTI_FALSE;
            }

            RtiThreadSleep(&send_period_sec);
        }
        return RTI_TRUE;
};
```

The *RTI Data Distribution Service* 3.x and 4.0 routines used in the above example are shown in Table 5.2 for comparison purposes.

Table 5.2 **C Routines for Publishing Data**

3.x routines	4.0 routines
NDDSInit	DDS_DomainParticipantFactory_get_instance
	DDS_DomainParticipantFactory_get_default_participant_qos
	DDS_DomainParticipantFactory_create_participant
NddsPublisherCreate	DDS_DomainParticipant_create_publisher
NddsPublicationCreate	
	HelloMsgTypeSupport_register_type
HelloMsgNddsRegister	DDS_DomainParticipant_create_topic
NddsPublisherPublication-Add	DDS_Publisher_create_datawriter
	HelloMsgDataType_createX
NddsPublisherSend	HelloMsgDataWriter_write

Compilable versions of both the *RTI Data Distribution Service* 3.x and 4.0 source code shown above, as well as other examples, are available for download (see Appendix A).

5.1.3 Subscribing To Data

In *RTI Data Distribution Service* 3.x, a Subscription can stand alone. It can also be added to a Subscriber if so desired. As you'll see in the *RTI Data Distribution Service* 4.0 source code, the Subscriber is instantiated first, followed by the DataReader.

RTI Data Distribution Service 3.x:

```
RTIBool HelloMsgCallback(const NDDSRecvInfo *issue, NDDSInstance *instance, void
        *callBackRtnParam)
{
   if (issue->status == NDDS_FRESH_DATA) {
      HelloMsg *item = (HelloMsg *)instance;

      HelloMsgPrint(item, 0);
      return RTI_TRUE;
   }
};

int subscriberMain(int nddsDomain, int nddsVerbosity)
{
   RTINtpTime                  deadline = {0,0};
   RTINtpTime                  min_separation = {0,0};
   NDDSSubscription            subscription;
   NDDSSubscriber              subscriber;
   NDDSSubscriptionProperties  properties;
   HelloMsg                    *instance = NULL;
   NDDSDomain                  domain;
   char               deadlineString[RTI_NTP_TIME_STRING_LEN];

   RtiNtpTimePackFromNanosec(deadline, 10, 0);
   RtiNtpTimePackFromNanosec(min_separation, 0, 0);

   NddsVerbositySet(nddsVerbosity);
   domain = NddsInit(nddsDomain, NULL, NULL);

   HelloMsgNddsRegister();

   printf("Allocate HelloMsg type.\n");
   instance = HelloMsgAllocate();

   subscriber = NddsSubscriberCreate(domain);

   subscription = NddsSubscriptionCreate(domain, NDDS_SUBSCRIPTION_IMMEDIATE,
               "Example HelloMsg", HelloMsgNDDSType, instance, deadline,
               min_separation, HelloMsgCallback, NULL, NDDS_USE_UNICAST);

   NddsSubscriptionPropertiesGet(subscription, &properties);
   properties.receiveQueueSize = 8;
   NddsSubscriptionPropertiesSet(subscription, &properties);

   NddsSubscriberSubscriptionAdd(subscriber, subscription);
```

```
   while (1) {
      printf("Sleeping for %s sec...\n",
      RtiNtpTimeToString(&deadline, deadlineString));
      NddsUtilitySleep(deadline);
   }
   return RTI_TRUE;
};
```

RTI Data Distribution Service 4.0

```
void MyListener_on_requested_deadline_missed(void* listener_data,
       DDS_DataReader* reader,
       const struct DDS_RequestedDeadlineMissedStatus *status) { }

void MyListener_on_requested_incompatible_qos(void* listener_data,
       DDS_DataReader* reader,
       const struct DDS_RequestedIncompatibleQosStatus *status) { }

void MyListener_on_sample_rejected(void* listener_data,
       DDS_DataReader* reader,
       const struct DDS_SampleRejectedStatus *status) { }

void MyListener_on_liveliness_changed(void* listener_data,
       DDS_DataReader* reader,
       const struct DDS_LivelinessChangedStatus *status) { }

void MyListener_on_sample_lost(void* listener_data,
       DDS_DataReader* reader,
       const struct DDS_SampleLostStatus *status) { }

void MyListener_on_subscription_match(void* listener_data,
       DDS_DataReader* reader,
       const struct DDS_SubscriptionMatchStatus *status) { }

void MyListener_on_data_available(void* listener_data, DDS_DataReader* reader)
{
    HelloMsgDataReader *HelloMsgReader = (HelloMsgDataReader *)reader;
    struct HelloMsgSeq data_seq =     DDS_NEW_EMPTY_SEQUENCE;
    struct DDS_SampleInfoSeq info_seq =DDS_NEW_EMPTY_SEQUENCE;
    DDS_ReturnCode_t                  retcode;
    int                               i;

    retcode = HelloMsgDataReader_take(HelloMsgReader, &data_seq, &info_seq,
           DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE,
           DDS_ANY_INSTANCE_STATE);
    if (retcode != DDS_RETCODE_OK) {
        return;
    }
```

```
    for (i = 0; i < HelloMsgSeq_get_length(&data_seq); ++i) {
        HelloMsgTypeSupport_printX(HelloMsgSeq_get_address(&data_seq, i));
    }

    HelloMsgDataReader_return_loan(HelloMsgReader, &data_seq, &info_seq);
}

int subscriberMain(int nddsDomain, int participantIndex, const char *peerHost,
       int peerMaxIndex)
{
    DDS_DomainParticipantFactory      *factory = NULL;
    struct DDS_DomainParticipantQos   participant_qos;
    DDS_DomainParticipant             *participant = NULL;
    DDS_Subscriber                    *subscriber = NULL;
    DDS_Topic                         *topic = NULL;
    struct DDS_DataReaderListener     listener = DDS_DATAREADER_LISTENER_DEFAULT;
    HelloMsgDataReader                *reader = NULL;
    DDS_ReturnCode_t                  retcode;
    RTINtpTime                        receive_period_sec = {0,0};
    char                              deadlineString[RTI_NTP_TIME_STRING_LEN];
    int                               count = 0;

    RtiNtpTimePackFromNanosec(receive_period_sec, 4, 0); /* 4 seconds */

    if(!(factory=DDS_DomainParticipantFactory_get_instance())) {
        return RTI_FALSE;
    }

    DDS_DomainParticipantFactory_get_default_participant_qos(factory,
       &participant_qos);
    participant_qos.discoveryX.participant_index = participantIndex;

    if (peerHost != NULL) {
        if (!RTINetioAddress_getIpv4AddressByName((RTINetioAddress *)
       &participant_qos.discoveryX.initial_peer_locators[0].address, peerHost))
       {
            return RTI_FALSE;
        }
        participant_qos.discov-
       eryX.initial_peer_locators[0].participant_index_limit = peerMaxIndex;

       participant_qos.discoveryX.initial_peer_locators_count = 1;
    }

    if(!(participant = DDS_DomainParticipantFactory_create_participant(factory,
                                       nddsDomain, &participant_qos, NULL))) {
        return RTI_FALSE;
    }
```

```
    if(!(subscriber = DDS_DomainParticipant_create_subscriber(participant,
                                    DDS_SUBSCRIBER_QOS_DEFAULT, NULL))) {
        return RTI_FALSE;
    }

    retcode = HelloMsgTypeSupport_register_type(participant, HelloMsgTYPENAME);
    if (retcode != DDS_RETCODE_OK) {
        return RTI_FALSE;
    }

    if(!(topic = DDS_DomainParticipant_create_topic(participant,
        "Example HelloMsg", HelloMsgTYPENAME, DDS_TOPIC_QOS_DEFAULT, NULL))) {
        return RTI_FALSE;
    }

    listener.on_requested_deadline_missed =
                                    MyListener_on_requested_deadline_missed;

    listener.on_requested_incompatible_qos =
                                    MyListener_on_requested_incompatible_qos;

    listener.on_sample_rejected =           MyListener_on_sample_rejected;
    listener.on_liveliness_changed =        MyListener_on_liveliness_changed;
    listener.on_sample_lost =               MyListener_on_sample_lost;
    listener.on_subscription_match =        MyListener_on_subscription_match;
    listener.on_data_available =            MyListener_on_data_available;

    if(!(reader = (HelloMsgDataReader *)DDS_Subscriber_create_datareader(
                            subscriber,  DDS_Topic_as_TopicDescription(topic),
                            DDS_DATAREADER_QOS_DEFAULT, &listener))) {
        return RTI_FALSE;
    }

    for (count=0;count>=0;count++) {
       RtiDebugPrint("Sleeping for %s sec...\n",
                     RtiNtpTimeToString(&receive_period_sec, deadlineString));
        RtiThreadSleep(&receive_period_sec);
    }
    return RTI_TRUE;
};
```

The *RTI Data Distribution Service* 3.x and 4.0 routines used in the above example are shown in Table 5.3 for comparison purposes.

Compilable versions of both the *RTI Data Distribution Service* 3.x and 4.0 source code shown above, as well as other examples, are available for download (see Appendix A).

Table 5.3 **C Routines for Subscribing to Data**

3.x	4.0
NDDSInit	DDS_DomainParticipantFactory_get_instance
	DDS_DomainParticipantFactory_get_default_participant_qos
	DDS_DomainParticipantFactory_create_participant
HelloMsgNddsRegister	
NddsSubscriberCreate	DDS_DomainParticipant_create_subscriber
NddsSubscriptionCreate	
	HelloMsgTypeSupport_register_type
	DDS_DomainParticipant_create_topic
NddsSubscriptionPropertiesGet	
NddsSubscriptionPropertiesSet	
NddsSubscriberSubscriptionAdd	DDS_Subscriber_create_datareader

5.2 Domain API

Recall that a domain is a distributed concept that links all applications that are able to communicate with each other and represents a communication plane. Only the Publishers and the Subscribers attached to the same domain may interact. A Domain is a manager (or factory) of every *RTI Data Distribution Service* object. Domains are global in nature and represent a communication plane.

Table 5.4 lists the *RTI Data Distribution Service* 3.x domain routines (in alphabetical order). Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality.

There are also several data-type specific routines, see Section 5.2.7.

Table 5.4 **Domain C API**

3.x	Reference to 4.0 Information
NddsClientPropertiesDefaultGet	Section 5.2.6
NddsDBaseRemove	Section 5.2.8
NddsDBaseFind	
NddsDBaseAdd	
NddsDestroy	Section 5.2.1
NddsDomainHandleGet	Section 5.2.2
NddsDomainIndexGet	
NDDSDomainListener	Section 5.8.1
NddsInit	Section 5.2.1
NddsPublicationPropertiesDefaultGet	Section 5.2.6
NddsServerPropertiesDefaultGet	
NddsSubscriptionPropertiesDefaultGet	

5.2.1 Domain Create/Delete Routines

NddsDestroy — Destroys the domain specified. In *RTI Data Distribution Service* 4.0, use DDS_DomainParticipantFactory_delete_participant().

NddsInit — Initializes *RTI Data Distribution Service*, instantiates a domain of a specified index, returns a domain handle given a domain number. In *RTI Data Distribution Service* 4.0, use these routines:

- ❏ DDS_DomainParticipantFactory_get_instance()
- ❏ DDS_DomainParticipantFactory_get_default_participant_qos()
- ❏ DDS_DomainParticipantFactory_create_participant()

5.2.2 Domain Index and Handle Retrieval Routines

NddsDomainIndexGet — Returns the Domain's index. In *RTI Data Distribution Service* 4.0, use DDS_DomainParticipant_get_domain_id().

NddsDomainHandleGet — Returns a domain handle given a domain number and provides a convenient means for retrieving an already created domain. In *RTI Data Distribution Service* 4.0, use DDS_DomainParticipantFactory_get_instance().

5.2.3 Domain Wait Routine

NddsDomainWait — Internally, *RTI Data Distribution Service* manages a database of all of the remote publications and subscriptions. When the system with a complex architecture boots up, the propagation of the database information can take some time. This routine can be used to "wait" for internal states to settle. This routine is not supported in *RTI Data Distribution Service* 4.0.

5.2.4 Wire Protocol Properties Routine

NddsWireProtocolPropertiesGet — Returns the wire protocol properties. See Section 4.1.8.

5.2.5 Manager and Application Host Routines

NddsManagerHostsGet — Returns the list of NDDS Managers. This is not required in *RTI Data Distribution Service* 4.0.

NddsAppHostsSet — Allows a list of application hosts to be defined. In *RTI Data Distribution Service* 4.0, use DDS_DiscoveryQosPolicyX.discoveryX.initial_peer_locators[].

NddsAppHostsGet — Retrieves the list of application hosts allowed to participate in the *RTI Data Distribution Service*-enabled network. In *RTI Data Distribution Service* 4.0, use DDS_DiscoveryQosPolicyX.discoveryX.initial_peer_locators[].

5.2.6 Default Properties Routines

NddsServerPropertiesDefaultGet — Not supported in *RTI Data Distribution Service* 4.0.

NddsClientPropertiesDefaultGet — Not supported in *RTI Data Distribution Service* 4.0.

NddsPublicationPropertiesDefaultGet — Retrieves the publication's property structure so that modifications can be made to the properties and subsequently create a publication in one atomic action. The properties associated with a Publication in *RTI Data Distribution Service* 3.x do not map directly to DataWriter properties in *RTI Data Distribution Service* 4.0, but object QoS can be retrieved for a given DataWriter by using the DDS_DomainParticipant_get_default_datawriter_qosX routine.

NddsSubscriptionPropertiesDefaultGet — Retrieves the Subscription's property structure so that modifications can be made to the properties and subsequently create a Subscription in one atomic action. The properties associated with a Subscription in *RTI Data Distribution Service* 3.x do not map directly to DataReader properties in *RTI Data Distribution Service* 4.0, but object QoS can be retrieved for a given DataReader by calling the DDS_DomainParticipant_get_default_datareader_qosX routine.

5.2.7 Type API

The Type API provides a set of functions that allow data types to be registered with the *RTI Data Distribution Service* middleware infrastructure. Table 5.5 lists the *RTI Data Distribution Service* 3.x data-type routines. Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality.

Table 5.5 **Data-Type Routines**

3.x	Reference to 4.0 Information
SerializeMethodType	Section 5.2.7.1
DeserializeMethodType	
PrintMethodType	Section 5.2.7.2
FreeMethodType	Section 5.2.7.3
MaxSizeMethodType	
ScopeRegisterMethodType	Section 5.2.7.4
NddsTypeRegister	Section 5.2.7.3
NddsTypeScopeRegister	Section 5.2.7.4
NddsTypeDestroy	Section 5.2.7.3

See Appendix A for examples of both the data-type definition and resulting source code generated by the 'nddsgen' utility for both *RTI Data Distribution Service* 3.x and 4.0.

In both *RTI Data Distribution Service* 3.x and 4.0, use of the provided 'nddsgen' utility is optional. You may choose to write the data-type routines yourself if so desired.

5.2.7.1 Serialization Routines

SerializeMethodType — Provides the prototype for the serialize routine required to serialize an object instance into the NDDSCDRStream when sending a message to a peer. All data types must provide this functionality when a data type is registered. In *RTI Data Distribution Service* 3.x, nddsgen automatically creates this routine and provides the source code. In *RTI Data Distribution Service* 4.0, nddsgen will create the equivalent serialization source code.

DeserializeMethodType — Provides the prototype for the deserialize routine required to deserialize the incoming data in the NDDSCDRStream into the object instance when receiving a message from a peer. In *RTI Data Distribution Service* 3.x, nddsgen automatically creates this routine and provides the source code. In *RTI Data Distribution Service* 4.0, nddsgen will create the equivalent deserialization source code.

5.2.7.2 Print Routine

PrintMethodType — Provides the prototype for the print routine used to print the contents of the particular NDDSType. In *RTI Data Distribution Service* 3.x, the 'nddsgen' utility automatically creates this routine and provides the source code. In *RTI Data Distribution Service* 4.0, nddsgen will create the equivalent print source code.

5.2.7.3 Other Type-Related Routines

Unless otherwise noted, the *RTI Data Distribution Service* 4.0 'nddsgen' utility will create the equivalent source code for each of the following functions:

FreeMethodType — Provides the prototype for the routine required to free the instance of the particular NDDSType. In *RTI Data Distribution Service* 3.x, the 'nddsgen' utility automatically creates this routine prototype and stubs out the routine. The actual functionality of the call is left to the user to implement.

MaxSizeMethodType — Provides the prototype for the maximum size routine used to determine the maximum size of an NDDSType object. This routine is used to inform *RTI Data Distribution Service* how much buffer space to allocate for publications, subscriptions, servers, and clients that will use this NDDSType. In *RTI Data Distribution Service* 3.x, *RTI Data Distribution Service* calls this routine to decide

how much buffer space to allocate for an instance of this NDDSType. In *RTI Data Distribution Service* 3.x, the 'nddsgen' utility automatically creates this routine and provides the source code.

NddsTypeRegister — Provides the prototype to actually register an NDDSType to *RTI Data Distribution Service*. It binds the necessary (and some optional) routines to an NDDSType. In *RTI Data Distribution Service* 3.x, if you use the 'nddsgen' utility, you don't have to use this function directly because 'nddsgen' generates the required routines and creates a wrapper function to register the type.

NddsTypeDestroy — Provides the prototype to destroy all registered C types. All applications that register an NDDSType should call this function before exiting to ensure that the memory used to register the NDDSTypes is freed. Note: nddsgen in *RTI Data Distribution Service* 3.x does not generate this functionality.

5.2.7.4 WaveScope Registration Routines

These routines are not supported in *RTI Data Distribution Service* 4.0. They are mentioned here only for completeness:

ScopeRegisterMethodType — Provides the prototype that allows the NDDSType to be registered for use with the NDDSScope graphical debug tool. In *RTI Data Distribution Service* 3.x, the 'nddsgen' utility automatically creates this routine and provides the source code.

NddsTypeScopeRegister — Provides the prototype to register an NDDSType to NDDSScope, which is a graphical debug tool. This routine binds the optional scope register routine to an NDDSType. In *RTI Data Distribution Service* 3.x, if you use 'nddsgen', you don't have to use this function directly because 'nddsgen' generates the required routines and creates a wrapper function to register the scope register routine of the type.

5.2.8 Database API

The *RTI Data Distribution Service* C API provides a set of functions that allow you to manipulate the *RTI Data Distribution Service* internal database if so desired. These routines are not supported in *RTI Data Distribution Service* 4.0:

NddsDBaseRemove — Provides the prototype to retrieve and remove an object from the *RTI Data Distribution Service* internal database.

NddsDBaseFind — Provides the prototype to locate an object in the *RTI Data Distribution Service* internal database.

NddsDBaseAdd — Provides the prototype to add an object to the *RTI Data Distribution Service* internal database.

5.3 Publication API

This section discusses the *RTI Data Distribution Service* 3.x publication function calls and how they map to the *RTI Data Distribution Service* 4.0 functionality. As indicated earlier, some of the routines will map directly, others will require some redesign of the application. Table 5.6 lists the *RTI Data Distribution Service* 3.x Publication routines (in alphabetical order). Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality.

Table 5.6 **Publication C API**

3.x	Reference to 4.0 Information
NddsPublicationCreate	Section 5.3.1
NddsPublicationCreateAtomic	
NddsPublicationDestroy	
NddsPublicationInstanceGet	Section 5.3.5
NddsPublicationListenerGet	Section 5.3.6
NddsPublicationListenerSet	
NddsPublicationListenerDefaultGet	
NddsPublicationPropertiesGet	Section 5.3.2
NddsPublicationPropertiesSet	
NddsPublicationReliableStatusGet	Section 5.3.4
NddsPublicationTopicGet	Section 5.3.5
NddsPublicationSend	Section 5.3.3
NddsPublicationSubscriptionWait	
NddsPublicationWait	

5.3.1 Publication Create/Delete Routines

NddsPublicationCreateAtomic — Allows you to create a new publication and provide all the associated publication parameters at once.

In *RTI Data Distribution Service* 4.0, the equivalent functionality is supported by using both a DataWriter and a Publisher. So an indirect mapping to *RTI Data Distribution Service* 4.0 exists in that you create a DataWriter (DDS_Publisher_create_datawriter) after you create a Publisher (DDS_DomainParticipant_create_publisher). In *RTI Data Distribution Service* 4.0, atomic behavior is handled by default.

NddsPublicationCreate — A Publication is used by the application to write instances of data for publication.

In *RTI Data Distribution Service* 4.0, the equivalent functionality is supported by using both a DataWriter and a Publisher. So an indirect mapping to *RTI Data Distribution Service* 4.0 exists in that you create a DataWriter (DDS_Publisher_create_datawriter) after you create a Publisher (DDS_DomainParticipant_create_publisher).

NddsPublicationDestroy — As discussed in Section 2.3, an *RTI Data Distribution Service* 3.x Publication is used by the application to write instances of data for publication and does not require being added to a Publisher.

In *RTI Data Distribution Service* 4.0, this same functionality is supported by using both a DataWriter and a Publisher. If the *RTI Data Distribution Service* 3.x application in question is employing a Publisher, and only a specific Publication is to be destroyed, then you need to used the DDS_Publisher_delete_datawriter routine. If on the other hand, the *RTI Data Distribution Service* 3.x application was not employing a Publisher (only a Publication), then you need to delete both the *RTI Data Distribution Service* 4.0 Publisher and DataWriter by using DDS_DomainParticipant_delete_publisher and DDSPublisher_delete_datawriter.

5.3.2 Publication Properties Routines

NddsPublicationPropertiesGet — The properties associated with a Publication in *RTI Data Distribution Service* 3.x do not map directly to DataWriter properties in *RTI Data Distribution Service* 4.0, but object QoS properties can be retrieved for a given DataWriter by using the extended QoS DDS_Publisher_get_default_datawriter_qos routine.

NddsPublicationPropertiesSet — The properties associated with a Publication in *RTI Data Distribution Service* 3.x do not map directly to DataWriter properties in *RTI Data Distribution Service* 4.0, but object QoS properties can be established for a given DataWriter by using the extended QoS DDS_Publisher_set_default_datawriter_qos routine.

5.3.3 Publication Send and Wait Routines

NddsPublicationSend — Sends or writes the publication issue. In *RTI Data Distribution Service* 4.0, for purposes of discussion, we'll refer to a representative user-defined data type topic named HelloMsg. The application would then repeatedly call the HelloMsgDataWriter_write routine to disseminate the publication.

NddsPublicationWait — Waits for send queue level to reach the same or lower level specified within the Wait routine. There is no *RTI Data Distribution Service* 4.0 DataWriter or Publisher routine that will perform this specific functionality.

NddsPublicationSubscriptionWait — Waits for the existence of a specified number of Subscriptions. There is no *RTI Data Distribution Service* 4.0 DataWriter or Publisher routine that will perform this specific functionality.

5.3.4 Publication Status Routines

NddsPublicationReliableStatusGet — When a Publication is publishing to a reliable Subscription, NddsPublicationReliableStatusGet() provides detailed information pertaining to the reliable status of the Publication:

```
typedef struct NDDSPublicationReliableStatus {
         NDDSPublicationReliableEvent event;// The reliable event
         const char *nddsTopic;
         int unacknowledgedIssues;// Number of unacknowledged issues
         int subscriptionReliable;// number of reliable subscriptions
         int subscriptionUnreliable;// number of unreliable subscriptions
} NDDSPublicationReliableStatus;
```

- **event** — Provides the latest event on the publication's reliable stream, where the events are defined as:
  - **NDDS_BEFORERTN_VETOED —** The sendBeforeRtn vetoed the Publication. Data was not serialized. This information is unavailable in *RTI Data Distribution Service* 4.0.

- **NDDS_QUEUE_EMPTY** — The send queue is empty. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **NDDS_LOW_WATER_MARK** — The send queue level fell to the low water mark. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **NDDS_HIGH_WATER_MARK** — The send queue level rose to the high water mark. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **NDDS_QUEUE_FULL** — The send queue is full. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **NDDS_SUBSCRIPTION_NEW** — A new reliable subscription has appeared. The Built-in Topics feature in *RTI Data Distribution Service* 4.0 can provide this information. See Section 5.8.1.
- **NDDS_SUBSCRIPTION_DELETE** — A reliable Subscription disappeared. Note that the Publication only detects the disappearance of a reliable Subscription after the expirationTime of the last refreshed subscription declaration expires and the Publication checks its database. The Built-in Topics feature in *RTI Data Distribution Service* 4.0 can provide this information. See Section 5.8.1.

❑ **nddsTopic —** The NDDSTopic of the Publication. This information is unavailable in *RTI Data Distribution Service* 4.0.

❑ **subscriptionReliable** — The number of reliable Subscriptions subscribed to this publication. This information is unavailable in *RTI Data Distribution Service* 4.0.

❑ **subscriptionUnreliable** — The number of unreliable Subscriptions subscribed to this Publication. This information is unavailable in *RTI Data Distribution Service* 4.0.

❑ **unacknowledgedIssues** — The number of unacknowledged issues. This information is unavailable in *RTI Data Distribution Service* 4.0.

5.3.5 Publication Topic and Instance Routines

NddsPublicationTopicGet — Retrieves the Publication's Topic. In *RTI Data Distribution Service* 4.0, the DataWriter provides the DDS_DataWriter_get_topic routine which returns the Topic associated with the DataWriter. This is the same Topic that was used to create the DataWriter.

NddsPublicationInstanceGet — Obtains a pointer to the Publication instance. In *RTI Data Distribution Service* 4.0, a Publisher can use the DDS_Publisher_lookup_datawriter routine to obtain the DataWriter's instance.

5.3.6 Publication Listener Routines

NDDSPublicationListener — See Section 5.8.1.2 for further details.

NddsPublicationListenerGet — Retrieves the Publication's Listener hooks. In *RTI Data Distribution Service* 4.0, a DataWriter's Listener can be obtained by using the DDS_DataWriter_get_listener routine.

NddsPublicationListenerSet — Allows the Publication's Listener to be modified. In *RTI Data Distribution Service* 4.0, a DataWriter's Listener can be modified by using the DDSDataWriter_set_listener routine.

NddsPublicationListenerDefaultGet — Allows the Publication's default Listener hooks to be retrieved. In *RTI Data Distribution Service* 4.0, a DataWriter's default listener can be retrieved by invoking the DDS_DataWriter_get_listener routine prior to creating the DataWriter object.

5.4 Subscription API

This section discusses the *RTI Data Distribution Service* 3.x subscription function calls and how they map to the *RTI Data Distribution Service* 4.0 functionality. As indicated earlier, some of the routines will map directly, others will require some redesign of the application. Table 5.7 lists the *RTI Data Distribution Service* 3.x Subscription routines (in alphabetical order). Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality.

Table 5.7 **Subscription API**

3.x	Reference to 4.0 Information
NDDSIssueListener	Section 5.4.1
NddsSubscriptionDestroy	Section 5.4.2
NddsSubscriptionInstanceGet	Section 5.4.6

Table 5.7 **Subscription API**

3.x	Reference to 4.0 Information
NddsSubscriptionIssueListenerGet.	Section 5.4.1
NddsSubscriptionIssueListenerSet	
NddsSubscriptionPublicationWait	Section 5.4.5
NddsSubscriptionTopicGet	Section 5.4.6
NddsSubscriptionCreate	Section 5.4.2
NddsSubscriptionCreateAtomic	
NddsSubscriptionIssueListenerDefaultGet	Section 5.4.1
NddsSubscriptionPoll	Section 5.4.5
NddsSubscriptionPropertiesGet	Section 5.4.3
NddsSubscriptionPropertiesSet	Section 5.4.3
NddsSubscriptionReliableCreate	Section 5.4.7
NddsSubscriptionReliableCreateAtomic	
NDDSSubscriptionReliableListener	
NddsSubscriptionReliableListenerGet	
NddsSubscriptionReliableListenerSet	
NddsSubscriptionReliableStatusGet	
NDDSSubscriptionReliableStatusRtn	
NddsSubscriptionStatusGet	Section 5.4.4

5.4.1 Issue Listener Routines

NDDSIssueListener — See Section 5.8.3 for further details.

NddsSubscriptionIssueListenerDefaultGet — Allows the Subscripton's default Listener hooks to be retrieved. In *RTI Data Distribution Service* 4.0, a DataReader's default listener can be obtained by using the DDS_DataReader_get_listener routine prior to creating the DataReader object.

NddsSubscriptionIssueListenerGet — Retrieves the Subscription's Listener. In *RTI Data Distribution Service* 4.0, the DDS_DataReader_get_listener routine can be used to retrieve the DataReader's listener.

NddsSubscriptionIssueListenerSet — Modifies the Subscription's Listener. In *RTI Data Distribution Service* 4.0, the DDS_DataReader_set_listener routine can be used to modify the DataReader's listener.

5.4.2 Subscription Create/Delete Routines

NddsSubscriptionCreateAtomic — Allows the user to create a new subscription and provide all the associated subscription parameters at once. In *RTI Data Distribution Service* 4.0, the equivalent functionality is supported by creating both a DataReader and a Subscriber. So an indirect mapping to *RTI Data Distribution Service* 4.0 exists in that a DataReader would need to be created after the creation of a Subscriber. So both the DDS_DomainParticipant_create_subscriber and DDS_Subscriber_create_datareader routines are required. In *RTI Data Distribution Service* 4.0, atomic behavior is handled by default.

NddsSubscriptionCreate — As discussed in Section 2.4, a Subscription is supported by creating both a DataReader and a Subscriber. So an indirect mapping to *RTI Data Distribution Service* 4.0 exists in that a DataReader would need to be created after the Subscriber has been created. So both the DDS_DomainParticipant_create_subscriber and DDS_Subscriber_create_datareader routines are required.

NddsSubscriptionDestroy — As discussed in Section 2.4, an *RTI Data Distribution Service* 3.x Subscription can stand alone and refers to exactly one Topic that identifies the data to be read. *RTI Data Distribution Service* 3.x also allows the application to manage a group of Subscriptions with a Subscriber. If a Subscriber is used within the *RTI Data Distribution Service* 3.x application, and one of the Subscription's is to be destroyed, then one would use the DDS_Subscriber_delete_datareader routine. If on the other hand, the *RTI Data Distribution Service* 3.x application was not using a Subscriber (only a Subscription), then you would need to both delete the *RTI Data Distribution Service* 4.0 Subscriber and DataReader by using DDS_DomainParticipant_delete_subscriber and DDS_Subscriber_delete_datareader.

5.4.3 Subscription Properties Routines

NddsSubscriptionPropertiesSet — Modifies the current Subscription properties. The properties associated with a Subscription in *RTI Data Distribution Service* 3.x do not map directly to DataReader properties in *RTI Data Distribution Service* 4.0, but object QoS properties can be established for a given DataReader by using the DDS_Subscriber_set_default_datareader_qos routine.

NddsSubscriptionPropertiesGet — The properties associated with a Subscription in *RTI Data Distribution Service* 3.x do not map directly to DataReader properties in *RTI Data Distribution Service* 4.0, but object QoS properties can be retrieved for a given DataReader by using the DDS_Subscriber_get_default_datareader_qos routine.

5.4.4 Subscription Status Routine

NddsSubscriptionStatusGet — Allows the status of an issue received on a subscription to be examined. The information provided by this call is the same information that is presented to the Subscription's callback routine. This function simply allows the application to retrieve the Subscription's current status (see Section 5.8.3).

5.4.5 Subscription Poll and Wait Routines

NddsSubscriptionPoll — Polls the Subscription for newly received issues since the last poll.

In *RTI Data Distribution Service* 4.0, an application uses Listeners to receive data. Listeners provide asynchronous notification of data-sample arrival.

NddsSubscriptionPublicationWait — Actively probes for a given number of Publications for this Subscription. This functionality is not currently supported in *RTI Data Distribution Service* 4.0.

5.4.6 Subscription Topic and InstanceRoutines

NddsSubscriptionTopicGet — Returns the Topic subscribed to. In *RTI Data Distribution Service* 4.0, the DataReader provides the DDS_DataReader_get_topicdescription routine which returns the topic description associated with the DataReader.

NddsSubscriptionInstanceGet — Obtains a pointer to a Subscription's instance. In *RTI Data Distribution Service* 4.0, the DDS_Subscriber_lookup_datareader can be used to retrieve the instance of a DataReader attached to a Topic.

5.4.7 Reliable Subscription Routines

NDDSSubscriptionReliableStatusRtn — Provides the ability to monitor the status of the reliable stream on the subscription side. If one registers a subscription reliable status routine, *RTI Data Distribution Service* 3.1x will invoke the routine upon occurrence of events pertaining to the reliable communication on the subscription side. You can then take appropriate actions based on the event and other parameters. The routine provides the following:

- **event** — can be either:

- **NDDS_ISSUES_DROPPED** – One (or more) issues have been missed by the subscription. In *RTI Data Distribution Service* 4.0, the application can employ the Subscriber Listener on_sample_lost routine to determine information pertaining to dropped data issues. One can also access the Sample Lost Status (plain communication status type) allowing the application to determine the total cumulative count of all samples lost across all published instances of a specific Topic.
- **NDDS_PUBLICATION_NEW** – the reliable issue is coming from a publication different from the one that sent the previous issue. *This functionality is not currently supported in RTI Data Distribution Service 4.0.*

❑ **issuesDropped** — Provides the number of issues dropped. In *RTI Data Distribution Service* 4.0, the application can use the Subscriber Listener on_sample_lost routine or the DDS_DataReader_get_sample_lost_status routine to access both the total_count and total_count_change data fields. The total_count provides the cumulative count of all samples lost across all instances of topics subscribed to by this Subscriber. The total_count_change provides the incremental number of samples lost since the last time the Listener was called or the status was read. These routines provided dropped issue counts for the entire Subscriber, not on a DataReader basis.

❑ **nddsTopic** — Provides the subscription's topic. In *RTI Data Distribution Service* 4.0, once the Subscriber Listener's on_sample_lost routine is invoked, the information associated with dropped issues is provided only on a Subscriber basis. There is currently no mechanism available to determine which dropped issues are associated with which Topic.

NDDSSubscriptionReliableListener — See the NDDSSubscriptionReliableStatusRtn routine discussion above.

NddsSubscriptionReliableListenerSet — Allows the application to register a reliable listener class for a reliable subscription. The functionality provided by this listener has been discussed above with the NDDSSubscriptionReliableStatusRtn routine.

NddsSubscriptionReliableListenerGet — Retrieves the current reliable listener for a reliable subscription. In *RTI Data Distribution Service* 4.0, the DataReader Listener functionality does not map directly to what was provided in *RTI Data Distribution Service* 3.x, but nevertheless, the Listener can be retrieved by using the DDS_DataReader_get_listener routine.

NddsSubscriptionReliableCreateAtomic — Creates a reliable subscription with the desired properties and listener. In *RTI Data Distribution Service* 4.0, there is no separate API call to create a reliable Subscribers/DataReader. Simply use the routines identified above when instantiating a Subscription, then employ the

DDS_DataReader_set_qos routine to specify a QoS policy of DDS_QOS_RELIABILITY_RELIABLE. In *RTI Data Distribution Service* 4.0, atomic behavior is handled by default.

NddsSubscriptionReliableCreate — A reliable subscription is similar to a regular subscription except the issues are received reliably and in the order in which they were published. In *RTI Data Distribution Service* 4.0, there is no separate API call to create a reliable Subscriber/DataReader. Simply use the routines articulated above to instantiate a DataReader, then use the DDS_DataReader_set_qos routine to specify a QoS policy of DDS_QOS_RELIABILITY_RELIABLE.

NddsSubscriptionReliableStatusGet — See the NDDSSubscriptionReliableStatusRtn routine discussion above.

5.5 Publisher API

This section discusses the *RTI Data Distribution Service* 3.x Publisher routines and how they map to the *RTI Data Distribution Service* 4.0 functionality. Some of the routines will map directly, others will require redesign of the application. Table 5.8 lists the *RTI Data Distribution Service* 3.x Publisher routines (in alphabetical order). Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality.

Table 5.8 **Publisher API**

3.x	Reference to 4.0 Information
NddsCustomSocketPublisherCreate	Section 5.5.1
NddsPublisherCreate	
NddsPublisherDestroy	
NddsPublisherIterate	Section 5.5.4
NddsPublisherPublicationAdd	Section 5.5.2
NddsPublisherPublicationFind	Section 5.5.4
NddsPublisherPublicationRemove	Section 5.5.2
NddsPublisherSend	Section 5.5.3
NddsPublisherSubscriptionWait	

The Publisher object in *RTI Data Distribution Service* 3.x also supported Signaled and Asynchronous modes of operation; these are not supported in *RTI Data Distribution Service* 4.0.

5.5.1 Publisher Create/Delete Routines

NddsPublisherCreate — A Publisher manages a group of Publications. In *RTI Data Distribution Service* 4.0, the DDS_DomainParticipant_create_publisher routine can be used to create a Publisher. The application would then need to further create a DataWriter to allow the application to 'publish' data.

NddsCustomSocketPublisherCreate — Allows the application to create a publisher using a socket provided as a parameter. No such functionality currently exists within *RTI Data Distribution Service* 4.0.

NddsPublisherDestroy — In *RTI Data Distribution Service* 3.x this function destroys a publisher. It is important to note that Publications within the Publisher are NOT destroyed so they'll have to be destroyed separately. The actual deallocation of memory for the publisher may not occur immediately to ensure safety among the different tasks. After calling this function the Publisher is invalid and should not be used.

In *RTI Data Distribution Service* 4.0, the DDS_DomainParticipant_delete_publisher routine would be used after all associated DataWriter entities were destroyed via the DDS_Publisher_delete_datawriter. Note that within *RTI Data Distribution Service* 4.0, DataWriter's cannot exist without a Publisher, unlike *RTI Data Distribution Service* 3.x where a Publication can exist with or without a Publisher.

5.5.2 Publisher Add/Remove Routines

NddsPublisherPublicationAdd — Adds a Publication to a Publisher. In *RTI Data Distribution Service* 4.0, since a DataWriter cannot be instantiated independent of a Publisher, the equivalent functionality would be to use the DDS_Publisher_create_datawriter routine. This not only creates the DataWriter object, but adds it to the Publisher entity.

NddsPublisherPublicationRemove — Removes a Publication from being managed by a Publisher. In *RTI Data Distribution Service* 4.0, since a DataWriter must be associated with a Publisher, this functionality is not supported unless multiple DataWriters exist within the Publisher. If this is the case, then one of the DataWriter's may be removed by using the DDS_Publisher_delete_datawriter routine.

5.5.3 Publisher Send and Wait Routines

NddsPublisherSend — Takes a snapshot of all the Publications managed by the Publisher and then sends the issues at once, coalescing individual Publications into a single message to maximum network bandwidth utilization. A direct mapping of this routine does not exist within *RTI Data Distribution Service* 4.0. Each DataWriter's write routine must be invoked individually to cause the Topic issue to be disseminated via the Publisher.

NddsPublisherSubscriptionWait — Forces the calling thread to wait for at least the number of Subscriptions to appear for each Publication managed by the Publisher. There is no *RTI Data Distribution Service* 4.0 DataWriter or Publisher routine currently available that will perform this specific functionality.

5.5.4 Publisher Find and Iterate Routines

NddsPublisherPublicationFind — Finds the Publication, of a supplied Topic string, that is managed by the Publisher. In *RTI Data Distribution Service* 4.0, the DDS_Publisher_lookup_datawriter routine would be used. This routine allows the application to supply a Topic string so that the associated DataWriter handle can be retrieved.

NddsPublisherIterate — Iterates over all managed Publications. *This functionality is supported within the DDS Specification, but will not be implemented in RTI Data Distribution Service 4.0.*

5.6 Subscriber API

This section discusses the *RTI Data Distribution Service* 3.x Subscriber routines and how they map to the *RTI Data Distribution Service* 4.0 functionality. Some of the routines will map directly, others will require redesign of the application. Table 5.9 lists the *RTI Data Distribution Service* 3.x Subscriber routines (in alphabetical order). Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality.

Table 5.9 **Subscriber API**

3.x	Reference to 4.0 Information
NDDSSubscriberCreate	Section 5.6.1
NDDSSubscriberDestroy	
NDDSSubscriberIterate	Section 5.6.5
NDDSSubscriberPatternAdd	Section 5.6.2
NDDSSubscriberPatternRemove	
NDDSSubscriberPoll	Section 5.6.4
NDDSSubscriberSubscriptionAdd	Section 5.6.3
NDDSSubscriberSubscriptionFind	Section 5.6.5
NDDSSubscriberSubscriptionRemove	Section 5.6.3

5.6.1 Subscriber Create/Delete Routines

NDDSSubscriberCreate — A Subscriber manages a group of Subscriptions. In *RTI Data Distribution Service* 4.0, the DDS_DomainParticipant_create_subscriber routine would be used. The application would then need to further create a DataReader to allow the application to 'subscribe' to the data of interest.

NDDSSubscriberDestroy — Destroys a Subscriber. Subscriptions that have been added with the NDDSSubscriberSubscriptionAdd routine are not destroyed, but Subscriptions automatically created through Pattern Subscription are destroyed. In *RTI Data Distribution Service* 4.0, the DDS_DomainParticipant_delete_subscriber routine would be used after all associated DataReader entities were destroyed via the DDS_Subscriber_delete_datareader routine. In *RTI Data Distribution Service* 4.0, DataReaders cannot exist without a Subscriber, unlike *RTI Data Distribution Service* 3.x where a Subscription can exist with or without a Subscriber.

5.6.2 Subscriber Pattern Routines

These routines have no equivalent mapping in *RTI Data Distribution Service* 4.0.

NDDSSubscriberPatternAdd — Adds a Pattern Subscription to the Subscriber. Patterns allow users to subscribe to a large set of publications. Pattern Subscriptions differ from single Subscriptions in that the topicPattern and typePattern are string patterns. Patterns usually contain wild characters such as "*". If you specify the sub-

scription type NDDS_SUBSCRIPTION_POLLED then you must call NddsSubscriberPoll to receive the issues. OnMatch() is called if the subscriber doesn't already contain the subscription.

NDDSSubscriberPatternRemove — Removes a previously added pattern, but does not delete the pattern listener passed in during the pattern registration.

5.6.3 Subscriber Add/Remove Routines

NDDSSubscriberSubscriptionAdd — Adds a Subscription to the Subscriber. In *RTI Data Distribution Service* 4.0, since a DataReader cannot be instantiated independent of a Subscriber, the equivalent routine would be to use the DDS_Subscriber_create_datareader routine. This not only creates the DataReader object, but automatically adds it to the Subscriber.

NDDSSubscriberSubscriptionRemove — Removes a Subscription from being managed by a Subscriber. In *RTI Data Distribution Service* 4.0, since a DataReader must be associated with a Subscriber, this functionality is not supported, unless there are multiple DataReaders created within the Subscriber. If this is the case, then one of the DataReader's may be removed by using the DDS_Subscriber_delete_datareader routine.

5.6.4 Subscriber Poll Routine

NDDSSubscriberPoll — Polls all Subscriptions. In *RTI Data Distribution Service* 4.0, there is no routine that provides this equivalent functionality directly. The user could create specific conditions to wait on, and attach them to a Wait-Set for each DataReader of interest. The application could then achieve polling functionality by repeatedly invoking the DDS_WaitSet_wait routine with a specified timeout value.

5.6.5 Subscriber Find and Iterate Routines

NDDSSubscriberSubscriptionFind — Finds the Subscription, of a supplied Topic string, that is managed by the Subscriber. In *RTI Data Distribution Service* 4.0, the DDS_Subscriber_lookup_datareader routine would be used. This function allows the application to supply a Topic string so that the associated DataReader handle can be retrieved.

NDDSSubscriberIterate — Iterates over all aggregated Subscriptions. *This functionality is supported within the DDS Specification, but will not be implemented in RTI Data Distribution Service 4.0.*

5.7 Client and Server APIs

None of the Client and Server APIs are supported in *RTI Data Distribution Service* 4.0. This feature set will be introduced in a future release of the product.

5.8 Listeners

Listeners provide a mechanism for *RTI Data Distribution Service* to asynchronously alert the application of the occurrence of relevant asynchronous events, such as arrival of data corresponding to a Subscription. Listeners are callback routines that the application implements. Each dedicated listener presents a list of callback functions that correspond to relevant events that the application may wish to respond to.

To continue with our API comparison, let's examine *RTI Data Distribution Service* listeners. Recall that in *RTI Data Distribution Service* 3.x, Listeners can be associated with a Domain, Publication, and Subscription. The DDS specification indicates that all DCPS entities support their own specialized listener, so *RTI Data Distribution Service* 4.0 will provide Listener support for each Entity.

5.8.1 Domain Listeners

The Domain Listener in *RTI Data Distribution Service* 3.x allows the application the ability to be notified upon the appearance and disappearance of Managers, Applications, Publications, Subscriptions, and Servers. This section looks at the *RTI Data Distribution Service* 3.x Domain Listener callback functions and how they can be mapped into the *RTI Data Distribution Service* 4.0 functionality.

The DomainParticipant Listener in *RTI Data Distribution Service* 4.0 does not provide the same callback function prototypes that are provided in the *RTI Data Distribution Service* 3.x Domain Listener. In order for an *RTI Data Distribution Service* 4.0-based application to implement similar *RTI Data Distribution Service* 3.x Domain Listener functionality, the 'built-in Topic' must be used. The DDS specification introduces a set of Built-in Topics

and corresponding DataReader objects that can be used by the application to monitor and keep track of new DCPS entities as they are discovered. The Built-in Topics can then be accessed.

5.8.1.1 Domain Listener Callback Routines

There is no direct mapping of these routines to *RTI Data Distribution Service* 4.0, but equivalent functionality can be implemented using *RTI Data Distribution Service* 4.0, as described below:

NDDSOnApplicationRemoteNewHook — Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSParticipant. The application can then gain access to the built-in Subscriber and associated DataReaders by using the DDS_DomainParticipant_get_builtin_subscriber routine provided by the DomainParticipant. The built-in DataReader objects can then be retrieved by using the DDS_Subscriber_get_datareaders routine. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote participant to participate within the network by using the DDS_DomainParticipant_ignore_participant routine.

NDDSOnApplicationRemoteDeleteHook — Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSParticipant. The application can then gain access to the built-in Subscriber and associated DataReaders by using the DDS_DomainParticipant_get_builtin_subscriber routine provided by the DomainParticipant. The built-in DataReader objects can then be retrieved by using the DDS_Subscriber_get_datareaders routine. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote participant to participate within the network by using the DDS_DomainParticipant_ignore_participant routine.

NDDSOnPublicationRemoteNewHook — Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSPublication. The application can then gain access to the built-in Subscriber and associated DataReaders by using the DDS_DomainParticipant_get_builtin_subscriber routine provided by the DomainParticipant. The built-in DataReader objects can then be retrieved by using the DDS_Subscriber_get_datareaders routine. This allows the application to gain access to the Built-in DCPSPublication Topic and monitor all traffic related to remote DataWriter activity. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote publisher to participate within the network by using the DDS_DomainParticipant_ignore_publication routine.

NDDSOnPublicationRemoteDeleteHook — Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSPublication. The application can then gain access to the built-in Subscriber and associated DataReaders by using the DDS_DomainParticipant_get_builtin_subscriber routine provided by the DomainParticipant. The built-in DataReader objects can then be retrieved by using the DDS_Subscriber_get_datareaders routine. This allows the application to gain access to the Built-in DCPSPublication Topic and monitor all traffic related to remote DataWriter activity. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote publisher to participate within the network by using the DDS_DomainParticipant_ignore_publication routine.

NDDSOnServerRemoteNewHook — There is no functionally equivalent capability within *RTI Data Distribution Service* 4.0 as client/server is not support. This feature set is planned for a future release.

NDDSOnServerRemoteDeleteHook — There is no functionally equivalent capability within *RTI Data Distribution Service* 4.0 as client/server is not support. This feature set is planned for a future release.

NDDSOnSubscriptionRemoteNewHook — Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSSubscription. The application can then gain access to the built-in Subscriber and associated DataReaders by using the DDS_DomainParticipant_get_builtin_subscriber routine provided by the DomainParticipant. The built-in DataReader objects can then be retrieved by using the DDS_Subscriber_get_datareaders routine. This allows the application to gain access to the Built-in DCPSSubscription Topic and monitor all traffic related to remote DataReader activity. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote subscriber to participate within the network by using the DDS_DomainParticipant_ignore_subscription routine.

NDDSOnSubscriptionRemoteDeleteHook — Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSSubscription. The application can then gain access to the built-in Subscriber and associated DataReaders by using the DDS_DomainParticipant_get_builtin_subscriber routine provided by the DomainParticipant. The built-in DataReader objects can then be retrieved by using the DDS_Subscriber_get_datareaders routine. This allows the application to gain access to the Built-in DCPSSubscription Topic and monitor all traffic related to remote DataReader activity. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote subscriber to participate within the network by using the DDS_DomainParticipant_ignore_subscription routine.

5.8.1.2 Domain Listener Retrieval Routines

NddsDomainListenerDefaultGet — Retrieves the domain's default listener hooks. Default callbacks are necessary to initialize the domain listener structure before one can modify the fields for application specific functionality. In *RTI Data Distribution Service* 4.0, this domain listener functionality is supported via built-in topics.

NddsDomainListenerGet — Retrieves the default domain listener hooks. Sets the hooks called whenever declarations for remote publications, subscriptions, clients, and servers are received. In *RTI Data Distribution Service* 4.0, this domain listener functionality is supported via built-in topics.

5.8.2 Publication Listeners

The *RTI Data Distribution Service* 3.x Publication Listener allows an application to tailor its behavior in response to middleware activity associated with individual publication events. The Publication Listener interface provides the following callback functions:

NDDSAfterSendRtn — This routine, if implemented by the application, is invoked by *RTI Data Distribution Service* directly after an issue is sent. In *RTI Data Distribution Service* 4.0, there is currently no direct mapping of this functionality.

NDDSSendBeforeRtn — This routine, if implemented by the application, is invoked by *RTI Data Distribution Service* prior to an issue being sent. In *RTI Data Distribution Service* 4.0, there is currently no direct mapping of this functionality.

NddsPublicationReliableStatusRtn — When a Publication is publishing to at least one reliable Subscription, this routine provides detailed information pertaining to the reliable status of the Publication. The following is a list of the status information the routine returns:

- **event** — provides the latest event on the publication's reliable stream where the events are defined as:
  - **NDDS_BEFORERTN_VETOED** — the sendBeforeRtn vetoed the Publication. Data was not serialized. This information is not available in *RTI Data Distribution Service* 4.0.
  - **NDDS_QUEUE_EMPTY** — the send queue is empty. This information is not available in *RTI Data Distribution Service* 4.0.

- **NDDS_LOW_WATER_MARK** — the send queue level fell to the low water mark. If the low water mark is 0, only NDDS_QUEUE_EMPTY will be called when the queue becomes empty. This information is not available in *RTI Data Distribution Service* 4.0.
- **NDDS_HIGH_WATER_MARK** — the send queue level rose to the high water mark. If the high water mark is the same as the send queue size, only NDDS_QUEUE_FULL will be called when the queue becomes full. This information is not available in *RTI Data Distribution Service* 4.0.
- **NDDS_QUEUE_FULL** — the send queue is full. This information is not available in *RTI Data Distribution Service* 4.0.
- **NDDS_SUBSCRIPTION_NEW** — a new reliable subscription has appeared. The Built-in Topics feature in *RTI Data Distribution Service* 4.0 can provide this information. See Section 5.8.1.
- **NDDS_SUBSCRIPTION_DELETE** — a reliable Subscription disappeared. Note that the Publication only detects the disappearance of a reliable Subscription after the expirationTime of the last refreshed subscription declaration expires and the Publication checks its database. The Built-in Topics feature in *RTI Data Distribution Service* 4.0 can provide this information.

❑ **nddsTopic** — the NDDSTopic of the Publication. This information is not available in *RTI Data Distribution Service* 4.0.

❑ **subscriptionReliable** — the number of reliable Subscriptions subscribed to this publication. This information is not available in *RTI Data Distribution Service* 4.0.

❑ **subscriptionUnreliable** — the number of unreliable Subscriptions subscribed to this Publication. This information is not available in *RTI Data Distribution Service* 4.0.

❑ **unacknowledgedIssues** — the number of unacknowledged issues. This information is not available in *RTI Data Distribution Service* 4.0.

5.8.3 Issue Listeners

The *RTI Data Distribution Service* 3.x Issue Listener provides the ability for the application to tailor its behavior in response to middleware activity associated with individual subscription events, including the reception of publications. The Issue Listener interface provides the following callback function prototype:

NDDSRecvCallBackRtn — this routine is provided by the application and is registered when the subscription is created and is invoked by *RTI Data Distribution Service* at different times depending on the subscription mode. If configured for IMMEDIATE Subscription, this routine is invoked as soon as the data issue is received. If configured for POLLED Subscription, this routine is invoked when the receiving application explicitly polls. If there are more than one issues received since the last poll, this routine will be executed multiple times for each issue. In *RTI Data Distribution Service* 4.0, the DataReaderListener's on_data_available routine would be used to receive incoming data. The NDDSRecvCallBackRtn's function prototype provides access to both received issue data and issue data status. The status that is available on a per issue basis is listed below:

- **localTimeWhenReceived** — Local time when the issue was received. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **nddsTopic** — tTopic of the Subscription receiving the issue. In *RTI Data Distribution Service* 4.0, the DDS_DataReader_get_topicdescription routine can be used to determine the DataReader's topic.
- **nddsType** — Type of the Subscription receiving the issue. This information is not currently available in *RTI Data Distribution Service* 4.0.
- **publicationId** — Publication's unique ID. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **publSeqNumber** — Sending (Publication) high and low sequence number. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **recvSeqNumber** — Receiving sequence high and low sequence number. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **remoteTimeWhenPublished** — Remote time when the issue was published. Once the read or take routine is used within the DataReader to gain access to the received data issue, the SampleInfo source_timestamp routine can be employed which provides the time-stamp provided by the DataWriter at the time the sample was produced.
- **senderAppId** — Sender's application ID. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **senderHostId** — Sender's host ID. This information is unavailablein *RTI Data Distribution Service* 4.0.
- **senderNodeIP** — Sender's IP address. This information is unavailable in *RTI Data Distribution Service* 4.0.

- **validRemoteTimeWhenPublished** — Whether or not a valid remote time was received. This information is unavailable in *RTI Data Distribution Service* 4.0.
- **status** — Status affects which fields are valid and returns:
  - **NDDS_DESERIALIZATION_ERROR** — Deserialization routine for the NDDS-Type returned an error. This information is not available in *RTI Data Distribution Service* 4.0.
  - **NDDS_FRESH_DATA** — A new issue received. In *RTI Data Distribution Service* 4.0, the application can determine the status of the received issue by inspecting the information provided by SampleInfo. SampleInfo information is provided along with each data issue and provides detailed information pertaining to that data instance. One can determine the state of the arriving issue by taking advantage of the information provided by the sample_state (READ or NOT_READ), view_state (NEW or NOT_NEW), and instance_state (ALIVE, NOT_ALIVE_DISPOSED, or NOT_ALIVE_NO_WRITERS).
  - **NDDS_NEVER_RECEIVED_DATA** — Never received an issue, but a deadline occurred. In *RTI Data Distribution Service* 4.0, one would use the DataReader Listener on_requested_deadline_missed routine. *RTI Data Distribution Service* will invoke this operation when the deadline has been missed. The application can also consult the entities status by directly using the DDS_DataReader_get_requested_deadline_missed_status routine.
  - **NDDS_NO_NEW_DATA** — Received at least one issue, and a deadline has occurred since the last issue was received. One can use a combination of features described above.
  - **NDDS_UPDATE_OF_OLD_DATA** — Received a new issue, whose time stamp is the same or older than the time stamp of the last fresh issue received. This information is unavailable in *RTI Data Distribution Service* 4.0.

5.8.4 Subscription Reliable Listeners

The *RTI Data Distribution Service* 3.x Subscription Reliable Listener provides the ability for the application to tailor its behavior in response to activity associated with individual reliable subscription events. The Reliable Subscription Listener interface defines the following function prototype:

NDDSSubscriptionReliableStatusRtn — Provides the ability to monitor the status of a reliable stream on the subscribing node. If you register a subscription reliable status routine in *RTI Data Distribution Service* 3.x, *RTI Data Distribution Service* will invoke the routine upon occurrence of events pertaining to the reliable communication on the subscription side. The NDDSSubscriptionReliableStatusRtn provides access to the following information:

- **event** — Provides the event concerning the reliable subscription. The event can be either:
  - **NDDS_ISSUES_DROPPED** — One (or more) issues have been missed by the subscription. In *RTI Data Distribution Service* 4.0, the application can employ the Subscriber Listener on_sample_lost routine to determine information pertaining to dropped data issues.
  - **NDDS_PUBLICATION_NEW** — The reliable issue is coming from a publication different from the one that sent the previous issue. This functionality is not currently supported in *RTI Data Distribution Service* 4.0
- **issuesDropped** — Number of issues dropped. In *RTI Data Distribution Service* 4.0, the application can use the Subscriber Listener on_sample_lost routine or the DDS_DataReader_get_sample_lost_status routine to access both the total_count and total_count_change data fields. The total_count provides the cumulative count of all samples lost across all instances of topics subscribed to by this Subscriber. The total_count_change provides the incremental number of samples lost since the last time the Listener was called or the status was read. These routines provided dropped issue counts for the entire Subscriber, not on a DataReader basis.
- **nddsTopic** — Subscription's topic. In *RTI Data Distribution Service* 4.0, once either the Subscriber Listener on_sample_lost routine or the DDS_DataReader_get_sample_lost_status routine is invoked, the information associated with dropped issues is provided only on a Subscriber basis. There is currently no mechanism available to determine which dropped issues are associated with which Topic.

5.8.5 Publisher and Subscriber Listeners

The *RTI Data Distribution Service* 3.x product does not support Publisher or Subscriber Listener interfaces.

5.8.6 Client and Server Listeners

The *RTI Data Distribution Service* 3.x Client and Server Listeners allow an application to tailor its behavior in response to middleware activity associated with individual Client and Server events. *RTI Data Distribution Service* 4.0 does not support a Client-Server API.

Chapter 6

Comparing the C++ APIs

This chapter describes how the *RTI Data Distribution Service* 3.x C++ API maps to the *RTI Data Distribution Service* 4.0 API. It focuses on how to port an *RTI Data Distribution Service* 3.x C++ application to *RTI Data Distribution Service* 4.0. This chapter addresses each *RTI Data Distribution Service* 3.x object and its supported routines and attempts to map the functionality to equivalent *RTI Data Distribution Service* 4.0 routines and/or functionality. Where direct or indirect mappings do not exist, we'll recommend alternate approaches for you to consider. Example source code (using both versions) will be used and will assume best-effort QoS and unicast network addressing. For additional examples of reliable communications, see Appendix A, which lists the buildable example source code that is available online.

This chapter includes the following sections:

6.1 Examples

We'll start with examples of how to instantiate a domain, and send and receive data.

6.1.1 Domain Instantiation

RTI Data Distribution Service 3.x:

```
int                             nddsDomain = NDDS_DOMAIN_DEFAULT;
int                             nddsVerbosity = NDDS_VERBOSITY_DEFAULT;
NDDSDomainClass                 *domain = NULL;

NddsVerbositySet(nddsVerbosity);

if(!(domain = NDDSDomainClass::Create(nddsDomain, NULL, NULL))) {
   printf("Unable to create the domain.\n");
   return RTI_FALSE;
};
```

RTI Data Distribution Service 4.0:

```
DDSDomainParticipantFactory   *factory = NULL;
DDS_DomainParticipantQos      participant_qos;
DDSDomainParticipant          *participant = NULL;
int                           nddsDomain = 0;
int                           participantIndex = 0;
char                          *peerHost = "10.10.10.1";
int                           peerMaxIndex     = 1;

if(!(factory = DDSDomainParticipantFactory::get_instance())) {
    return RTI_FALSE;
}

factory->get_default_participant_qos(participant_qos);
participant_qos.discoveryX.participant_index = participantIndex;
if (peerHost != NULL) {
    if (!RTINetioAddress_getIpv4AddressByName((RTINetioAddress
            *)&participant_qos.discoveryX.initial_peer_locators[0].address,
        peerHost)) {
         return RTI_FALSE;
    }
    participant_qos.discoveryX.initial_peer_locators[0].participant_index_limit
        = peerMaxIndex;
    participant_qos.discoveryX.initial_peer_locators_count = 1;
}
```

```
if(!(participant = factory->create_participant(nddsDomain, participant_qos,
        NULL))) {
    return RTI_FALSE;
}
```

Notice that there are a few more 'infrastructure' calls that must be used in *RTI Data Distribution Service* 4.0 prior to the 'create_participant' method that instantiates the actual DomainParticipant. The *RTI Data Distribution Service* 3.x and 4.0 methods used to instantiate a domain are shown in Table 6.1 for comparison purposes.

Table 6.1 **C++ Methods for Creating a Domain**

3.x Method	4.0 Method
NDDSDomainClass::Create	DDSDomainParticipantFactory::get_instance
	factory->get_default_participant_qos
	factory->create_participant

Only a few methods are required to actually create the Domain itself. Once the Domain exists, the application is provided a rich set of methods that can be used to support application functionality. Below you'll find example source code of how an application publishes and subscribes data using both *RTI Data Distribution Service* 3.x and 4.0. We'll cover each of the Domain methods available in *RTI Data Distribution Service* 3.x and discuss how they map to the *RTI Data Distribution Service* 4.0 DDS compliant product.

6.1.2 Publishing Data

In *RTI Data Distribution Service* 3.x, a Publication can stand alone. It can also be added to a Publisher if so desired. As you'll see in *RTI Data Distribution Service* 4.0 source code, the Publisher is initially instantiated, with the DataWriter creation to follow.

RTI Data Distribution Service 3.x:

```
extern "C" int publisherMain(int nddsDomain, int nddsVerbosity)
{
   int                         count = 0;
   RTINtpTime                  send_period_sec = {0,0};
   NDDSPublicationProperties   properties;
   NDDSDomainClass             *domain = NULL;
   HelloMsg                    *instance = NULL;
   NDDSPublicationClass        *publication = NULL;
   NDDSPublisherClass          *publisher = NULL;

   RtiNtpTimePackFromNanosec(send_period_sec, 4, 0); /* 4 seconds */
   NddsVerbositySet(nddsVerbosity);
```

```
    // Create the domain.
    if(!(domain = NDDSDomainClass::Create(NDDS_DOMAIN_DEFAULT, NULL, NULL))) {
       printf("Unable to create domain.\n");
       return RTI_FALSE;
    };

    if(!(instance = new HelloMsg())) {
       return RTI_FALSE;
    };

    domain->PublicationPropertiesGet(&properties);
    RtiNtpTimePackFromNanosec(properties.persistence, 15, 0); /* 15 seconds */
    properties.strength = 1;

    if(!(publication = domain->PublicationCreate("Example HelloMsg", instance,
         &properties))) {
       printf("Unable to create publication\n");
       return RTI_FALSE;
    };

    publisher = domain->PublisherCreate(NDDS_PUBLISHER_SIGNALLED);
    publisher->PublicationAdd(publication);

    for (count=0;;count++) {
       printf("Sampling publication, count %d\n", count);

       /* modify the data to be sent here */
       sprintf(instance->msg, "Hello Universe! (%d)", count);

       publisher->Send();
       NddsUtilitySleep(send_period_sec);
     }
     return RTI_TRUE;
};
```

RTI Data Distribution Service 4.0:

```
extern "C" int publisherMain(int nddsDomain, int participantIndex, const char
        *peerHost, int peerMaxIndex)
{
    DDSDomainParticipantFactory*factory = NULL;
    DDS_DomainParticipantQos  participant_qos;
    DDSDomainParticipant              *participant = NULL;
    DDSPublisher              *publisher = NULL;
    DDSTopic                  *topic = NULL;
    HelloMsgDataWriter        *writer = NULL;
    HelloMsg                  *instance = NULL;
    DDS_ReturnCode_t          retcode;
    DDS_InstanceHandle_t              instance_handle = DDS_HANDLE_NIL;
```

```
int                         count = 0;
RTINtpTime                  send_period_sec = {0,0};

RtiNtpTimePackFromNanosec(send_period_sec, 4, 0); /* 4 seconds */

if(!(factory = DDSDomainParticipantFactory::get_instance())) {
    return RTI_FALSE;
}

factory->get_default_participant_qos(participant_qos);
participant_qos.discoveryX.participant_index = participantIndex;
if (peerHost != NULL) {
    if (!RTINetioAddress_getIpv4AddressByName((RTINetioAddress
            *)&participant_qos.discoveryX.initial_peer_locators[0].address,
    peerHost)) {
        return RTI_FALSE;
    }
    participant_qos.discov-
    eryX.initial_peer_locators[0].participant_index_limit = peerMaxIndex;
    participant_qos.discoveryX.initial_peer_locators_count = 1;
}
if(!(participant = factory->create_participant(nddsDomain, participant_qos,
    NULL))) {
    return RTI_FALSE;
}

if(!(publisher = participant->create_publisher())) {
    return RTI_FALSE;
}

retcode = HelloMsgTypeSupport::register_type(participant);
if (retcode != DDS_RETCODE_OK) {
    return RTI_FALSE;
}

if(!(topic = participant->create_topic("Example HelloMsg", HelloMsgTYPE-
    NAME))) {
    return RTI_FALSE;
}

if(!(writer = (HelloMsgDataWriter *)publisher->create_datawriter(topic))) {
    return RTI_FALSE;
}

if(!(instance = HelloMsgTypeSupport::createX())) {
    return RTI_FALSE;
}
```

```
    for (count=0;count>=0;count++) {
        RtiDebugPrint("C++ API: Publishing best-effort/unicast example, count
        %d\n", count);

        /* modify the data to be sent here */
        sprintf(instance->msg, "C++ API: Publishing best-effort/unicast exam-
        ple, count %d", count);

        retcode = writer->write(*instance, instance_handle);
        if (retcode != DDS_RETCODE_OK) {
            return RTI_FALSE;
        }
        RtiThreadSleep(&send_period_sec);
    }

    return RTI_TRUE;
};
```

The *RTI Data Distribution Service* 3.x and 4.0 methods used in the above example are shown in Table 6.2 for comparison purposes.

Table 6.2 **C++ Methods for Publishing Data**

3.x Method	4.0 Method
	DDSDomainParticipantFactory::get_instance
	factory->get_default_participant_qos
NDDSDomainClass::Create	factory->create_participant
domain->PublicationPropertiesGet	
domain->PublicationCreate	
domain->PublisherCreate	participant->create_publisher
	HelloMsgTypeSupport::register_type
	participant->create_topic
	publisher->get_default_datawriter_qos
publisher->PublicationAdd	publisher->create_datawriter
	HelloMsgTypeSupport::createX
publisher->Send	writer->write

Compilable versions of both the *RTI Data Distribution Service* 3.x and 4.0 source code shown above, as well as other examples, are available for download at the RTI website. A list of example, fully-compilable source code projects is available in Appendix A.

6.1.3 Subscribing to Data

In *RTI Data Distribution Service* 3.x, a Subscription can stand alone. It can also be added to a Subscriber if so desired. As you'll see in the *RTI Data Distribution Service* 4.0 source code, the Subscriber is initially instantiated, with the DataReader creation to follow.

RTI Data Distribution Service 3.x:

```
class MyListener : public HelloMsgListener {
public:
  virtual RTIBool OnIssueReceived(const NDDSRecvInfo *issue,
                                  class NDDSTypeClass *instance);
};

RTIBool MyListener::OnIssueReceived(const NDDSRecvInfo *issue,
                                    class NDDSTypeClass *instance)
{
   if (issue->status == NDDS_FRESH_DATA) {
      instance->Print(0);
      return RTI_TRUE;
   }
};

extern "C" int subscriberMain(int nddsDomain, int nddsVerbosity)
{
   NDDSSubscriptionProperties properties;
   NDDSSubscriptionClass*subscription = NULL;
   MyListener                 *listener = NULL;
   NDDSDomainClass            *domain = NULL;
   HelloMsg                   *instance = NULL;
   NDDSSubscriberClass        *subscriber = NULL;
   char                       deadlineString[RTI_NTP_TIME_STRING_LEN];

   NddsVerbositySet(nddsVerbosity);

   if(!(domain = NDDSDomainClass::Create(NDDS_DOMAIN_DEFAULT,NULL,NULL))) {
      printf("Unable to create the domain.\n");
      return RTI_FALSE;
   };

   domain->SubscriptionPropertiesGet(&properties);
   RtiNtpTimePackFromNanosec(properties.minimumSeparation,0,0);
   RtiNtpTimePackFromNanosec(properties.deadline,10,0);
   properties.mode = NDDS_SUBSCRIPTION_IMMEDIATE;

   if(!(instance = new HelloMsg())) {
      return RTI_FALSE;
   }
```

```
    if(!(listener = new MyListener())) {
        return RTI_FALSE;
    }

    if(!(subscription = domain->SubscriptionCreate("Example HelloMsg", instance,
         listener, &properties, NDDS_USE_UNICAST))){
        return RTI_FALSE;
    }

    subscriber = domain->SubscriberCreate();
    subscriber->SubscriptionAdd(subscription);

    while (1) {
        printf("Sleeping for %s sec...\n",
        RtiNtpTimeToString(&properties.deadline, deadlineString));
        NddsUtilitySleep(properties.deadline);
    }
    return RTI_TRUE;
};
```

RTI Data Distribution Service 4.0:

```
class MyListener : public DDSDataReaderListener {
  public:
    virtual void on_data_available(DDSDataReader* reader);
};

void MyListener::on_data_available(DDSDataReader* reader)
{
    HelloMsgDataReader *HelloMsgreader = (HelloMsgDataReader *)reader;
    HelloMsgSeq       data_seq;
    DDS_SampleInfoSeq info_seq;
    DDS_ReturnCode_t  retcode;
    int               i;

    retcode = HelloMsgreader->take(data_seq, info_seq, DDS_LENGTH_UNLIMITED,
            DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);
    if (retcode != DDS_RETCODE_OK) {
        return;
    }

    for (i = 0; i < data_seq.length(); ++i) {
        HelloMsgTypeSupport::printX(&data_seq[i]);
    }

    HelloMsgreader->return_loan(data_seq, info_seq);
}
```

```
extern "C" int subscriberMain(int nddsDomain, int participantIndex, const char
        *peerHost, int peerMaxIndex)
{
    DDSDomainParticipantFactory *factory = NULL;
    DDS_DomainParticipantQos  participant_qos;
    DDSDomainParticipant      *participant = NULL;
    DDSSubscriber             *subscriber = NULL;
    DDSTopic                  *topic = NULL;
    MyListener                *listener = NULL;
    HelloMsgDataReader        *reader = NULL;
    DDS_ReturnCode_t          retcode;
    RTINtpTime                receive_period_sec = {0,0};
    Char                      deadlineString[RTI_NTP_TIME_STRING_LEN];
    int                       count = 0;

    RtiNtpTimePackFromNanosec(receive_period_sec, 4, 0); /* 4 seconds */

    if(!(factory = DDSDomainParticipantFactory::get_instance())) {
        return RTI_FALSE;
    }

    factory->get_default_participant_qos(participant_qos);
    participant_qos.discoveryX.participant_index = participantIndex;
    if (peerHost != NULL) {
        if (!RTINetioAddress_getIpv4AddressByName((RTINetioAddress *)
                &participant_qos.discoveryX.initial_peer_locators[0].address,
                 peerHost)) {
            return RTI_FALSE;
        }
       participant_qos.discov-
        eryX.initial_peer_locators[0].participant_index_limit = peerMaxIndex;

       participant_qos.discoveryX.initial_peer_locators_count = 1;
    }

    if(!(participant = factory->create_participant(nddsDomain, participant_qos,
               NULL))) {
        return RTI_FALSE;
    }

    if(!(subscriber = participant->create_subscriber())) {
        return RTI_FALSE;
    }

    retcode = HelloMsgTypeSupport::register_type(participant);
    if (retcode != DDS_RETCODE_OK) {
        return RTI_FALSE;
    }
```

```
    if(!(topic = participant->create_topic("Example HelloMsg",
                HelloMsgTYPENAME))) {
        return RTI_FALSE;
    }

    if(!(listener = new MyListener())) {
        return RTI_FALSE;
    }

    if(!(reader = (HelloMsgDataReader *)subscriber->create_datareader(topic,
       DDS_DATAREADER_QOS_DEFAULT, listener))) {
        return RTI_FALSE;
    }

    for (count=0;count>=0;count++) {
        RtiDebugPrint("Sleeping for %s sec...\n", RtiNtpTimeTo-
       String(&receive_period_sec, deadlineString));
        RtiThreadSleep(&receive_period_sec);
    }
    return RTI_TRUE;
};
```

The *RTI Data Distribution Service* 3.x and 4.0 methods used are shown in Table 6.3 for comparison purposes.

Table 6.3 **C++ Methods for Subscribing to Data**

3.x Method	4.0 Method
	DDSDomainParticipantFactory::get_instance
	factory->get_default_participant_qos
NDDSDomainClass::Create	factory->create_participant
domain->SubscriptionPropertiesGet	
domain->SubscriptionCreate	
domain->SubscriberCreate	participant->create_subscriber
	HelloMsgTypeSupport::register_type
	participant->create_topic
publisher->SubscriptionAdd	subscriber->create_datareader

Compilable versions of both the *RTI Data Distribution Service* 3.x and 4.0 source code shown above, as well as other examples, are available for download at the RTI website. A list of example, fully-compilable source code projects is available in Appendix A.

6.2 Domain Methods

Recall that a domain is a distributed concept that links all applications that are able to communicate with each other and represents a communication plane. Only the Publishers and the Subscribers attached to the same domain may interact. A Domain is a manager (or factory) of every *RTI Data Distribution Service* object. Domains are global in nature and represent a communication plane.

Table 6.4 lists the *RTI Data Distribution Service* 3.x methods for a Domain (in alphabetical order). Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality:

Table 6.4 **Domain C++ Methods**

3.x Method	Reference to 4.0 Information
Domain->ClientCreate	Section 6.2.8
Domain->ClientDestroy	
Domain->Destroy	Section 6.2.1
Domain->HandleGet	Section 6.2.2
Domain->PublicationCreate	Section 6.2.3
Domain->PublicationDestroy	
Domain->PublicationPropertiesGet	Section 6.2.7
Domain->PublicationPropertiesSet	
Domain->PublisherCreate	Section 6.2.4
Domain->PublisherDestroy	
Domain->ServerCreate	Section 6.2.8
Domain->ServerDestroy	

6.2.1 Methods for Creating/Destroying Domains

NDDSDomainClass::Create — Use the DDSDomainParticipantFactory get_instance and DDSDomainParticipantFactory create_participant methods to instantiate the DomainParticipant.

Destroy — Use the DDSDomainParticipantFactory class delete_participant method.

6.2.2 Domain Method for Retrieving a Domain Handle

HandleGet — Use the DDSDomainParticipantFactory class get_instance method.

6.2.3 Domain Methods for Creating/Destroying Publications

PublicationCreate — As discussed in Section 2.3, a Publication is used by the application to write instances of data for publication. In *RTI Data Distribution Service* 4.0, the equivalent functionality is supported by using both a DataWriter and a Publisher. So an indirect mapping to *RTI Data Distribution Service* 4.0 exists in that a DataWriter will need to be created after the Publisher has been created. So both the DDSDomainParticipant create_publisher and DDSPublisher create_datawriter methods will be required.

PublicationDestroy — As discussed in Section 2.3, an *RTI Data Distribution Service* 3.x Publication is used by the application to write instances of data for publication. In *RTI Data Distribution Service* 4.0, this is supported by using both a DataWriter and a Publisher. If a Publisher is used within the *RTI Data Distribution Service* 3.x application, and only a specific Publication is to be destroyed, then one would use the DDSPublisher delete_datawriter method. If on the other hand, if the *RTI Data Distribution Service* 3.x application does not use a Publisher (only a Publication), then one would need to both delete the *RTI Data Distribution Service* 4.0 Publisher and DataWriter by using the DDSDomainParticipant delete_publisher and DDSPublisher delete_datawriter.

6.2.4 Domain Methods for Creating/Destroying Publishers

PublisherCreate — A Publisher manages a group of Publications. In *RTI Data Distribution Service* 4.0, use the DDSDomainParticipant class create_publisher method to create a Publisher, then create a DataWriter to allow the application to 'publish' data.

PublisherDestroy —In *RTI Data Distribution Service* 3.x this function destroys a publisher. It is important to note that Publications within the Publisher are NOT destroyed so they will have to be destroyed separately. The actual deallocation of memory for the publisher may not occur immediately to ensure safety among the different tasks. After calling this function the Publisher is invalid and should not be used. In *RTI Data Distribution Service* 4.0, use the DDSDomainParticipant class delete_publisher method after all associated DataWriter entities are destroyed via

the DDSPublisher class delete_datawriter method. Note that in *RTI Data Distribution Service* 4.0, DataWriters cannot exist without a Publisher, unlike *RTI Data Distribution Service* 3.x where a Publication can exist with or without a Publisher.

6.2.5 Domain Methods for Creating/Destroying Subscriptions

SubscriptionCreate — As discussed in Section 2.4, a Subscription is supported by using both a DataReader and a Subscriber. So an indirect mapping to *RTI Data Distribution Service* 4.0 exists in that a DataReader will need to be created after the Subscriber has been created. So both the DDSDomainParticipant create_subscriber and DDSSubscriber create_datareader methods will be required.

SubscriptionDestroy — As discussed in Section 2.4, an *RTI Data Distribution Service* 3.x Subscription can stand alone and refers to exactly one Topic that identifies the data to be read. *RTI Data Distribution Service* 3.x also allows you to manage a group of Subscriptions with a Subscriber. If a Subscriber is used within the *RTI Data Distribution Service* 3.x application, and only the Subscription is to be destroyed, then use the DDSSubscriber delete_datareader method. If on the other hand, that the *RTI Data Distribution Service* 3.x application was not using a Subscriber (only a Subscription), then you need to delete both the *RTI Data Distribution Service* 4.0 Subscriber and DataReader by using the DDSDomainParticipant delete_subscriber and DDSSubscriber delete_datareader.

SubscriptionReliableCreate — There is no direct mapping for this method. You can use the DDSDomainParticipant create_subscriber and DDSSubscriber create_datareader methods and then set the RELIABLE QoS to support reliable communications.

6.2.6 Domain Methods for Creating/Destroying Subscribers

SubscriberCreate — A Subscriber manages a group of Subscriptions. In *RTI Data Distribution Service* 4.0, use the DDSDomainParticipant class create_subscriber method to create a Subscriber, then create a DataReader to allow the application to 'subscribe' to data of interest.

SubscriberDestroy —This function destroys a Subscriber. Subscriptions that have been added with the NDDSSubscriberSubscriptionAdd routine are not destroyed, but Subscriptions automatically created through Pattern Subscription are destroyed. In *RTI Data Distribution Service* 4.0, use the DDSDomainParticipant class delete_subscriber method after all associated DataReader entities are destroyed

via the DDSSubscriber class delete_datareader method. In *RTI Data Distribution Service* 4.0, DataReaders cannot exist without a Subscriber, unlike *RTI Data Distribution Service* 3.x where a Subscription can exist with or without a Subscriber.

6.2.7 Domain Methods for Getting/Setting Properties

PublicationPropertiesGet — This routine retrieves the publication's property structure. The properties associated with a Publication in *RTI Data Distribution Service* 3.x do not map directly to DataWriter properties in *RTI Data Distribution Service* 4.0, but object QoS properties can be retrieved by using the DDSPublisher class get_default_datawriter_qos method.

PublicationPropertiesSet —The properties associated with a Publication in *RTI Data Distribution Service* 3.x do not map directly to DataWriter properties in *RTI Data Distribution Service* 4.0, but object QoS policies can be established for a given DataWriter by using the DDSPublisher class set_default_datawriter_qos method.

SubscriptionPropertiesGet — This routine retrieves the Subscription's property structure. The properties associated with a Subscription in *RTI Data Distribution Service* 3.x do not map directly to DataReader properties in *RTI Data Distribution Service* 4.0, but object QoS properties can be retrieved by using the extended QoS DDSSubscriber class get_default_datareader_qos method. One can also gain access to the DataReader's QoS policies by using the DDSDomainParticipant get_default_datareader_qosX method.

SubscriptionPropertiesSet — The properties associated with a Subscription in *RTI Data Distribution Service* 3.x do not map directly to DataReader properties in *RTI Data Distribution Service* 4.0, but object QoS properties can be established for a given DataReader by using the DDSSubscriber class set_default_datareader_qos method.

6.2.8 Domain Methods for Clients and Servers

None of the client or server methods in the Domain class are supported in *RTI Data Distribution Service* 4.0.

6.3 Publication Methods

This section discusses the *RTI Data Distribution Service* 3.x publication methods and how they map to the *RTI Data Distribution Service* 4.0 functionality. As indicated earlier, some of the methods will map directly, others will require redesign of the application. Table 6.5 lists the Publication class methods (in alphabetical order).

Table 6.5 **Publication C++ Methods**

3.x Method	Reference to 4.0 Information
Publication->ListenerSet	Section 6.3.1
Publication->ListenerGet	
Publication->NddsInstanceGet	Section 6.3.2
Publication->NddsTopicGet	
Publication->PropertiesGet	Section 6.3.3
Publication->PropertiesSet	
Publication->ReliableStatusGet	Section 6.3.4
Publication->Send	Section 6.3.5
Publication->SubscriptionWait	
Publication->Wait	

6.3.1 Publication Listener Methods

ListenerGet — Retrieves the Publication's Listener. In *RTI Data Distribution Service* 4.0, a DataWriter's Listener can be obtained by using the DDSDataWriter class get_listener method.

ListenerSet — Allows the Publication's Listener to be modified. In *RTI Data Distribution Service* 4.0, a DataWriter's Listener can be modified by using the DDSDatatWriter class set_listener method.

6.3.2 Publication Methods for Retrieving Instances and Topics

InstanceGet — Obtains a pointer to the Publication instance. In *RTI Data Distribution Service* 4.0, a Publication can use the DDSPublisher class lookup_datawriter method to obtain the DataWriter's instance.

NddsTopicGet — Retrieves the Publication's Topic. The DDSDomainParticipant class provides the find_topic method which gives access to an existing enabled Topic, based on its name. The method expects two arguments: name of the Topic, and a time-out value.

6.3.3 Publication Methods for Getting/Setting Properties

PropertiesGet — The properties associated with a Publication in *RTI Data Distribution Service* 3.x do not map directly to DataWriter properties in *RTI Data Distribution Service* 4.0, but QoS policies can be retrieved for a given DataWriter by using the DDSPublisher class get_default_datawriter_qos method.

PropertiesSet — The properties associated with a Publication in *RTI Data Distribution Service* 3.x do not map directly to DataWriter properties in *RTI Data Distribution Service* 4.0, but QoS policies can be established for a given DataWriter by using the DDSPublisher class set_default_datawriter_qos method.

6.3.4 Publication Method for Getting Status

ReliableStatusGet — When a Publication is publishing to at least one reliable Subscription, ReliableStatusGet provides detailed information pertaining to the reliable status of the Publication. The following is a list of the information returned by the ReliableStatusGet method:

- **event** – Provides the latest event on the publication's reliable stream where the events are defined as:
  - **NDDS_BEFORERTN_VETOED** — The sendBeforeRtn vetoed the Publication. Data was not serialized. This information is not available in *RTI Data Distribution Service* 4.0.
  - **NDDS_QUEUE_EMPTY** — The send queue is empty. This information is not available in *RTI Data Distribution Service* 4.0.

- **NDDS_LOW_WATER_MARK** — The send queue level fell to the low water mark. If the low water mark is 0, only NDDS_QUEUE_EMPTY will be called when the queue becomes empty. This information is not available in *RTI Data Distribution Service* 4.0.
- **NDDS_HIGH_WATER_MARK** — The send queue level rose to the high water mark. If the high water mark is the same as the send queue size, only NDDS_QUEUE_FULL will be called when the queue becomes full. This information is not available in *RTI Data Distribution Service* 4.0.
- **NDDS_QUEUE_FULL** — The send queue is full. This information is not available in *RTI Data Distribution Service* 4.0.
- **NDDS_SUBSCRIPTION_NEW** — A new reliable subscription has appeared. The Built-in Topics feature in *RTI Data Distribution Service* 4.0 can provide this information.
- **NDDS_SUBSCRIPTION_DELETE** — A reliable Subscription disappeared. Note that the Publication only detects the disappearance of a reliable Subscription after the expirationTime of the last refreshed subscription declaration expires and the Publication checks its database. The Built-in Topics feature in *RTI Data Distribution Service* 4.0 can provide this information.

❑ **nddsTopic** — NDDSTopic of the Publication. This information is not available in *RTI Data Distribution Service* 4.0.

❑ **subscriptionReliable** — Number of reliable Subscriptions subscribed to this publication. This information is not available in *RTI Data Distribution Service* 4.0.

❑ **subscriptionUnreliable** — Number of unreliable Subscriptions subscribed to this Publication. This information is not available in *RTI Data Distribution Service* 4.0.

❑ **unacknowledgedIssues** — Number of unacknowledged issues. This information is not available in *RTI Data Distribution Service* 4.0.

6.3.5 Publication Methods for Sending/Waiting

Send — Sends or writes the publication issue. In *RTI Data Distribution Service* 4.0, the equivalent functionality would be to use the DDSDataWriter class write method.

SubscriptionWait — Waits for the existence of Subscriptions. There is no *RTI Data Distribution Service* 4.0 DataWriter or Publisher method that will perform this specific functionality.

Wait — Waits for send queue level to reach the same or lower level specified within the Wait method. There is no *RTI Data Distribution Service* 4.0 DataWriter or Publisher method that will perform this specific functionality.

6.4 Publisher Methods

This section discusses the *RTI Data Distribution Service* 3.x publisher methods and how they map to the *RTI Data Distribution Service* 4.0 functionality. Some of the methods will map directly, others will require redesign of the application. Table 6.6 lists the Publisher class methods (in alphabetical order). Next we will look at how they map to *RTI Data Distribution Service* 4.0 functionality.

Table 6.6 **Publisher C++ Methods**

3.x Method	Reference to 4.0 Information
Publisher->Iterate	Section 6.4.4
Publisher->PublicationAdd	Section 6.4.1
Publisher->PublicationFind	Section 6.4.2
Publisher->PublicationRemove	Section 6.4.1
Publisher->Send	Section 6.4.3
Publisher->SubscriptionWait	

Note: The Publisher object in *RTI Data Distribution Service* 3.x also supported Signaled and Asynchronous modes of operation. Neither of these is supported within the *RTI Data Distribution Service* 4.0 product at this time.

6.4.1 Publisher Methods for Adding/Removing Publications

PublicationAdd — Adds a Publication to a Publisher. In *RTI Data Distribution Service* 4.0, since a DataWriter cannot be instantiated independent of a Publisher, the equivalent method would be to use the DDSPublisher class create_datawriter method. This not only creates the DataWriter object, but adds it to the Publisher.

PublicationRemove — Removes a Publication from being managed by a Publisher. In *RTI Data Distribution Service* 3.x, the notion of being able to remove a Publication from being managed by a Publisher was supported. In *RTI Data Distribution Ser-*

vice 4.0, since a DataWriter must be associated with a Publisher, this functionality is not supported unless multiple DataWriters exist within the Publisher. In this case, the DDSPublisher delete_datawriter method would be used.

6.4.2 Publisher Methods for Finding Publications

PublicationFind — Finds the Publication, of a supplied Topic string, that is managed by the Publisher. In *RTI Data Distribution Service* 4.0, the DDSPublisher class supports the lookup_datawriter method. This method allows the application to supply a Topic string so that the associated DataWriter handle can be retrieved.

6.4.3 Publisher Methods for Sending/Waiting

Send — Takes a snapshot of all the Publications managed by the Publisher and then sends the issues at once, coalescing individual Publications into a single message to maximum network bandwidth utilization. In *RTI Data Distribution Service* 4.0, the NDDSPublisher class does not provide equivalent functionality. Each DataWriter's write method must be used to actually send the data.

SubscriptionWait — Forces the calling thread to wait for at least the number of Subscriptions to appear for each Publication managed by the Publisher. There is no *RTI Data Distribution Service* 4.0 DataWriter or Publisher method that will perform this specific functionality.

6.4.4 Publisher Methods for Iterating

Iterate — Iterates over all managed Publications. This functionality is supported within the DDS Specification, but will not be implemented within the *RTI Data Distribution Service* 4.0 release.

OnMatch — This is a method of the NDDSPublisherIter class. It is called for each publication matching the NDDSTopic and NDDSType specified in the NDDSPublisherClass:Iterate(). *This functionality is supported in the DDS Specification, but will not be supported in RTI Data Distribution Service 4.0.*

6.5 Subscription Methods

This section examines the *RTI Data Distribution Service* 3.x Subscription methods and how they map to *RTI Data Distribution Service* 4.0 functionality. Some of the methods will map directly, others will require redesign of the application. Table 6.7 lists the *RTI Data Distribution Service* 3.x NDDSSubscription class methods (in alphabetical order).

Table 6.7 **Subscription C++ Methods**

3.x Method	Reference to 4.0 Information
InstanceGet	Section 6.5.1
IsReliable	Section 6.5.2
IssueListenerGet	Section 6.5.3
IssueListenerSet	
NddsTopicGet	Section 6.5.1
Poll	Section 6.5.4
PropertiesGet	Section 6.5.1
PropertiesSet	
PublicationWait	Section 6.5.4

6.5.1 Subscription Method for Getting Instances and Topics

InstanceGet — Obtains a pointer to a Subscription's instance. In *RTI Data Distribution Service* 4.0, use the DDSSubscriber class lookup_datareader method to retrieve the instance of a DataReader attached to a Topic.

NddsTopicGet — Returns the Topic subscribed to. In *RTI Data Distribution Service* 4.0, use the DDSDomainParticipant's find_topic method to access an existing enabled Topic, based on its name. The method expects two arguments: name of the Topic, and a time-out value.

6.5.2 Subscription Method for Checking Reliability

IsReliable — Sees if a given Subscription is reliable or best-effort. In *RTI Data Distribution Service* 4.0, use the DDSDataReader's get_qos method to ascertain the QoS reliability policies.

6.5.3 Subscription Methods for Getting/Setting Listeners

IssueListenerGet — Retrieves the Subscription's Listener. In *RTI Data Distribution Service* 4.0, use the DDSDataReader's get_listener method to retrieve the DataReader's listener.

IssueListenerSet — Modifies the Subscription's Listener. In *RTI Data Distribution Service* 4.0, use the DDSDataReader's set_listener method to modify the DataReader's listener.

6.5.4 Subscription Methods for Polling and Waiting

Poll — Polls the Subscription for newly received issues since the last poll. In *RTI Data Distribution Service* 4.0, you receive data by using Listeners. Listeners provide asynchronous notification of data-sample arrival. (In a future version, you will be able to use Condition and Wait-sets, which will provide a way for an application to block until specific conditions are satisfied.)

PublicationWait — Actively probe for a give number of Publications for this Subscription. This functionality is not a supported DDS feature.

6.5.5 Subscription Methods for Getting/Setting Properties

PropertiesGet — Retrieves the current properties of a Subscription. The properties associated with a Subscription in *RTI Data Distribution Service* 3.x do not map directly to DataReader properties in *RTI Data Distribution Service* 4.0, but object QoS policies can be retrieved for a given DataReader by using the DDSDataReader class get_qos method.

PropertiesSet — Modifies the current Subscription properties. The properties associated with a Subscription in *RTI Data Distribution Service* 3.x do not map directly to DataReader properties in *RTI Data Distribution Service* 4.0, but object properties can be established for a given DataReader by using the DDSDataReader class set_qos method.

6.5.6 Subscription Methods for Getting Status

RecvStatusGet — Allows you to enquire to the subscriptions current status. Returns the following:

- **localTimeWhenReceived** — Local time when the issue was received. This information is not available in *RTI Data Distribution Service* 4.0.
- **nddsTopic** — Topic of the Subscription receiving the issue. In *RTI Data Distribution Service* 4.0, the DDSDataReader class get_topicdescription method can be used to determine the DataReader's topic.
- **nddsType** — Type of the Subscription receiving the issue. This information is not available in *RTI Data Distribution Service* 4.0.
- **publicationId** — Publication's unique ID. This information is not currently available in *RTI Data Distribution Service* 4.0.
- **publSeqNumber** — Sending (Publication) high and low sequence number. This information is not currently available in *RTI Data Distribution Service* 4.0.
- **recvSeqNumber** — Receiving sequence high and low sequence number. This information is not currently available in *RTI Data Distribution Service* 4.0.
- **remoteTimeWhenPublished** — Remote time when the issue was published. Once the read or take routine is used within the DataReader to gain access to the received data issue, the SampleInfo class source_timestamp method can be employed which provides the time-stamp provided by the DataWriter at the time the sample was produced.
- **senderAppId** — Sender's application ID. This information is not available in *RTI Data Distribution Service* 4.0.
- **senderHostId** — Sender's host ID. This information is not available in *RTI Data Distribution Service* 4.0.
- **senderNodeIP** — Sender's IP address. This information is not available in *RTI Data Distribution Service* 4.0.
- **validRemoteTimeWhenPublished** — Whether or not a valid remote time was received. This information is not available in *RTI Data Distribution Service* 4.0.
- **status** — Status affects which fields are valid and returns:
  - **NDDS_DESERIALIZATION_ERROR** — Deserialization method for the NDDSType returned an error. This information is not currently available in *RTI Data Distribution Service* 4.0.
  - **NDDS_FRESH_DATA** — A new issue received. In *RTI Data Distribution Service* 4.0, the application can determine the status of the received issue by inspecting the information provided by SampleInfo. SampleInfo information is provided along with each data issue and provides detailed information pertaining to that data instance. You can determine the state of the arriving issue by taking advantage of the information provided by the

sample_state (READ or NOT_READ), view_state (NEW or NOT_NEW), and instance_state (ALIVE, NOT_ALIVE_DISPOSED, or NOT_ALIVE_NO_WRITERS).

- **NDDS_NEVER_RECEIVED_DATA** — Never received an issue, but a deadline occurred. In *RTI Data Distribution Service* 4.0, use the DataReader Listener on_requested_deadline_missed routine. *RTI Data Distribution Service* will invoke this operation when the deadline has been missed. The application can also consult the entities status by directly using the DDSDataReader class get_requested_deadline_missed_status method.
- **NDDS_NO_NEW_DATA** — Received at least one issue, and a deadline has occurred since the last issue was received. Use a combination of the features described above.
- **NDDS_UPDATE_OF_OLD_DATA** — Received a new issue, whose time stamp is the same or older than the time stamp of the last fresh issue received. This information is not available in *RTI Data Distribution Service* 4.0.

The status of any DomainParticipant entity can be obtained via the StatusCondition class. Further status can be ascertained by using the DataReader's get_liveliness_changed_status, get_requested_deadline_missed_status, get_requested_incomptibile_qos_status, and get_sample_rejected_status methods.

6.6 Subscriber Methods

This section examines the *RTI Data Distribution Service* 3.x Subscriber methods and how they map to *RTI Data Distribution Service* 4.0 functionality. Some of the methods will map directly, others will require redesign of the application. Table 6.8 lists the NDDSSubscriber class methods (in alphabetical order).

Table 6.8 **Subscriber C++ Methods**

3.x Method	Reference to 4.0 Information
Iterate	Section 6.6.4
PatternAdd	Section 6.6.5
PatternRemove	
Poll	Section 6.6.3

Table 6.8 **Subscriber C++ Methods**

3.x Method	Reference to 4.0 Information
SubscriptionAdd	Section 6.6.1
SubscriptionFind	Section 6.6.2
SubscriptionRemove	Section 6.6.1

6.6.1 Subscriber Methods for Adding/Removing Subscriptions

SubscriptionAdd — Adds a Subscription to the Subscriber. In *RTI Data Distribution Service* 4.0, since a DataReader cannot be instantiated independent of a Subscriber, the equivalent method would be to use the DDSSubscriber class create_datareader method. This not only creates the DataReader object, but automatically adds it to the Subscriber.

SubscriptionRemove — removes a Subscription from being managed by a Subscriber. In *RTI Data Distribution Service* 3.x, the notion of being able to remove a Subscription from being managed by a Subscriber was supported. In *RTI Data Distribution Service* 4.0, you can remove a DataReader by using the Subscriber's 'delete_datareader' method.

6.6.2 Subscriber Methods for Finding Subscriptions

SubscriptionFind — Finds the Subscription, of a supplied Topic string, that is managed by the Subscriber. In *RTI Data Distribution Service* 4.0, the DDSSubscriber class provides a lookup_datareader method, which retrieves a previously created DataReader belonging to the Subscriber that is attached to a Topic with a matching Topic_name.

6.6.3 Subscriber Method for Polling Subscriptions

Poll — Polls all Subscriptions. There is no equivalent mechanism in *RTI Data Distribution Service* 4.0. In a future version, similar functionality will be available by using DDS objects known as Conditions and Wait-sets.

6.6.4 Subscriber Method for Iterating

Iterate — Iterates over all aggregated Subscriptions. This functionality is supported within the DDS Specification, but will not be implemented within the *RTI Data Distribution Service* 4.0 release.

6.6.5 Subscriber Methods for Adding/Removing Patterns

The concept of patterns does not exist in *RTI Data Distribution Service* 4.0, so there is no equivalent functionality for these methods:

PatternAdd — Adds a pattern Subscription to the Subscriber.

PatternRemove — Removes a previously added pattern, but does not delete the pattern listener passed in during the pattern registration.

6.7 Client and Server Methods

Client/Server functionality is not supported in *RTI Data Distribution Service* 4.0. There is no equivalent functionality for the *RTI Data Distribution Service* 3.x Client and Server methods.

6.8 Listeners

Listeners provide a mechanism for *RTI Data Distribution Service* to asynchronously alert the application of the occurrence of relevant asynchronous events, such as arrival of data corresponding to a Subscription. Listeners are interfaces that the application implements. Each dedicated listener class presents a list of pure virtual methods that correspond to relevant events that the application may wish to respond to.

To continue with our API comparison, let's examine *RTI Data Distribution Service* object listeners. Recall that within *RTI Data Distribution Service* 3.x, Listeners could be associated with a Domain, Publication, and Subscription. The DDS specification indicates that all DCPS entities support their own specialized listener, thus *RTI Data Distribution Service* 4.0 will provide Listener support for each Domain entity.

6.8.1 Domain Listeners

The Domain Listener in *RTI Data Distribution Service* 3.x allows the application the ability to be notified upon the appearance and disappearance of Managers, Applications, Publications, Subscriptions, and Servers. We'll discuss the *RTI Data Distribution Service* 3.x Domain Listener virtual methods that are available and how they can potentially be mapped into the *RTI Data Distribution Service* 4.0 functionality, but first we must address the concept of 'built-in Topics' in *RTI Data Distribution Service* 4.0. The DomainParticipant Listener within *RTI Data Distribution Service* 4.0 does not provide the same virtual functions that are provided in the *RTI Data Distribution Service* 3.x Domain Listener. In order for an *RTI Data Distribution Service* 4.0 application to implement similar *RTI Data Distribution Service* 3.x Domain Listener functionality, the 'built-in Topic' must be used. The DDS specification introduces a set of Built-in Topics and corresponding DataReader objects that can be used by the application to monitor and keep track of new DCPS entities as they are discovered. The Built-in Topics can then be accessed as if it was normal application data.

Table 6.9 lists the *RTI Data Distribution Service* 3.x Domain Listeners (in alphabetical order). Next we will look at how at how they map to *RTI Data Distribution Service* 4.0 functionality.

Table 6.9 **Domain Listeners**

3.x Method	Reference to 4.0 Information
OnApplicationRemoteNew	Section 6.8.1.1
OnApplicationRemoteDelete	
OnManagerRemoteNew	Section 6.8.1.4
OnManagerRemoteDelete	
OnPeerRemoteNew	Section 6.8.1.1
OnPublicationRemoteNew	Section 6.8.1.2
OnPublicationRemoteDelete	
OnServerRemoteNew	Section 6.8.1.5
OnServerRemoteDelete	
OnSubscriptionRemoteNew	Section 6.8.1.3
OnSubscriptionRemoteDelete	

6.8.1.1 Domain Listeners for New/Deleted Applications/Peers

OnApplicationRemoteNew — There is no direct mapping of this method to *RTI Data Distribution Service* 4.0. Equivalent functionality can be provided by 'subscribing' to the built-in Topic "DCPSParticipant." The application can gain access to the built-in Subscriber and associated DataReaders by using the DomainParticipant get_builtin_subscriber and get_datareader methods to gain access to the built-in DCPSParticipant Topic and monitor all traffic related to remote DomainParticipant activity. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote participant to participate within the network by using the ignore_participant method.

OnApplicationRemoteDelete — There is no direct mapping of this method to *RTI Data Distribution Service* 4.0. Equivalent functionality can be provided by 'subscribing' to the built-in Topic "DCPSParticipant." The application can gain access to the built-in Subscriber and associated DataReaders by using the DomainParticipant get_builtin_subscriber and get_datareader methods to gain access to the built-in DCPSParticipant Topic and monitor all traffic related to remote DomainParticipant activity.

OnPeerRemoteNew — See OnApplicationRemoteNew above.

6.8.1.2 Domain Listener for New/Deleted Publications

OnPublicationRemoteNew — there is no direct mapping of this method to *RTI Data Distribution Service* 4.0. Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSPublication. The application can gain access to the built-in Subscriber and associated DataReaders by using the DomainParticipant get_builtin_subscriber and get_datareader methods to gain access to the built-in "DCPSPublication" Topic and monitor all traffic related to remote DataWriter activity. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote participant to participate within the network by using the ignore_publication method.

OnPublicationRemoteDelete — there is no direct mapping of this method to *RTI Data Distribution Service* 4.0. Equivalent functionality can be provided by 'subscribing' to the built-in Topic "DCPSPublication." The application can gain access to the built-in Subscriber and associated DataReaders by using the DomainParticipant get_builtin_subscriber and get_datareader methods to gain access to the built-in "DCPSPublication" Topic and monitor all traffic related to remote DataWriter activity.

6.8.1.3 Domain Listener for New/Deleted Subscriptions

OnSubscriptionRemoteNew — there is no direct mapping of this method to *RTI Data Distribution Service* 4.0. Equivalent functionality can be provided by 'subscribing' to the built-in Topic "DCPSSubscription." The application can gain access to the built-in Subscriber and associated DataReaders by using the DomainParticipant get_builtin_subscriber and get_datareader methods to gain access to the built-in "DCPSSubscription" Topic and monitor all traffic related to remote DataWriter activity. This allows the application to make decisions based on the remote Participant's activity such as not allowing the remote participant to participate within the network by using the ignore_subscription method.

OnSubscriptionRemoteDelete — there is no direct mapping of this method to *RTI Data Distribution Service* 4.0. Equivalent functionality can be provided by 'subscribing' to the built-in Topic DCPSSubscription. The application can gain access to the built-in Subscriber and associated DataReaders by using the DomainParticipant get_builtin_subscriber and get_datareader methods to gain access to the built-in "DCPSSubscription" Topic and monitor all traffic related to remote DataWriter activity.

6.8.1.4 Domain Listeners for NDDS Managers

There is no NDDS Manager in *RTI Data Distribution Service* 4.0, so there are no equivalent methods for the *RTI Data Distribution Service* 3.x Manager Listeners (OnManagerRemoteNew and OnManagerRemoteDelete).

6.8.1.5 Domain Listeners for Servers

Client/Server functionality is not supported in *RTI Data Distribution Service* 4.0. There is no equivalent functionality for the *RTI Data Distribution Service* 3.x Server Listener methods (OnServerRemoteNew and OnServerRemoteDelete).

6.8.2 Publication Listeners

The *RTI Data Distribution Service* 3.x Publication Listener provides the ability for the application to tailor its behavior in response to middleware activity associated with individual publication events. The Publication Listener interface provides the following virtual methods—there is no equivalent functionality in *RTI Data Distribution Service* 4.0 for any of these:

OnAfterIssueSent — This method, if implemented by the application, is invoked by *RTI Data Distribution Service* directly after an issue is sent.

OnBeforeIssueSent — This method, if implemented by the application, is invoked by *RTI Data Distribution Service* prior to an issue being sent.

OnReliableStatus — This method, if implemented by the application, is invoked when publication events related to the send queue occur. With this listener method, an application can monitor a reliable Publication's send queue and control the send rate by responding to specific events.

6.8.3 Issue Listeners

The *RTI Data Distribution Service* 3.x Issue Listener provides the ability for the application to tailor its behavior in response to middleware activity associated with individual subscription events. The Issue Listener interface provides the following virtual methods:

IssueTypeMatch — The *RTI Data Distribution Service* 3.x product uses this method to check for type safety when creating a Subscription. This is necessary to ensure that the Issue Listener can handle the NDDSType the Subscription expects. In *RTI Data Distribution Service* 4.0, this functionality is not used within the user's code.

OnIssueReceived — This method is invoked by *RTI Data Distribution Service* at different times depending on the subscription mode. If configured for IMMEDIATE Subscription, this method is invoked as soon as the issue is received. If configured for POLLED Subscription, this method is invoked when the receiving application explicitly polls. If there are more than one issues received since the last poll, this method will be executed multiple times for each issue. In *RTI Data Distribution Service* 4.0, the DataReaderListener class on_data_available method would be used to receive incoming data.

6.8.4 Subscription Reliable Listeners

The *RTI Data Distribution Service* 3.x Subscription Reliable Listener provides the ability for the application to tailor its behavior in response to activity associated with individual reliable subscription events. The Reliable Subscription Listener interface provides the following virtual method:

OnReliableStatus — This method handles reliable events on the Subscription side and provides the following information:

- **event** — the event for a reliable Subscription. The event parameter can be defined as one of the following:

- **NDDS_ISSUES_DROPPED** — one (or more) issues have been missed by the subscription. In *RTI Data Distribution Service* 4.0, the application can employ the Subscriber Listener on_sample_lost method to determine information pertaining to dropped data issues. One can also access the Sample Lost Status (plain communication status type) allowing the application to determine the total cumulative count of all samples lost across all published instances of a specific Topic.
- **NDDS_PUBLICATION_NEW** — the reliable issue is coming from a publication different from the one that sent the previous issue. This is caused by a publication with a higher value of the strength parameter, or by the expiration of the persistence of the current publication. This information is not available in *RTI Data Distribution Service* 4.0.

❑ **issuesDropped** — the number of issues dropped. In *RTI Data Distribution Service* 4.0, the application can use the Subscriber Listener on_sample_lost routine or the DDSDataReader class get_sample_lost_status method to access both the total_count and total_count_change data fields. The total_count provides the cumulative count of all samples lost across all instances of topics subscribed to by this Subscriber. The total_count_change provides the incremental number of samples lost since the last time the Listener was called or the status was read. These routines provided dropped issue counts for the entire Subscriber, not on a DataReader basis.

❑ **nddsTopic** — the subscription's topic. In *RTI Data Distribution Service* 4.0, once the Subscriber Listener on_sample_lost routine is invoked, the information associated with dropped issues is provided only on a Subscriber basis. There is currently no mechanism available to determine which dropped issues are associated with which Topic.

6.8.5 Publisher and Subscriber Listeners

There are no Publisher or Subscriber Listeners in *RTI Data Distribution Service* 3.x.

6.8.6 Client and Server Listeners

Client/Server functionality is not supported in *RTI Data Distribution Service* 4.0. There is no equivalent functionality for the *RTI Data Distribution Service* 3.x Server Listener methods or the Client Listener method.

Chapter 7

RTI Data Distribution Service Product Lifecycle

The *RTI Data Distribution Service* product lifecycle is composed of three phases:

- ❏ Early Access Phase (Section 7.1)
- ❏ Production Phase (Section 7.2)
- ❏ Retirement Phase (Section 7.3)

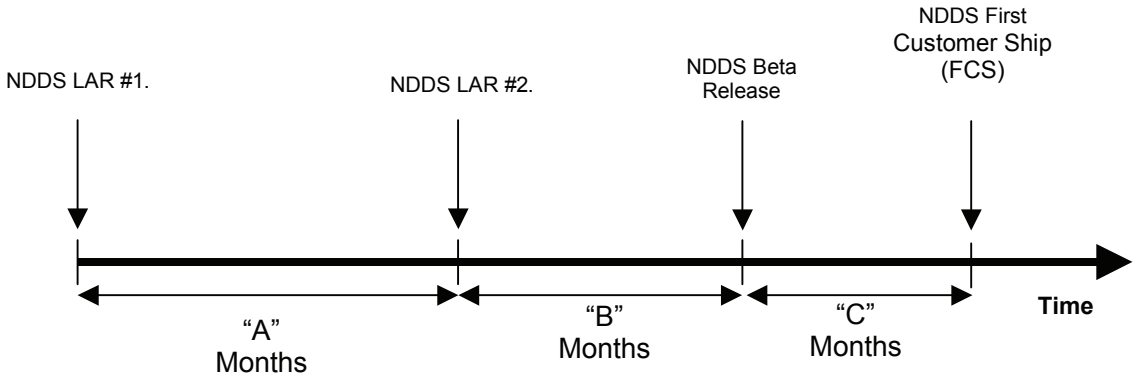
7.1 Early Access Phase

This phase of product development provides key customers access to pre-production product with Limited Access Releases (LARs), as well as a Beta cycle (defined by 'C Months'). The chart below is not to scale and the specific time-frames associated with A, B, and C will vary depending on the specific version of the product in question. Figure 7.1 represents the early access phase of product development.

7.2 Production Phase

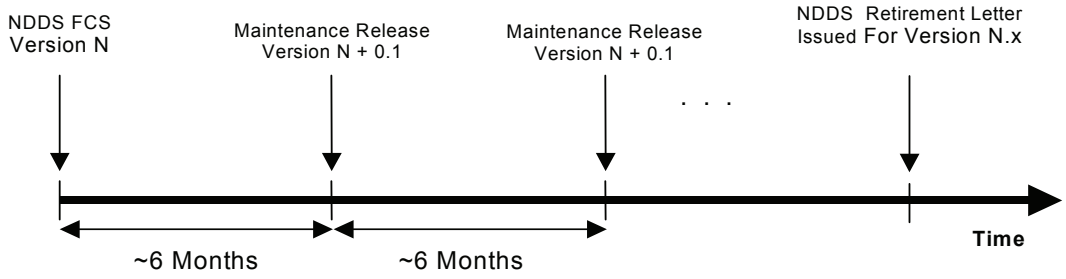
This phase of product development provides customer with access to maintenance releases introduced roughly every 6 months. Maintenance releases include bug fixes and new compiler support for support operating systems, as well as new features when appropriate. At the end of product production, a retirement letter will be sent to each customer articulating the retirement schedule, and maintenance options available to the customer. Figure 7.2 shows the production phase of product development.

Figure 7.1 **Early Access Phase**



"Early Access Phase of Product Development"

Figure 7.2 **Production Phase**

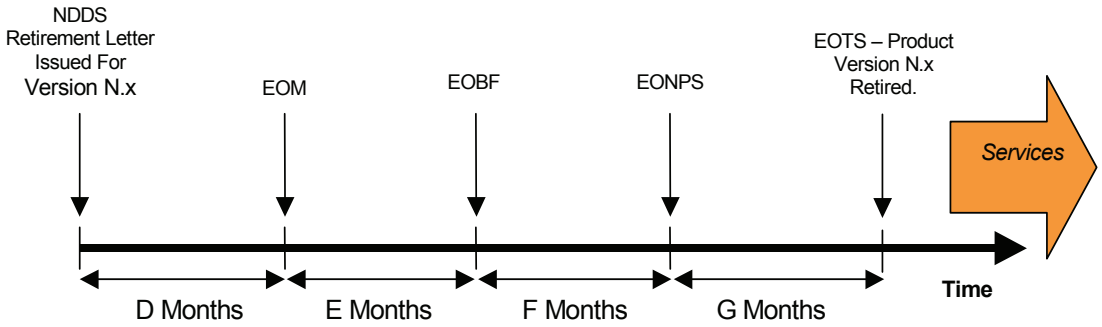


"Production Phase of Product Development"

7.3 Retirement Phase

This phase of product development allows customers the ability to plan migration strategies and factor in how long the product will be supported, and maintained. RTI's standard retirement policy will focus on older releases of *RTI Data Distribution Service* with particular focus on niche architectures, all balanced with current customer usage and market demand. Figure 7.3 represents the retirement phase of product development.

Figure 7.3 **Retirement Phase**



EOM – End of Maintenance: End of maintenance releases and feature enhancements.

EOBF – End of Bug Fixes.

EONPS – End of New Product Sales: End of web-shipments and design win sales.

EOTS – End of Technical Support. Product is retired.

Services – Customer can purchase a service agreement to further maintain the retired release.

Each version of *RTI Data Distribution Service* will have a table similar Table 7.1, which will be available from your local RTI Sales Team.

Each version's retirement table will differ, since each product supports a slightly different OS/compiler configuration. If you have further questions concerning product retirement, or would like further clarification, please contact sales@rti.com for additional details.

Table 7.1 **Sample Retirement Schedule Template**

Version N.x	D	E	F	G
Windows NT				
Windows 2000				
Windows XP				
Solaris 2.7				
Solaris 2.8				
Solaris 2.9				
LynxOs – x86				
LynxOs – PPC				
VxWorks 5.4 – x86				
VxWorks 5.4 – PPC				
VxWorks 5.4 – ARM/StrongArm				
VxWorks 5.5 – x86				
VxWorks 5.5 – PPC				

Chapter 8

Summary

The goal of this document is to provide sufficient information so that an application developer familiar with the 3.x API can assess the level of effort necessary to port an existing application to the *RTI Data Distribution Service* 4.0 API.

To that end, the document reviewed the *RTI Data Distribution Service* 3.x API and mapped the methods to the 4.0 DDS API. In those cases where a direct mapping was not possible, an indirect mapping was presented. In the cases where no direct or indirect mapping was available, viable alternatives were suggested. The document also addressed the extended QoS of the current *RTI Data Distribution Service* 3.x product and how one might transition some of *RTI Data Distribution Service*-specific internal functionality to the DDS-compliant product.

It is important to keep in mind that the document took the vantage point of looking at the current *RTI Data Distribution Service* 3.x APIs and parameters. Therefore, some of the newly introduced DDS functionality was not addressed. To gain a full perspective of the features provided by DDS, download and review the DDS whitepapers available on the RTI website, www.rti.com (see the Resources page). The DDS specification is available at www.omg.org/cgi-bin/doc?ptc/2004-03-07.

The example source code presented in this document for both *RTI Data Distribution Service* 3.x and 4.0 is available from the same URL used to download this document. The examples are provided as compilable source code.

We welcome your comments, questions, and suggestions concerning the document and source code examples. Please email them to support@rti.com.

Appendix A

Buildable Source Code Examples

The projects listed below can be downloaded from the same URL used to download this document. The projects include publish/subscribe examples using the *RTI Data Distribution Service* 3.1a and 4.0 APIs, in C and C++.

Note: The project files for the *RTI Data Distribution Service* 3.1 examples are compatible with Visual Studio .NET 2002 (Visual C++ 7.0). The *RTI Data Distribution Service* 4.0 examples' project files are compatible with Visual C++ 6.0. Makefiles for additional environments can be made available upon request.

Table A.1 **RTI Data Distribution Service 3.1a C and C++ API Examples**

Delivery Mode	Description
Best-effort	Unicast
	Unicast with Publisher/Subscriber
	Multicast
	Multicast with Multicast Meta-traffic
Reliable	Unicast
	Multicast

Table A.2 **RTI Data Distribution Service 4.0 C and C++ API Examples**

Delivery Mode	Description
Best-effort	Unicast
Reliable	Unicast

Index

M

N

O

P

Q

R

S

T

U

W