NDDS®

Network Data Delivery Service

The Real-Time Publish-Subscribe Connectivity Solution

Tutorial

NDDS Version 3.0

Printed in U.S.A. First printing.
September, 2002.

Trademarks

Copy and Use Restrictions

Technical Support

Real-Time Innovations, Inc.
155A Moffett Park Drive
Sunnyvale, CA 94089
Phone: 408-734-4200
Fax: 408-734-5009
Email: support@rti.com
Website: http://www.rti.com

Contents

Figures

Chapter 1

NDDS Overview

NDDS is network middleware for distributed real-time applications

Network Data Delivery Service (*NDDS*®) is network middleware for distributed real-time applications. *NDDS* simplifies application development, deployment and maintenance and provides fast, deterministic distribution of time-critical data over standard networks.

The *NDDS* architecture is based on the publish-subscribe model for data distribution. Real-Time Innovations® (RTI) has added formal extensions to make this model applicable to distributed real-time applications. The *NDDS* real-time, publish-subscribe (RTPS) middleware enables users to:

- ❑ Perform complex one-to-many and many-to-many network communications; the application uses the simple *NDDS* API to publish and subscribe to the data.
- ❑ Customize application operation to meet various real-time, reliability, and quality-of-service goals.
- ❑ Provide application-transparent fault tolerance and application robustness.

1.1 Purpose of This Course

This Tutorial introduces you to *NDDS*'s features and application programming interface (API). The tutorial is based on a series of exercises in which you use the *NDDS* tools, your development tools, and *NDDS* libraries to build *NDDS* publications, subscrip-

tions, clients and servers. At the end of the course, you will have learned the principles of the *NDDS* communications interface.

1.2 Reading and Printing Guide

This Tutorial contains basic and advanced lessons. The basic lessons give you hands-on experience—you will edit code and build applications. In the advanced lessons, you will simply review the supplied example code[1] and read the explanations (you can also build and run the example code).

The Basic Lessons cover these topics:

- ❑ Lesson 1: Use nddsgen to Auto-Create NDDS Types (Section 2.2)
- ❑ Lesson 2: Create a Publication and a Subscription (Section 2.3)
- ❑ Lesson 3: Create a Polled Subscription (Section 2.4)
- ❑ Lesson 4: Create a Publisher and a Subscriber (Section 2.5)
- ❑ Lesson 5: Create a Client and a Server (Section 2.6)

The Advanced Lessons cover these topics:

- ❑ Lesson 6: Publish and Subscribe Reliably (Section 3.1)
- ❑ Lesson 7: Publish and Subscribe Using Multicast (Section 3.2)
- ❑ Lesson 8: Subscribe Using Patterns (Section 3.3)

Use the following guidelines to read and print only the chapters you need:

- ❑ **C++ Users**
  - This NDDS Overview chapter.
  - Basic Exercises (Chapter 2), which provides theory on each basic topic, followed by steps for you to perform.
  - *Optional:* Advanced Exercises (Chapter 3).
- ❑ **C Users**
  - This NDDS Overview chapter.

1. To see the example code, you must install *NDDS*.

- Basic Exercises and Basic C Exercises (Chapter 2 and Chapter 4). Chapter 2 provides theory on each basic topic; Chapter 4 provides the steps for you to follow.
- *Optional:* Advanced Exercises and Advanced C Exercises (Chapter 3 and Chapter 5).

To print a specific chapter from a PDF file using Adobe® Acrobat®:

1. Click the chapter title in the Bookmark list on the left (you may have to select the **Bookmarks** tab). In this example, we'll print Basic Exercises, Chapter 2.
2. Observe the page number at the bottom of the window, such as "23 of 106."
3. Click the chapter title of the *following* chapter (Advanced Exercises) in the Bookmark list.
4. Observe the page number at the bottom of the window, such as "57 of 106." So in the PDF file, the Basic Exercises chapter starts on page 23 and ends on page 56.
5. Select **File, Print**.
6. In the Print Window's Print Range pane, select **Pages from:** and enter 23 in the first box and 56 in the second box.
7. Make any other changes required for your printer, then click **OK**.

1.3 NDDS Documentation Guide

NDDS documentation includes the following documents in PDF format, which is convenient for printing. The following documents are located in the **pdf/** directory of the *NDDS* installation.

- ❑ This *NDDS Tutorial (***NDDSTutorial.pdf**). You must download and install *NDDS*[1] to obtain the example code used in the lessons.
- ❑ The *NDDS Getting Started Guide* (**GettingStarted.pdf**), which includes release notes, installation and compiling instructions. This guide can also be downloaded separately from the distribution.
- ❑ The *NDDS User's Manual (***Manual.pdf**), which contains detailed descriptions on how to use *NDDS* to design, build, and run applications.

1. Download and installation instructions are provided in the *NDDS Getting Started Guide.*

NDDS documentation also includes the following, in HTML format, in the main directory where you installed *NDDS*.

- ❑ The *NDDS* API documentation (**NDDS.html**).

1.4 Publish-Subscribe Architecture

Publish-subscribe is the simplest and most efficient method for one-to-many and many-to-many data distribution

The publish-subscribe communications model provides a more efficient model for broad data distribution over a network than point-to-point, client-server, and distributed-object models. Rather than each node directly addressing other nodes to exchange data, publish-subscribe provides a communications layer that delivers data transparently from the nodes publishing the data to the nodes subscribing to the data. The publish-subscribe model simplifies the application programming effort in the following ways:

- ❑ Replaces socket programming with the simple publish-subscribe API.
- ❑ Decouples publishers from subscribers so each can act independently of each other.
- ❑ Reduces network traffic and promotes system robustness.

Figure 1.1 illustrates the basic publish-subscribe communications model. In this figure, the publish-subscribe communications layer is illustrated as middleware that resides between the application and the operating system's network interface.

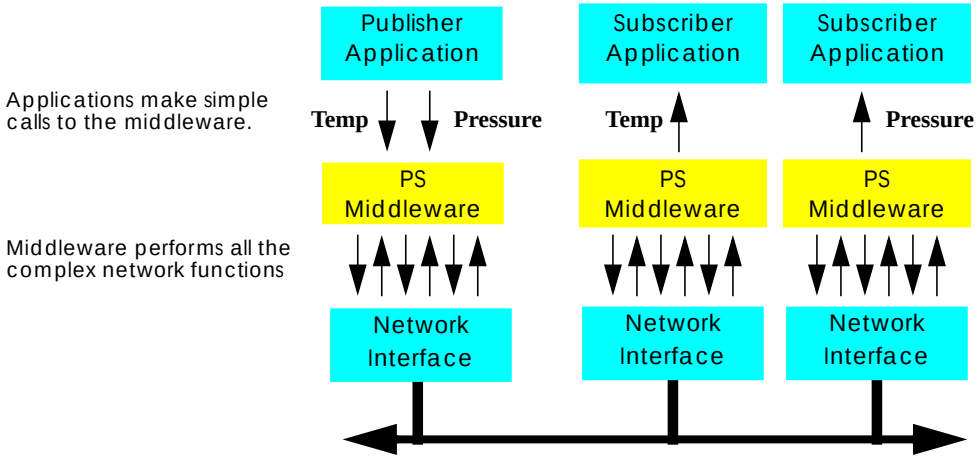
1.4.1 Publish-Subscribe Characteristics

The publish-subscribe (PS) model is defined by the following characteristics:

Distinct publisher and subscriber roles Network communications is composed of applications publishing and subscribing to data. An application can be either a publisher, a subscriber, or both.

One-to-many or many-to-many communications In its simplest form, one application is publishing data and other applications are subscribing to that data. The system is composed of a complex mix of publishers and subscribers deployed on a dynamic set of network nodes. The PS middleware keeps track of what nodes provide (publish) and consume (subscribe to) data, which dramatically reduces application development, debug, and maintenance efforts.

Figure 1.1 **Publish-Subscribe Architecture**



Publish-subscribe middleware presents a simple application programming interface (API) to the publisher and subscriber tasks and takes care of the low-level network communication chores: addressing, packet building, and message transfer. Publishers send the information anonymously; subscribers receive the information anonymously.

Named publications Publishers and subscribers label each publication that will be distributed using a topic (name) rather than the publisher's or subscriber's node addresses.

Declaration and delivery PS communications occurs in three steps:

1. Declaration of intent to publish a topic by publishers.
2. Declaration of interest to receive a topic by subscribers.
3. Propagation of the topic issues (data).

Event-driven transfers Publishers and subscribers operate independently and asynchronously, sending an issue and receiving it are individual events with no interdependencies.

Data marshalling and demarshalling Data is converted and distributed on-the-wire in a form recognizable by all subscribers, regardless of their processor and platform.

1.4.2 Publish-Subscribe in Real Time

Real-time applications are programs that require data transfers within deterministic time constraints. *Distributed* real-time applications interject a network between data publishers and subscribers. Network data-flow can have different characteristics in real-

time applications. For example, many real-time applications have all of the following types of data flow:

- **Signals** Rapidly generated and time-critical data. In most instances, it is more important to get the next issue than to retry a dropped issue.
- **Events** Sporadically generated, time-critical messages which must be delivered reliably.
- **Commands** Sequential instructions which must be received in order.
- **Status** Persistent data about state or goals. Its timeliness differs from one application to the next.
- **Requests** Two-way request-reply transactions for a specific service or data.

Publish-subscribe has several advantages for distributed real-time applications over client-server and distributed-object architectures:

- PS is more efficient in both latency & bandwidth for periodic data distribution.
- PS inherently supports fault tolerance through redundant publishers and subscribers.

Publish-subscribe alone cannot handle the requirements of real-time

The basic publish-subscribe model must be optimized and extended before it is suitable for real-time applications. Modifications are needed to provide the following requirements:

- **Delivery timing control** Real-time subscribers are concerned about when the data is delivered, how long it remains valid, and other issue timing-related factors. For example, a subscriber needs to protect itself from publishers that send data faster than it can handle it. Delivery timing will often differ between flow types; for example, the timing controls must be flexible enough to handle rapid signal and infrequent status data.
- **Reliability control** Reliable delivery conflicts with deterministic delivery; retrying for dropped packets can preclude getting the next issue in a timely manner. Each real-time subscriber must be able to set up its own reliability characteristics for signal, event, command, and status data.
- **Request-reply semantics** Complex real-time applications often have one-time requests for actions or data transactions.
- **Thread priority awareness** Network communications must often be performed without affecting other publisher or subscriber threads.

Safety and robustness add to the requirements

Real-time systems must also function in environments where safety and availability are driving concerns. In addition, the following publish-subscribe extensions are required for these applications:

- ❑ **Fault tolerance** Application-transparent, "hot standby" publishers and / or subscribers are required.
- ❑ **Robustness** The communications layer should not introduce any single-point of failure potential.
- ❑ **Selective degradation** Each publisher-to-subscriber logical data channel must be protected from the others. The performance of one channel should not be affected by others.
- ❑ **Dynamic node detection** Publishers and subscribers should be able to join and leave the system at any time without affecting the operation of the application and middleware configuration parameters.

1.5 RTPS Overview

To use the publish-subscribe communications model for real-time data distribution, developers need a time-aware communications model. *NDDS* is based on a formal Real-Time Publish-Subscribe (RTPS) communications model with the following characteristics:

- ❑ Models time and timestamps every transaction.
- ❑ Allows the application to trade-off timing and reliable delivery.
- ❑ Controls memory usage.
- ❑ Works in the real-time operating system environment.

RTPS defines sets of object parameters and communications characteristics to give developers the control they need to manage all aspects of communication (such as fast delivery, reliability, fault tolerance, and robustness) in a real-time environment.

1.5.1 RTPS Publication Parameters

Real-time extensions let subscribers arbitrate among publications for hot-standby support

Each publication is described by four parameters: *topic*, *type*, *strength* and *persistence*. The *topic* is the label that identifies a specific publication across the network. The *type* defines the publication issue format. Publishers provide *strength* and *persistence* so that subscribers can arbitrate among issues from different publishers.

- **Strength** Specifies the publisher's weight relative to other publishers of the same topic.
- **Persistence** Indicates how long an issue is valid.

Fault-tolerant applications can use redundant publishers to ensure continuous operation. The primary publisher has the highest strength so that issues from the backup publisher are ignored. Should the primary publisher fail, the subscriber receives the issue from the next strongest publisher after the persistence of the primary publisher's issue has expired.

1.5.2 RTPS Subscription Parameters

Real-time extensions let subscribers model their timing constraints

Each subscription is described by four parameters: *topic*, *type*, *minimum separation*, and *deadline*. The *topic* and *type* label and define the publication as described above. Applications use the *minimum separation* and *deadline* parameters to model their timing constraints for each subscription.

- **Minimum separation** Defines a period during which no new issues are accepted.
- **Deadline** Establishes how long the subscriber is willing to wait for the next issue.

Once a subscriber has received an issue, it will not receive another issue for at least the minimum separation. If a new issue does not arrive by the deadline, the application is notified.

1.5.3 Reliable Communications Characteristics

RTPS gives subscribers a method to reconcile real-time versus reliable communications requirements

Reliable delivery means the issues are guaranteed to arrive in the order published. The RTPS reliability model recognizes that the optimal balance between time-determinism and data-delivery reliability varies widely among applications and can vary among different publications within the same application. For example, individual issues of signal data can often be dropped because their value disappears when the next issue is sent. However, each issue of command data must be received and it must be received in the

order sent. RTPS provides a mechanism to customize the determinism / reliability trade-off on a per subscription basis.

The reliability mechanism involves the subscriber and the publisher. A subscriber sets a receive window size in terms of a number of issues. Out-of-sequence issues are stored in this buffer while the middleware retries for the lost issues.

The publisher must be prepared to re-send past issues in response to subscriber requests. RTPS publishers maintain a history queue for each publication tagged as reliable. It re-sends an issue when a subscriber requests it and purges an issue once it is assured that all subscribers have received it.

1.5.4 Request-Reply Service Parameters

RTPS simplifies request-reply semantics by using service names rather than distributed objects

Real-time applications often need to request data residing on a server. Requests differ from publications in that they inherently imply a two-way transaction: the client sends a request to a server; the server sends back a reply. RTPS defines a request-reply mechanism built on top of the named publication model.

RTPS clients and servers model the request-reply slightly differently. Each RTPS server is described by four parameters: *service name*, *request type*, *reply type*, and *strength*. The *name* is a label for the service; the *request* and *reply types* define the data format for the request and reply data, respectively. The *strength* value sets the server's weight relative to other servers supporting the same service name.

RTPS clients define services by *service name, request type, reply type, minimum wait*, and *maximum wait*. The *name* and *type* parameters have the same meaning for clients as servers. The *minimum* and *maximum wait* specify the client's timing constraints. The middleware will not accept a reply received before the minimum wait and will return an error if a reply is not received by the expiration of the maximum wait.

1.6 NDDS: An Implementation of the RTPS Model

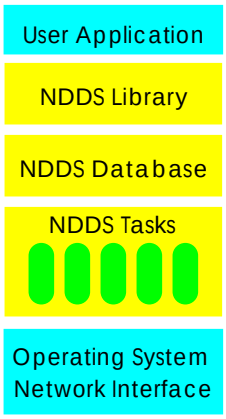
NDDS is network middlewarefor interconnecting real-time, desktop, and workstation platforms

NDDS is network middleware that implements the RTPS communication model. Figure 1.2 illustrates the *NDDS* architecture. *NDDS* is available on number of platforms, including Wind River's VxWorks®, Windows® CE, Windows 2000, Windows NT® 4.0, and UNIX®.

NDDS is implemented on top of UDP / IP (see Figure 1.3) so that real-time applications can use standard IP networks and can coexist with non-real-time (that is, TCP-based) applications on the same network. The *NDDS* managers require little space and perform

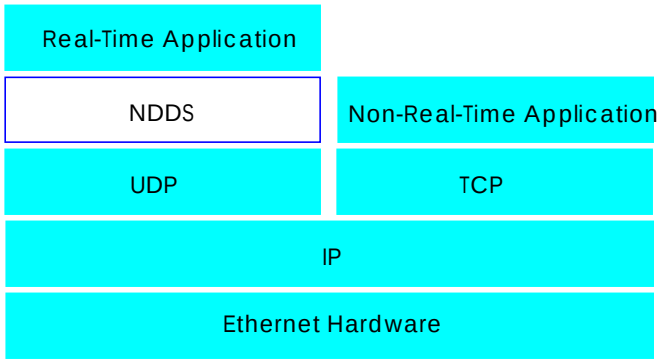
the standard middleware services (manage the publications and subscribers database and serialize, deliver, and deserialize issues). They are also optimized to minimize network communication latency and overhead.

Figure 1.2 **NDDS Architecture**



The NDDS middleware presents developers with a simple API, maintains publications and subscriptions database in each node, and uses the operating system's network stack for all network communications.

Figure 1.3 **Network Stacks**



TCP's reliable communications are fundamentally non-deterministic. NDDS is built on UDP, a packet-oriented protocol that does not affect determinism.

1.6.1 Real-Time Distributed-Application Support

NDDS provides a full publish-subscribe model with real-time extensions

NDDS provides comprehensive publish-subscribe model support. An application can have any combination of publishers, subscribers, clients, servers. The underlying middleware performs all of the messaging services—from fast, best-effort delivery for signal data to fully reliable delivery for events and command data, as well as request-reply transactions for client services.

Applications call *NDDS* through a simple API. The RTPS features are performed by the underlying middleware to simplify the programming effort:

- **Delivery Timing Control** Applications set the minimum separation and deadline when they create the subscription. Once the subscription is created, *NDDS* waits for the minimum separation to expire and either interrupts the application when a new issue arrives or caches it until the application polls.
- **Reliability Control** The application customizes reliable subscriptions by setting a buffer size and deadline according to the reliability requirements on a per subscription basis. *NDDS* uses the buffer to cache out-of-sequence issues while it retries for dropped issues. The application receives the issues in the order sent or is notified that *NDDS* could not retrieve an issue before the deadline.
- **Request-reply Semantics** The application specifies a name, minimum and maximum wait when it creates the client. The server specifies a name and strength. When a client requests a service, *NDDS* delivers the request to all servers with that name, waits the minimum period, and then returns the response from the server with the highest strength received during that period. After the minimum wait expires, *NDDS* returns the first response.
- **Thread Priority Awareness** The *NDDS* publisher lets the application send an issue within the application thread context for minimum latency. For application threads that cannot tolerate network anomalies, *NDDS* provides separate, lower priority threads to handle the network interface.
- **Fault Tolerance** The application sets strength and persistence on a per publication basis. In subscribers, *NDDS* uses the strength and persistence parameters to arbitrate among issues with the same topic. The subscriber application always gets just the issue from the publication with the highest strength whose persistence has not expired. Servers also have a strength value so that *NDDS* can arbitrate among servers with the same name.
- **Robustness** *NDDS* has no central or special nodes or servers. The failure of a publisher, subscriber, client or server will be noticed by the *NDDS* tasks but will not cause their failure.

- **Selective Degradation** *NDDS* publishers and subscribers are decoupled from each other. Each issue is sent to subscribers separately. The failure of one subscriber has no affect on distribution to other subscribers.

1.6.2 Enhanced Publish-Subscribe Capabilities

NDDS has other features that help reduce RTPS application development efforts:

- **Automatic Code Generation** *NDDS* provides a utility, **nddsgen**, that generates the issue serialization and deserialization routines. The utility generates the code from CDR-language publication type descriptions. The code must be linked with the application.
- **Group Publishing and Subscribing** Managing large numbers of publications individually can be confusing and code intensive. *NDDS* publishers and subscribers can be used to manage groups of publications and subscriptions. Publishing in groups offers other advantages too. For example, an *NDDS* publisher can be configured to operate in three modes:
  - synchronously: sends the group of issues immediately upon the application's command, using its own thread context for lowest latency,
  - signalled: sends the group of issues immediately upon the application's command but via a separate thread for application thread protection,
  - asynchronously: checks for new issues on a user-defined periodic basis and sends those issues that have changed.

  Subscriptions can be grouped as well under a single subscriber. *NDDS* can interrupt the application when any of the subscriptions arrive or can hold on to them until the application polls for the lot.
- **Pattern Subscriptions** *NDDS* allows subscribers to name publications with wildcard matches. Thus, a system monitor program could subscribe to "Alarm*", and receive all alarms generated by any node. This facility also encourages hierarchical signal management by allowing subscriptions such as "/Vat/*/logs", which would receive all log messages from the "Vat" subsystem.
- **Multicast Support** *NDDS* uses multicasting to efficiently support high-bandwidth communication. Applications can mix unicast and multicast subscriptions to minimize network load and enable high-speed traffic.

- **Dynamic Node Detection** New subscription and publication declarations can be made at any time. All declarations of publications and subscriptions are aged and eventually discarded, so old information does not affect long-term system health.
- **Multiple NIC Support** The need for reliability also extends to the network: in some applications, the system must continue even if the network itself goes down. *NDDS* provides multiple NICs (network interface cards) for redundant networks. Publications are issued through each network simultaneously.
- **Debugging Tool** *NDDS* includes a utility for monitoring publications. **nddsSpy** uses the *NDDS* pattern subscription capability to subscribe to any available publication that matches user-specified topic pattern. For example, **nddsSpy** monitors all publications when the pattern "*" is entered.
- **C and C++ API** The *NDDS* application programming interface is provided in both C functions and C++ object / method forms.
- **Multi-platform Support** *NDDS* is available for execution on Wind River's VxWorks, Microsoft Windows CE, Windows 2000, Windows NT, Sun Solaris™ and Linux™.

This concludes the introduction to the publish-subscribe communications model, the real-time publish-subscribe communications model, and *NDDS*.

Chapter 2

Basic Exercises

The lessons in this chapter provide instructions on how to use the most basic capabilities of *NDDS*:

- ❏ Lesson 1: Use nddsgen to Auto-Create NDDS Types (Section 2.2)
- ❏ Lesson 2: Create a Publication and a Subscription (Section 2.3)
- ❏ Lesson 3: Create a Polled Subscription (Section 2.4)
- ❏ Lesson 4: Create a Publisher and a Subscriber (Section 2.5)
- ❏ Lesson 5: Create a Client and a Server (Section 2.6)

What's your language?

Each lesson consists of tutorial information on the lesson's topic, followed by an exercise to give you hands-on experience. You can work through the exercises in C++ or C. The steps to follow for C++ are included in this chapter. For C, read the tutorial information in this chapter and then refer to Chapter 4 to complete each lesson.

The completed source code for each exercise is in the <**NDDSHOME**>/**examples/tutorial** directory of your *NDDS* installation, which will be referred to as <**NDDSTutorialDir**>. To work through each exercise in C++, use the **<NDDSTutorialDir>/CPP** directory. As a documentation shortcut, we will leave off the **Cpp** or **C** part of the path and just refer to this directory as **<NDDSTutorialDir>** directory.

2.1 Getting Started

Follow the download and installation instructions in Chapter 2 of the *NDDS Getting Started Guide.*

What's your platform?

In addition to the two languages, you also have the option of using UNIX, Windows, or VxWorks as your platform. For each platform, there are a few preliminary steps to get set up. The *NDDS Getting Started Guide* includes a setup checklist for each platform, which you should confirm before starting the lessons:

- ❏ Compiling UNIX Applications (Section 3.1) in the *NDDS Getting Started Guide*
- ❏ Compiling Windows Applications (Section 3.2) in the *NDDS Getting Started Guide*
- ❏ Compiling VxWorks Applications (Section 3.3) in the *NDDS Getting Started Guide*

2.2 Lesson 1: Use nddsgen to Auto-Create NDDS Types

Goal

Learn how to create an *NDDS* type called "HelloMsg" to send short messages to other applications. The resulting source code should be similar to what is in the **<NDDSTutorialDir>/hello/** directory.

The first step in designing an *NDDS* application is to define each publication's *NDDS type*, which defines the format of the data that you want *NDDS* to distribute and manage for you. *NDDS* requires routines for each type to convert your data types to a form that can be transmitted over the network and vice-versa (serialization and deserialization). Once you register these routines, you can spend your time working on your application while *NDDS* does the "dirty" chores.

The **nddsgen** command will automatically create these routines, allowing you to begin developing your application as soon as you specify each *NDDS* type.

2.2.1 Topic and Type Resolution

A publisher application sends issues[1] at its discretion, unaware of any prospective subscription. A subscription subscribes to an *NDDS topic* that it wants without knowing who is publishing it.

1. An issue contains an instantaneous value of the publication data you want to distribute.

Issues are identified by their *NDDS topic*. The scope of the topic extends to all the applications in the *NDDS* domain among the list of peers. Two publications sending issues with the same *NDDS* topic are viewed as different sources for the *same* topic and are indistinguishable to the subscriptions. For two publications to be distinguishable by any subscription, they must have different *NDDS* topics.

Issues must be of a known type. In this lesson you will use **nddsgen** to create an *NDDS* types and the associated serialize / deserialize routines based on a data structure that you define in the *NDDS* exchange language format.

This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the hands-on part of the lesson. If you are using C, refer to Section 4.1 to complete the lesson.

2.2.2 Create an NDDS Type

To create an NDDS type called HelloMsg:

1. Create a directory called **myhello.**
2. In the **myhello** directory, create a file called **Hello.x** that contains:

```
const MAX_MSG_LEN = 128;

/*nddsgen.C++.NDDSType       HelloMsg;*/
/*nddsgen.C++.output.extension cxx;*/
/*nddsgen.C++.IssueListener  HelloMsg;*/

struct HelloMsg {
    string msg<MAX_MSG_LEN>;
};
```

The keyword *"string"* within the **HelloMsg** structure is the type used in the *NDDS* exchange language for NULL-terminated strings. The value in the angle brackets, < >, specifies a maximum size for the message, which is **MAX_MSG_LEN** here.

The first line:

```
/*nddsgen.C++.NDDSType       HelloMsg;*/
```

- Tells **nddsgen** to generate code in C++.
- Declares an *NDDS* type called HelloMsg.
- Tells **nddsgen** to look for a structure called HelloMsg to build the HelloMsg *NDDS* type.

Specify file extension

The second line:

```
/* nddsgen.C++.Extension cxx;*/
```

results in **.cxx** extensions for the generated files.

Specify issue listener

The third line:

```
/*nddsgen.C++.IssueListener HelloMsg;*/
```

causes **nddsgen** to generate an issue listener class derived from **NDDSIssueListenerClass** for the **HelloMsg** *NDDS* type. An issue listener class is necessary to handle the issues received on the subscription side. Lesson 2 walks you through how to use an issue listener.

3. Type at the command prompt:

```
nddsgen Hello.x
```

2.2.3 Review the Generated Files

nddsgen generates all of the serialize/deserialize code for the structure **HelloMsg** in the file **Hello.x** and puts all of the code in the files **Hello.h** and **Hello.cxx**. The **Hello.cxx** file contains code for the methods of the issue listener class and serialize/deserialize and print methods for the HelloMsg class. The **Hello.h** file contains the declaration of the two classes.

To correct mistakes or modify the *NDDS* type, use the **-replace** argument to overwrite the current files.

The following are the relevant lines from **Hello.h**:

```
#define MAX_MSG_LEN 128

class HelloMsg : public NDDSTypeClass {
public:
static class HelloMsg *New();
virtual const char *NddsTypeGet() const;
virtual RTIBool Serialize(struct NDDSCDRStream *nddsds, int option);
virtual RTIBool Deserialize(struct NDDSCDRStream *nddsds);
virtual RTIBool Print(unsigned int indent);
virtual int MaxSize(int size) const;

public:
char *msg;
};
```

HelloMsg is a class that contains a single data field: the character pointer ***msg***. Although **MAX_MSG_LEN** was used to specify the maximum size of the string in **Hello.x** and was defined with a **#define in Hello.h, *msg*** does not have a predetermined size. This gives you flexibility in allocating space for the string.

2.3 Lesson 2: Create a Publication and a Subscription

Goal

Create a publication and subscription pair to send and receive "Hello World!" messages in a best-effort real-time manner. Best-effort real-time mode means that the issues will be delivered as deterministically as the underlying OS and system allow. If an issue is not received by a subscriber before the deadline expires, the subscriber is notified of the failure (but the publisher continues publishing the data). *NDDS* also supports reliable mode, where the issues are guaranteed to be delivered in order.

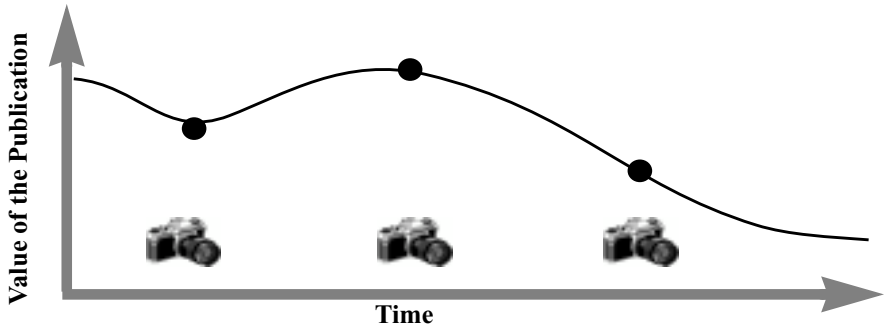
In this exercise, you will create a publication/subscription pair of the **HelloMsg** *NDDS* type created in Lesson 1. The publication will publish a "Hello World!" message, appended by an integer count to identify each message sent. This count will make it obvious to any subscription receiving the message if it has missed an issue, regardless of whether it is a reliable subscription or a best-effort subscription. The publication, however, will not keep a queue to re-send a missing issue; any lost issue is lost forever with this publication. The subscription will receive the "Hello World!" messages from the publication and display it to you, along with its *NDDS* topic and type.

The final code should be the same as that in the **<NDDSTutorialDir>/hello** directory.

2.3.1 Publication Characteristics

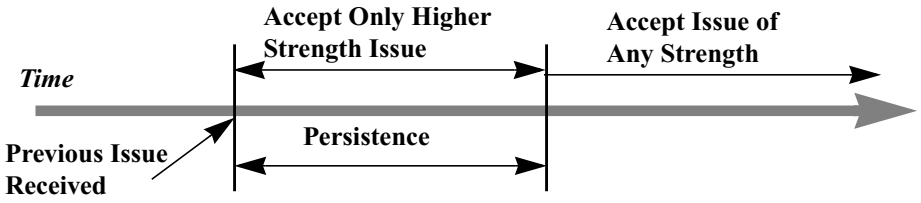
Creating a publication and then sending an issue with the publication is the simplest way of publishing an issue. *Sending an issue* is the process of taking a snapshot of the values of each publication. The value of the publication is combined with the publication's strength, persistence, and time stamp to build the issue. Figure 2.1 illustrates a case where the sending rate is fixed.

Figure 2.1 **Sending Issues at a Fixed Rate**



A publication is characterized by three attributes: *strength, persistence,* and *NDDS topic.* There may be multiple publications with the same *NDDS* topic, usually for redundancy reasons. In this case, *NDDS* uses the strength and persistence to arbitrate among the multiple publications of the same *NDDS* topic. Figure 2.2 illustrates the multiple publication arbitration algorithm.

Figure 2.2 **Multiple Publication Arbitration**



Multiple publications may publish issues with the same NDDSTopic. Subscriptions accept the issue from the strongest active publication whose persistence has not expired.

Typically, a publication that sends issues every period ***T*** will set its persistence to some time ***Tp*** where ***Tp*** > ***T***. While that publication is functional, it will take precedence over any publication of lesser strength. If a publication stops sending issues (willingly or due to a failure), another publication with the same *NDDS* topic will take over after ***Tp***

elapses. This mechanism establishes an inherently fault-tolerant communication channel between the strongest publication of an *issue* and its subscriptions.

2.3.2 Subscription Characteristics

A subscription is the simplest entity that allows an application to receive issues from a publication. A subscription is characterized by five attributes: *subscription mode*, *NDDS topic*, *deadline*, *minimum separation*, and *listener method/callback routine*.[1]

When an *issue* arrives, subscriptions matching the *NDDS topic* are notified through either a subscription *listener method* (in a C++ program) or a *callback routine* (in a C program). In either case, the *issue* is passed as an argument back to the application. The *subscription mode* specifies when this callback routine or listener method is called. The two subscription modes are:

NDDS_SUBSCRIPTION_IMMEDIATE An immediate subscription calls the subscription callback routine or the listener method as soon as a valid issue is received.

NDDS_SUBSCRIPTION_POLLED A polled subscription will not notify the application about the newly received issue until the application "polls." This may add more latency to the data, but is useful in programs that are not thread-safe or cannot handle concurrency.

Deadline and minimum separation control when issues are received

You can specify a desired rate at which issues are received with *minimum separation*. You can also set a *deadline* where the callback is called if an issue is not received within a given time. Of course, a subscription cannot enforce how fast a publication sends issues; a publication could be down or non-existent, after all. However, *NDDS* will notify the receiving application when no new issue has been received within the *deadline* since the last notification, as illustrated in Figure 2.3.

Setting:

$$\text{deadline} = \frac{1}{\text{minimum rate}}$$

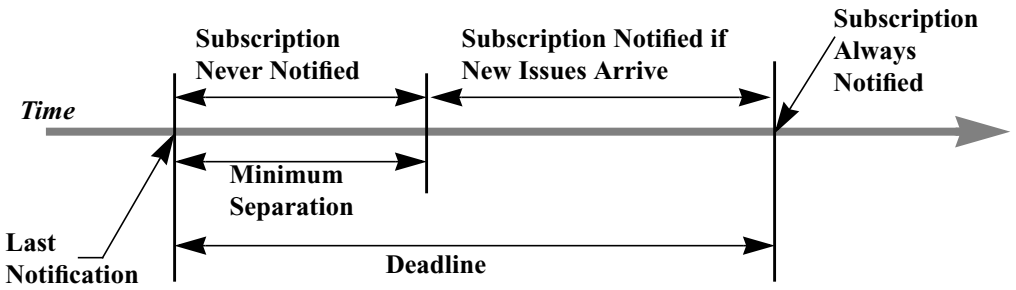
allows the application to detect when the data rate falls below the minimum rate. Similarly, setting:

$$\text{minimum separation} = \frac{1}{\text{maximum rate}}$$

limits the incoming data rate. The minimum separation protects a slow receiver application.

1. Minimum separation is not an attribute of a reliable subscription. For instructions on sending and receiving issues reliably, see Lesson 6.

Figure 2.3 **Customizing Subscription Notification**



The application is notified of issues based on two properties: the minimum separation time and its deadline. Once the application is called with an issue, NDDS will not notify it again until the minimum separation time expires.

If a new issue does not arrive before the deadline, N *seconds, the application is notified of the timeout. This notification occurs between* N *and 2 *** N *seconds.*

This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the hands-on part of the lesson. If you are using C, refer to Section 4.2 to complete the lesson.

2.3.3 Generate Example Publication and Subscription Code

In addition to creating an *NDDS* type, **nddsgen** can also create simple example programs. To help you build these example programs, **nddsgen** creates:

- ❏ A makefile for UNIX platforms.
- ❏ A workspace file (**.dsw**) and project files (**.dsp**) for each program on Windows platforms.

You will need to use **nddsgen** to obtain the example code for the publication and the subscription. Using the same **.x** file created in Lesson 1, use the **-example** flag to tell **nddsgen** to create stub code for simple publication and subscription programs.

In the **myhello/** directory that you created in Lesson 1, type:

```
nddsgen Hello.x -example <architecture>:publish
```

where **<*architecture*>** is the architecture abbreviation for your system, as given in Table 2.1. (For the most recent list of supported architectures, refer to Section 1.1.1 in the *NDDS Getting Started Guide*.)

Table 2.1 **nddsgen Architecture Switch for UNIX and Windows Platforms**

Operating System	CPU	Compiler	RTI Architecture Abbreviation
Linux, 2.4 kernel	Pentium	gcc 2.96	i86Linux2.4gcc2.96
LynxOS 4.0	Pentium	gcc2.95.3	i86Linux4.0gcc2.95.3
Solaris[a] 2.7	SPARC®	Sun CC 5.0	sparcSol2.7cc5.0
		gcc 2.7.2	sparcSol2.7gcc2.7.2
		gcc 2.95	sparcSol2.7gcc2.95
Solaris 2.8	UltraSPARC	Sun CC 5.0	sparcSol2.8cc5.0
		Sun CC 5.2	sparcSol2.8cc5.2
		gcc 2.95	sparcSol2.8gcc2.95
VxWorks 5.4/ Tornado 2.0.x (Solaris or Windows®)	ARM7TDMI®	gcc 2.7.9	arm7tdmiVx5.4gcc
	SA-110™	gcc 2.7.9	armsa110Vx5.4gcc
	Intel® 486	gcc 2.7.2	i486Vx5.4gcc
	Intel Pentium®	gcc 2.7.2	pentiumVx5.4gcc
	PowerPC® 603	gcc 2.7.2	ppc603Vx5.4gcc
	PowerPC 860 (long jump)	gcc 2.7.2	ppc860Vx5.4gcc
	PowerPC 860 (no long jump)	gcc 2.7.2	ppc860Vx5.4gcc.nljmp
	PowerPC EC603	gcc 2.7.2	ppcEC603Vx5.4gcc
	UltraSPARC® (SPARCV9™)	gcc 2.96	sparcv9Vx5.4gcc
	Motorola® 68020	gcc 2.7.2	m68020Vx5.4gcc
VxWorks 5.4.x/ Tornado 2.1.x (Solaris or Windows)	PowerPC 604	gcc 2.96	ppc604Vx5.4gcc

Table 2.1 **nddsgen Architecture Switch for UNIX and Windows Platforms**

Operating System	CPU	Compiler	RTI Architecture Abbreviation
VxWorks 5.5/ Tornado 2.2 (Solaris or Windows)	Pentium®	gcc 2.9	pentiumVx5.5gcc
	Pentium II	gcc 2.9	pentim2Vx5.5gcc
	Pentium III	gcc 2.9	pentium3Vx5.5gcc
	Pentium IV	gcc 2.9	pentium4Vx5.5gcc
	PowerPC 603	gcc 2.96	ppc603Vx5.5gcc
		Diab 5.0.1	ppc603Vx5.5diab
	PowerPC 604	gcc 2.96	ppc604Vx5.5gcc
		Diab 5.0.1	ppc604Vx5.5diab
	PowerPC 860	gcc 2.96	ppc860Vx5.5gcc
		Diab 5.0.1	ppc860Vx5.5diab
Windows NT, Windows 2000, Windows XP	Pentium	Visual C++ 6.0	i86Win32VC60

a. Run **uname -a** to verify your operating system. If your operating system is SunOS, subtract 3 from the version number to obtain the Solaris version number.

Note: Since you did not specify the **-replace** flag, **nddsgen** will not overwrite the existing **Hello.cxx** or **Hello.h** files created in Lesson 1.

The **:publish** switch tells **nddsgen** to generate source code for a publish-subscribe example, as opposed to the client-server example.

You now should have **Hello.h** and three **.cxx** files in the **myhello/** directory: **Hello.cxx** (created in Lesson 1), **Hello_publisher.cxx** and **Hello_subscriber.cxx** (created by using the **-example** flag). In addition, you should either have:

- **makefile_Hello_<*architecture*>**, if you specified a UNIX architecture.
- Or **Hello.dsw**, **Hello_publisher.dsp**, and **Hello_subscriber.dsp**, if you specified a Windows architecture.

2.3.4 Edit Hello_publisher.cxx

To edit the generated publication code to send the "Hello World!" message:

1. Open **myhello/Hello_publisher.cxx** and review the generated code.

Specify the verbosity

NDDS runs at the silent verbosity setting by default; so you see only error messages. Increasing verbosity can reveal what *NDDS* is doing under the hood, and is helpful for advanced debugging. To change the verbosity to 3, initialize **nddsVerbosity** to 3 in **main()**.

RtiNtpTimePackFromNanosec() converts a time value from seconds and nanoseconds into the format used by *NDDS* (a structure of type RTINtpTime).

Initialize an NDDS application

NDDSDomainDerivable() initializes an application in the *NDDS* domain of your choice. This domain creation function must be called before you can create any publications or subscriptions.

Instantiate an object for the NDDSType

HelloMsg() allocates enough memory for **MAX_MSG_LEN** (defined in **Hello.x**) number of characters. The pointer **instance** now points to the **HelloMsg** object you will give to *NDDS* to send an issue.

Create a publication

The following creates a publication for the hello message:

```
if(!(publication = new NDDSPublicationDerivable(domain,
                              "Example HelloMsg", instance, &properties)) ||
   !publication->IsValid()){
    return 0;
}
```

The generated code creates a publication using the values listed in Table 2.2. While there are other fields in the properties structure, for this example we are only interested in the persistence and strength fields.

Table 2.2 **Arguments to NDDSPublicationDerivable()**

Argument	Value	Description
nddsDomain	0	domain ID
ndds*Topic*	"Example HelloMsg"	*NDDS* topic
instance	address of a HelloMsg object	The instance is an object of the **HelloMsg** class
properties.persistence	15 seconds	Refer to Section 2.3.1
properties.strength	1	Refer to Section 2.3.1

Send the message

2. You need to add one line to make the code work. Underneath the line, `/*` modify the data to be sent here `*/`, add the following code, which modifies the message to be sent:

```
sprintf(instance->msg, "Hello Universe! (%d)", count);
```

The ***count*** variable already exists and is incremented by the publication task every time it sends an issue with the call:

```
publication->Send();
```

You can simulate the effect of fixed rate issuing by "sleeping" for **send_period_sec** (which has been set to 4 seconds):

```
NddsUtilitySleep(send_period_sec);
```

Warning: Do not use **NddsUtilitySleep()** to conduct performance tests, such as throughput or latency tests. **NddsUtilitySleep()** cannot guarantee the resolution required in a performance test. Instead, use the **for()** loop or its equivalent, which has much finer resolution. For hints, see the performance test examples in the **examples/performance/** directory.

3. Save your changes. The final code should be the same as that in the **<NDDSTutorialDir>/hello** directory.

2.3.5 Review Hello_subscriber.cxx

To review the subscription code for the "Hello World!" message:

1. Open **myhello/Hello_subscriber.cxx** and review the generated code. No changes are required to run this code.

   The generated code for **Hello_subscriber.cxx** parallels **Hello_publisher.cxx** in three ways:

   a. Sets *NDDS* verbosity with **NddsVerbositySet()**.

   b. Initializes the application in an *NDDS* domain with **NDDSDomainDerivable()**.

   c. Instantiates a **HelloMsg** object.

Review the generated subscription listener class

In Lesson 1, **nddsgen** created a class for the *NDDS* type described in **hello.x**. **nddsgen** also declared an issue listener class for the *NDDS* type. Both classes are in **Hello.h**. An issue listener class specifies what to do with received issues. The **OnIssueReceived()** method in **Hello_subscriber.cxx** is derived from the ***NDDS*IssueListenerClass** and remains a pure virtual method.

In **hello.h**:

```
class HelloMsgListener : public NDDSIssueListenerClass {
  public:
    virtual ~HelloMsgListener() {};
    HelloMsgListener() {};
```

```
     virtual RTIBool IssueTypeMatch(class NDDSTypeClass *instance);
    class HelloMsg *InstanceGet(class NDDSTypeClass *instance)
                         { return (HelloMsg *)instance; }
};
```

Implement your own OnIssueReceived() method (optional)

Since the **OnIssueReceived()** method in the **HelloMsgListener** class is pure virtual, you must implement the **OnIssueReceived()** method by deriving your own listener class from the **HelloMsgListener** class, as the generated subscription code does in **Hello_subscriber.cxx**:

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

RTIBool MyListener::OnIssueReceived(const NDDSRecvInfo * /*info*/,
                                 class NDDSTypeClass *instance)
{
    instance->Print(0);
    return RTI_TRUE;
}
```

Create an immediate subscription

Notice that the generated code in **subscriberMain()** calls **NDDSSubscriptionDerivable()** to create a subscription using the values listed in Table 2.3:

```
new NDDSSubscriptionDerivable(domain, "Example HelloMsg", instance,
                                listener, &properties, NDDS_USE_UNICAST)
```

Sleep while waiting for issues

There is nothing left to do after creating the subscription but wait for issues.

Table 2.3 **Arguments to NDDSSubscriptionDerivable()**

Argument	Value	Description
nddsDomain	0	domain ID
nddsTopic	"Example HelloMsg"	Must be the same as the *NDDSTopic* assigned to the publication.
instance	address of a HelloMsg object	The instance is an object of the **HelloMsg** class.
listener	listener	A pointer to your subscription listener instance
properties.deadline	10.0 seconds	Alert the application if no new data is received in 10 seconds. See Section 2.3.2.

Table 2.3 **Arguments to NDDSSubscriptionDerivable()**

Argument	Value	Description
properties.minimum-Separation	0.0 seconds	The subscription will accept all issues from the publications no matter how fast they are published. See Section 2.3.2.
multicastAddress	NDDS_USE_UNICAST	Receive issues using unicast. To learn about using multicast in *NDDS*, see Lesson 7.

2. *(Optional)* The generated **OnIssueReceived()** method simply prints the content of the issue. Of course, you can do more useful things in your application. To give you a flavor for what you might do for your application, you can print some detailed information about the issue.
3. Save your changes (if any). The final code should be the same as that in the **<NDDSTutorialDir>/hello** directory.

2.3.6 Build the Publication and Subscription Programs

Instructions on building publication and subscription programs are provided in the following sections:

- ❏ UNIX: Section 2.3.6.1
- ❏ Windows: Section 2.3.6.2
- ❏ VxWorks: Section 2.3.6.3

2.3.6.1 UNIX Systems

To build the two applications using the generated makefile:

1. In the **myhello** directory, type:

   ```
   gmake -f makefile_Hello_<architecture>
   ```

   where **<*architecture*>** is the architecture switch you specified to **nddsgen** (see Table 2.1).

   If all goes well, you will see intermediate files for **Hello_publisher** and **Hello_subscriber** in the **objs/<*architecture*>/** directory.

   If the build fails, it is likely that you do not have the compiler and linker specified in the makefile. See your system administrator or contact RTI for advice.

2.3.6.2 Windows Systems

NDDS requires Microsoft Visual C++ 6.0 Service Pack 3 running on either Windows NT 4.0 Service Pack 5 (or higher) or Windows 2000.

To build the publication and the subscription programs:

Set up the workspace

1. Start Microsoft Visual C++ 6.0.
2. From the **File** menu, select **Open Workspace...**

   The **Open Workspace** Window pops up.
3. In the **Open Workspace** Window, browse to the **myhello** directory.
   a. Select **Hello.dsw**.
   b. Click on the **Open** button.

Build a publication program

4. At the bottom of the Workspace Window, click on the **FileView** tab and right-click **Hello_publisher files** to display its pop-up menu. (If you don't see the WorkSpace Window, select **View, Workspace.**)
5. From the pop-up menu, select **Build**.

   This builds a debug version of the program. To build the release version, change the build target in the Build menu (which can be displayed by right-clicking over the menu area).
6. Confirm that you now have **Hello_publisher.exe** in the directory, **myhello\objs\i86Win32VC60**.

Build a subscription program

7. To build the subscription program, repeat Step 4 through Step 6 substituting **Hello_publisher** with **Hello_subscriber**.

Warning: The generated projects already specify the Multi-threaded DLL build option. If you link other objects or libraries to your *NDDS* application, they must also build with the Multi-threaded DLL build option. For instructions on how to specify this option, see Section 3.2.2 of the *NDDS Getting Started Guide*.

2.3.6.3 VxWorks Systems

Write your own makefile to build for your VxWorks target. Your build settings should reflect Table 2.4.

To build the publication and the subscription programs using your own makefile:

1. Build the **Hello_publisher** object from **Hello.cxx** and **Hello_publisher.cxx**.
2. Build the **Hello_subscriber** object from **Hello.cxx** and **Hello_subscriber.cxx**.

Table 2.4 **VxWorks Architecture-Independent Build Settings**

Field	Values
Compiler DEFINES	RTI_VXWORKS
INCLUDES Directories	$(NDDSHOME[a])/include/vx
Libraries to link with	does not apply in VxWorks

a. NDDSHOME is the root directory of the installation

2.3.7 Run the Subscription Program

Start the subscription program before the publication program so that you do not miss any issues.

To run the subscription program:

❑ On UNIX systems, type at a command prompt:

```
objs/<architecture>/Hello_subscriber
```

where **<*architecture*>** is the architecture switch for your system (see Table 2.1).

❑ On Windows systems, type at a command prompt:

```
objs\i86Win32VC60\Hello_subscriber
```

❑ On VxWorks systems:

1. Make sure you have:
   - Set the host name of the target.
   - Specified the IP addresses of all the peers you wish to communicate with.
   - Set the **NDDS_PEER_HOSTS** environment variable with the *putenv* command.
   - Loaded the appropriate libraries.

   Once again, **examples/vxWorks/login.cmd** is a good starting point.
2. Since VxWorks does not have a **main()** entry point, enter the program at the subscription program's main body of code:

```
int subscriberMain(int nddsDomain, int nddsVerbosity)
```

Type at the *windsh*:

```
sp (subscriberMain, 0, 0)
```

2.3.7.1 Subscription Screen Output

If you did not modify the generated subscriber program, you should see:

```
Allocate HelloMsg type.
Sleeping for 10.000000 sec...
Sleeping for 10.000000 sec...
HelloMsg:
msg: ""
```

The subscription sleeps for 10 seconds while waiting for issues. Since you did not start the publication program yet, there is no data to receive. At the 10 second deadline, it prints whatever it received last, which is nothing in this case.

2.3.8 Run the Publication Program

Each node must specify its peer(s) in the **NDDS_PEER_HOSTS** environment variable or file.

To run the publication program:

- ❑ On UNIX systems, type at a command prompt:

  ```
  objs/<architecture>/Hello_publisher
  ```

  where ***<architecture>*** is the architecture switch for your system (see Table 2.1).

- ❑ On Windows systems, type at a command prompt:

  ```
  objs\i86Win32VC60\Hello_publisher
  ```

- ❑ On VxWorks systems, type at the *windsh*:

  ```
  sp (publisherMain, 0, 0)
  ```

2.3.9 Review the Screen Output on Each Side

On the publication screen, you should see:

```
Allocate HelloMsg type.
Sampling publication, count 0
Sampling publication, count 1
```

```
...
```

The subscription will start receiving issues as soon as the publication program starts running. The subscription should display the following:

```
HelloMsg:
    msg: ""
HelloMsg:
    msg: "Hello Universe! (0)"
Sleeping for 10.000000 sec...
HelloMsg:
    msg: "Hello Universe! (1)"
...
```

2.3.10 Experiment with the Programs

Try killing and restarting the publication and the subscription programs. No matter what you do, the subscription gets the newest issue published.

To kill the exercise programs:

- ❑ On UNIX and Windows systems, type at a command prompt:

  <Ctrl-C>[1]

- ❑ On VxWorks systems:

  1. To see the summary of tasks, type at the console or *winsh*:

     ```
     i
     ```

  2. Identify the task ID of the task you want to stop. This is the hexadecimal number in the TID column.
  3. Type (where ***<#tid>*** is the task ID number of the task in hexadecimal):

     ```
     td <#tid>
     ```

2.3.11 Congratulations!

Your first *NDDS* applications are running. You used the immediate subscription mode to receive the issues in an event-driven manner. The following is the sequence from publication to delivery:

1. Press the "c" character while the Ctrl key is pressed.

1. *NDDS* marshals (serializes and deserializes) the data into an exchange language format and sends it directly to the subscription application.
2. *NDDS* at the subscription application demarshals the received issue, puts it into the memory address (**instance**), and calls the callback routine specified at the subscription creation time.

Immediate mode yields the minimum latency possible. In Lesson 3 you will learn how to use the polled mode for subscriptions that cannot have interrupted threads.

2.4 Lesson 3: Create a Polled Subscription

Goal

Modify the subscription created in Lesson 2 to poll for issues from the publication created in Lesson 2.

The immediate subscription mode is simple and yields minimum latency. However, immediate subscription mode may be inappropriate for an application that cannot be interrupted. The immediate subscription mode is also dangerous if the subscription callback routine calls a non-reentrant function(s) because an internal *NDDS* thread executes the callback routine as soon as a new issue is received, regardless of what the other application threads are doing at that moment. If your application is constrained in either of these two ways, the polled mode is the safer alternative.

In this lesson, you will use the polled subscription mode to receive issues.

The final code should be the same as that in the **<NDDSTutorialDir>/polled** directory.

2.4.1 Polled Subscription Characteristics

As the name suggests, if you use the polled subscription mode, you have to actively poll. The role of deadline and minimum separation are the same. But since *NDDS* cannot wake you up at deadline, you will only find out about an expired deadline when you poll. This suggests that you must poll at a period smaller than your *deadline*. In fact, you should poll faster than the average rate of data arrival, which is controlled by the subscription's minimum separation.

Buy insurance with the receive queue

But no matter how fast you poll, there is a possibility of issues arriving in a bursts, resulting in dropped issues. Increasing the receive queue size protects you against such events.

This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the hands-on part of the lesson. If you are using C, refer to Section 4.3 to complete the lesson.

2.4.2 Edit the Subscription Source Code

Continue working in the **myhello** directory you used in the previous lessons.

There are three changes required in the subscription to poll for "Example HelloMsg":

- ❑ Change the subscription mode to **NDDS_SUBSCRIPTION_POLLED**.
- ❑ Increase the receive queue size to 8.
- ❑ Poll the subscription at a 10 second rate.

Note: You will be polling slower than the send rate of the publication. Initially, the receive queue will cache the issues, but you will drop issues eventually. For the purpose of this exercise, this is fine.

To edit the Hello_subscriber.cxx code for polled mode:

1. Open **Hello_subscriber.cxx** for edit.
2. In the section of code commented “set subscription properties” change **NDDS_SUBSCRIPTION_IMMEDIATE** to **NDDS_SUBSCRIPTION_POLLED**.
3. After the line modified in the above step, **properties.mode = NDDS_SUBSCRIPTION_POLLED**, add the following line to increase the receive queue size:

   ```
   properties.receiveQueueSize = 8;
   ```

4. Change the line:

   ```
   /* subscription->Poll() is only needed... */
   ```

   in the while(1) loop to:

   ```
   subscription->Poll();
   ```

Note: You poll the subscription at the *deadline* rate due to:

```
NddsUtilitySleep(properties.deadline);
```

5. Save your changes. The final code should be the same as that in the **<NDDSTutorialDir>/polled** directory.

Note: Publication code is unaffected for the polled mode.

2.4.3 Build the Publication and Subscription Programs

See Section 2.3.6 for instructions on building the publication and subscription programs.

2.4.4 Run the Subscription and Publication Programs

There is no difference in output content from Lesson 2. But if you look closely, the subscription does not report new data right away, but in bursts every 10 seconds, which is the polling rate.

2.5 Lesson 4: Create a Publisher and a Subscriber

Goal

Use a signalled publisher to send the "Hello World!" message to a subscription managed by a subscriber. The final code should be the same as that in **<NDDSTutorialDir>/publisherSubscriber** directory.

2.5.1 Why Use Publishers?

Creating a publication to send issues is fine for simple applications such as those covered in this Tutorial. But a real application may publish hundreds to thousands of publications. If you publish 100 topics, calling the send function for each of the 100 publications is tedious and invites errors.

Send multiple publications with a single call

The solution is to use *publishers* to manage a group of publications that can be sent together. Adding a group of publications to a publisher and then calling the send function of the publisher will send issues for all publications in one shot.

Control how issues are sent

Another benefit of using publishers is more control over how the issues are sent. If you use the publication to send issues, they are sent in the calling task's context. But this can

affect the real-time behavior and is not suitable for time-critical tasks. With publishers, you can choose among the following publisher modes:

Publishing modes: synchronous, asynchronous, and signalled

NDDS_PUBLISHER_SYNCHRONOUS This is the behavior you get if you use publications to send issues. The issue is distributed immediately by the task that calls the send method.

- **Advantage** Minimum communication latency.
- **Disadvantage** The application task takes a non-deterministic amount of time to send issues. This is unsuitable for time-critical tasks.

NDDS_PUBLISHER_SIGNALLED The application task signals another *NDDS* thread to send the issue.

- **Advantage** Almost the same low communications latency as the **NDDS_PUBLISHER_SYNCHRONOUS** mode without slowing down the application task. This is the recommended publisher type for time-critical tasks.
- **Disadvantage** Additional thread created for each signalled publisher.

NDDS_PUBLISHER_ASYNCHRONOUS The issues are saved for later delivery by an *NDDS* thread that executes at a fixed rate.[1] In this mode, multiple issues can be packaged together and the message overhead per issue is smaller.

- **Advantage** The application task takes a deterministic amount of time to send issues suitable for time-critical tasks, without the overhead of an additional thread per signalled publisher.
- **Disadvantage** Increased latency.

Design considerations affect which publisher mode to use. For example, if the OS of the node that will run the publisher program is not multi-threaded, **NDDS_PUBLISHER_SYNCHRONOUS** is the only option. In another example, if you are trying to achieve the maximum bandwidth possible **NDDS_PUBLISHER_ASYNCHRONOUS** might do the job. Regardless of the publisher's mode, there is *no outward difference* to you in the API or usage; you *just send*. *NDDS* takes care of the all underlying chores. Just be aware of which publisher mode you are using.

Note You can still use a publication to send an issue even after adding it to a publisher. But if you use the publication to send the issues, they are sent in the **NDDS_PUBLISHER_SYNCHRONOUS** mode, regardless of the mode of the publisher to which the publication belongs.

1. A mode is called asynchronous when the thread that is actually doing the sending is not synchronized to the thread that called the ***Send*** function.

2.5.2 Why Use Subscribers?

Subscribers can poll in one shot

The subscriber-subscription relationship parallels the publisher-publication relationship. Adding a group of polled subscriptions to a subscriber and polling the subscriber results in all subscriptions being polled together.

Subscribe to pattern topics

Subscribers also allow you to subscribe to pattern topics. You will learn about the pattern subscription feature in Lesson 8.

This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the hands-on part of the lesson. If you are using C, refer to Section 4.4 to complete the lesson.

2.5.3 Edit Hello_publisher.cxx

Continue working in the **myhello** directory you used in the previous lessons.

To edit the Hello_publisher.cxx code to use a signalled publisher:

1. Open **Hello_publisher.cxx** for edit.
2. Add the NDDS**PublisherDerivable** declaration after the NDDS**PublicationDerivable** declaration line:

```
NDDSPublicationDerivable *publication = NULL;
NDDSPublisherDerivable   *publisher = NULL;
```

3. Create a signalled publisher just after calling **new NDDSPublicationDerivable()** by adding the following lines:

```
if(!(publisher = new NDDSPublisherDerivable(domain,
                                            NDDS_PUBLISHER_SIGNALLED)) ||
   !publisher->IsValid()) {
    return 0;
}
```

4. *After* you've created the publication and the publisher, add the publication to the publisher by adding the following lines:

```
if(!(publisher->PublicationAdd(publication))){
    return 0;
}
```

5. Use the publisher to send issues. Change:

```
publication->Send();
```

to:

```
publisher->Send();
```

6. Save your changes. The final code should be the same as that in **<NDDSTutorialDir>/publisherSubscriber** directory.

2.5.4 Edit Hello_subscriber.cxx

Continue working in the **myhello** directory you used in the previous lessons.

To edit the Hello_subscriber.cxx code:

1. Open **Hello_subscriber.cxx** for edit.
2. Add the NDDS**SubscriberClass** declaration after the NDDS**SubscriptionDerivable** declaration line:

```
NDDSSubscriptionDerivable *subscription = NULL;
NDDSSubscriberDerivable   *subscriber = NULL;
```

3. Create a subscriber just after calling **new NDDSSubscriptionDerivable()** by adding the following lines:

```
if(!(subscriber = new NDDSSubscriberDerivable(domain))) {
        return 0;
}
```

4. *After* creating the subscription and the subscriber, add the subscription to the subscriber by adding the following lines:

```
if(!subscriber->SubscriptionAdd(subscription)) {
        return 0;
}
```

5. If the subscription were in the immediate mode, you could stop here. But since this is a polled subscription, you must now use the subscriber to poll. Change:

```
subscription->Poll();
```

to:

```
subscriber->Poll();
```

6. Save your changes. The final code should be the same as that in **<NDDSTutorialDir>/publisherSubscriber** directory.

2.5.5 Build the Publisher and Subscriber Programs

For instructions on how to build the publisher and subscriber programs, see Section 2.3.6.

2.5.6 Run the Subscription and Publication Programs

Using publishers and subscribers does not change the apparent behavior. The outputs should be the same as in Lesson 2.

2.6 Lesson 5: Create a Client and a Server

Goal

Create a client-server pair and implement a service that adds a pair of numbers. In this exercise, you will create a client that sends two numbers to a server, which will add them and return the result to the client.

The final code should be the same as that in the **<NDDSTutorialDir>/clientServer** directory.

2.6.1 Client-Server Transactions

Real-time applications occasionally need to make specific requests for data or an action. Requests differ from publications in that they inherently imply a two-way transaction—the client sends the request along with some parameters, and the server performs an action and returns a response to that specific request.

Clients and servers are identified only by their service name

NDDS implements a client-server transaction mechanism consistent within the publish-subscribe paradigm. Client-server transactions are built around the concept of *services.* A service consists of a *service name* (similar to an *NDDS* topic), and *NDDS* types for the request and reply messages. An application can declare itself as a *server* or a *client* for any given service.

2.6.1.1 Server Characteristics

A server has the following characteristics:.

- **ServiceName** Analogous to the *NDDS* topic, in that it uniquely identifies a service in the *NDDS* domain.
- **Server mode** Parallels the subscription mode discussed in Lesson 2.
  - **NDDS_SERVER_IMMEDIATE** Calls the server the listener method (in the C++ language API) or callback routine (in the C language API) as soon as a valid request is received.
  - **NDDS_SERVER_POLLED** Does not notify the server about the newly received request until the server "polls." This may add more latency to the response, but is useful in programs that must not be interrupted or cannot handle concurrency.
- **server listener (in C++ API)** The **OnServiceRequest()** method of the specified server listener object executes when a service request is received.
- **strength** Enables a client to arbitrate replies from multiple servers with the same service name.

2.6.1.2 Client Characteristics

A client invokes a service by its *service name* and specifies the following parameters:

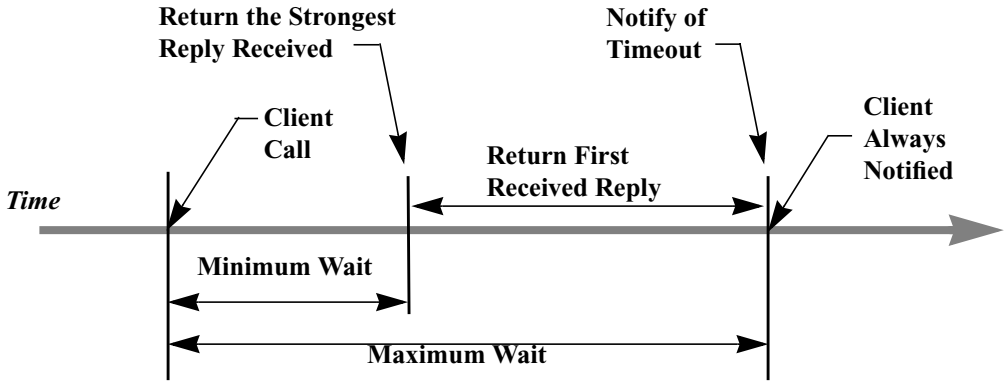
- minimum wait time
- maximum wait time for the server reply
- blocking mode

2.6.1.3 Client-Multi-Server Semantics

NDDS client-server transaction mechanism supports *client-multi-server* semantics. This occurs transparently whenever multiple *servers* are created for the same service name. The request will return with the best—highest strength—response that arrived within the *minimum wait* period. If no reply is received within that period, *NDDS* returns the first response received until the maximum wait time expires. Figure 2.4 illustrates this "best then first" semantics.

Of course, the traditional case of multiple clients for a server also works. Combining the two cases, you can implement a multi-client multi-server scenario.

Figure 2.4 **Best Then First Semantics**



This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the hands-on part of the lesson. If you are using C, refer to Section 4.5 to complete the lesson.

2.6.2 Create the Request and Reply NDDS Types

To create the request and reply NDDS Types for the *add* service:

1. Create a new directory called **myClientServer/**.
2. In the new directory, create a file named **Add.x** which contains:

```
1   /* nddsgen.C++.NDDSType RequestMsg; */
2   /* nddsgen.C++.NDDSType ReplyMsg; */
3   /* nddsgen.C++.ServerListener ReplyMsg:RequestMsg; */
4   /* nddsgen.C++.output.extension cxx; */
5
6   struct RequestMsg {
7       int x;
8       int y;
9   };
10
11  struct ReplyMsg {
```

```
12      int sum;
13  };
```

Lines 1-2 As in Lesson 1, the *NDDSType* token tells **nddsgen** which exchange language structure to use to build the *NDDS* type.

Line 3 What is different from Lesson 1 is the **ServerListener** token in place of the **IssueListener** token. Since a server listener must know about the *NDDS* type of both the reply message and the request message, the reply and request *NDDS* types are specified with a **:** between them—the reply *NDDS* type followed by **:** and then the request *NDDS* type.

3. Type at the command prompt:

```
nddsgen Add.x -example <architecture>:client
```

where **<*architecture*>** is the architecture switch for your system (see Table 2.1).

4. Verify that **nddsgen** generated **Add.h**, **Add.cxx**, **Add_client.cxx**, **Add_server.cxx**, and the following:

- makefile_Add_**<*architecture*>,** if you specified a UNIX architecture.
- **Add.dsw, Add_client.dsp,** and **Add_server.dsp,** if you specified a Windows architecture.

2.6.3 Edit Add_server.cxx

Create a server that adds two integers sent by the client, and then sends the results back to the client. Once created, the server just sits in a **while()** loop. The real action takes place in the **OnServiceRequest()** method of the server listener class you provide when creating the server.

Continue working in the **myClientServer** directory you created in Section 2.6.2.

To create a server for the *add* service:

1. Open **Add_server.cxx** and review the generated code.

If you finished Lesson 2, you are familiar with the beginning part of the code, which:

- Sets the verbosity level with **NddsVerbositySet()**.
- Initializes the application in an *NDDS* domain with **NDDSDomainClass::Create()**.
- Instantiates the request and reply *NDDS* type objects.

Derive and implement your server listener class

A server performs a service by listening for a client request to process. To listen for a request, a server needs to derive its own listener class and implement the **OnServiceRequest()** method. This method is invoked when a service request is received. In this exercise, you will add the two numbers from the request message.

2. At the beginning of **MyServiceListener::OnServiceRequest()**, add:

```
ReplyMsg   *itemReply   = ReplyGet(reply);
RequestMsg *itemRequest = RequestGet(request);
```

to cast the data to the correct class.

3. Below the line:

```
/* modify the data replied here */
```

Add:

```
itemReply->sum = itemRequest->x + itemRequest->y;
```

to add the two numbers in the request message.

Create a server

4. Notice the following lines before the **while()** loop in **serverMain()**; they create a server for "Service1":

```
if(!(server = new NDDSServerDerivable(domain, "Service1", instanceReply,
                                      instanceRequest, serviceListener,
                                      properties)) ||
   !server->IsValid()) {
    return 0;
}
```

Table 2.5 lists the arguments specified.

Table 2.5 **Arguments to ServerCreate()**

Argument	Value	Description
domain	NDDSDomain-Derivable object	A previously constructed NDDSDomainDerivable object.
serviceName	"Service1"	This subscription will invoke the subscription callback routine as soon as a valid issue is received.
reply	instanceReply	Temporary holder for the outgoing reply to the client.
request	instanceRequest	Temporary holder for the incoming request from clients.

Table 2.5 **Arguments to ServerCreate()**

Argument	Value	Description
listener	serviceListener	OnServiceReceived() method of this NDDSServerListenerClass is invoked when a valid request is received.
properties.mode	NDDS_SERVER_IMMEDIATE	ServerRoutine() will be invoked as soon as a valid request is received.
properties.strength	1	Allows arbitration among multiple servers. The value does not matter if there is only one server for "Service1".

5. Save your changes. The final code should be the same as that in **<NDDSTutorialDir>/clientServer** directory.

2.6.4 Edit Add_client.cxx

Goal Create a client that will continuously send a pair of increasing integers. For each service request wait at least 2.0 seconds and at most 5.0 seconds for the reply.

As illustrated in Figure 2.4, the minimum and maximum waits are the parameters governing the "best and then first" semantics. That is, if you ran multiple servers for "Service1", with one server stronger than the rest, you would see that the client waits for 2.0 seconds, and then comes back with the reply from the strongest server. If the client did not get a reply from any server within the 2.0 second wait, then it will report the first reply it gets from any server from that point on, until the deadline expires after 5.0 seconds.

Continue working in the **myClientServer** directory you created in Section 2.6.2.

To create a client for the *add* service:

1. Open **Add_client.cxx** and review the generated code.

   The beginning part of the code does the same thing as the server code:

   - Sets the verbosity level with **NddsVerbositySet()**.
   - Initializes the application in an *NDDS* domain with **new NDDSDomainDerivable()**.
   - Instantiates the request and reply *NDDSType* objects.

Create a client

2. Before calling **ServerWait()** in **clientMain()**, add the following lines to create a client to "Service1" with **new NDDSClientDerivable()**:

```
if(!(client = new NDDSClientDerivable(domain, "Service1", instanceReply,
                                      instanceRequest)) ||
   !client->IsValid()) {
    return 0;
}
```

Table 2.6 lists the arguments specified.

Table 2.6 **Arguments to NDDSClientDerivable()**

Argument	Value	Description
domain	NDDSDomainDerivable object	A handle to a NDDSDomainDerivable object.
serviceName	"Service1"	Must match the ServiceName for the server.
reply	instanceReply	Temporary holder of for the outgoing reply to the client.
request	instanceRequest	Temporary holder for the incoming request from clients.

Wait for a server to appear (optional)

3. *(Optional)* Since clients and servers are anonymous, the clients do not know by default whether a server for the desired service exists. If there is in fact no server, or something prevents the client from receiving the server reply, the client would only find out after the maximum wait for the request expires. Optionally, you can ensure that a server exists by calling **ServerWait()**.

After creating the client and before sending the request, add:

```
if (client->ServerWait(waitTime, 5, 1) == RTI_FALSE) {
    printf("There is no server. Exiting...\n");
    return RTI_FALSE;
}
```

where waitTime is 2 seconds. With this call, you are telling the client to try to find at least one server, retrying up to 5 times and waiting 2 seconds between retries. If the client still cannot find a server, this exercise simply exits the program.

Enter the request message

4. Since you already have the count variable, which is incremented in the **for()** loop, send its value in the request message.

Below the line:

```
/* modify the data requested here */
```

Add the following lines:

```
instanceRequest->x = count;
instanceRequest->y = count;
```

Send the request and wait

When sending a request, clients can either:

- Block for a reply by calling **CallAndWait()**.
- Continue with other tasks while passively waiting for the reply by calling **Call().**

Since you have nothing else to do (probably not true in your real application), block waiting for the reply for up to 5.0 seconds. Even if the reply comes back immediately, you will need to wait for at least 2.0 seconds.

Check the status of the reply

When the reply comes back before the deadline, report the contents of the reply with **Print()**, which is defined in **Add.h**.

5. Save your changes. The final code should be the same as that in **<NDDSTutorialDir>/clientServer** directory.

2.6.5 Build the Client and Server Programs

Building the client-server pair is no different than building the publication-subscription pairs in Lessons 2. For more information on building *NDDS* applications, see Section 2.3.6.

2.6.6 Run the Client Only

Try starting the client to see what happens. It should fail after 10 seconds.

To run the client program:

❑ On UNIX systems, type at a command prompt:

```
objs/<architecture>/Add_client
```

where ***<architecture>*** is the architecture switch for your system (see Table 2.1).

❑ On Windows systems, type at a command prompt:

```
objs\i86Win32VC60\Add_client
```

❑ On VxWorks systems, type at the console or *windsh*:

```
sp (clientMain, 0, 0)
```

2.6.6.1 Client Screen Output

The client should wait for a server to appear for 10 seconds and then quit. You should see:

```
Allocate ReplyMsg and RequestMsg types.
There is no server. Exiting...
```


2.6.7 Start the Server First and Then the Client

Start the server first.

To run the server program:

- ❑ On UNIX systems, type at a command prompt:

  ```
  objs/<architecture>/Add_server
  ```

  where **<*architecture*>** is the architecture switch for your system (see Table 2.1).

- ❑ On Windows systems, type at a command prompt:

  ```
  objs\i86Win32VC60\Add_server
  ```

- ❑ On VxWorks systems, type at the console or *windsh*:

  ```
  sp (serverMain, 0, 0)
  ```

2.6.7.1 Client Screen Output

Now start the client as specified in Section 2.6.6. You should see:

```
Allocate ReplyMsg and RequestMsg types.
Requesting the time, count 0
ReplyMsg:
    sum: 0
Requesting the time, count 1
ReplyMsg:
    sum: 2
Requesting the time, count 2
ReplyMsg:
    sum: 4
Requesting the time, count 3
ReplyMsg:
    sum: 6
...
```

2.6.7.2 Server Screen Output

The server reports the requests it is receiving.

```
Allocate RequestMsg and ReplyMsg types.
Sleeping for 10.000000 sec...
RequestMsg:
    x: 0
    y: 0
RequestMsg:
    x: 1
    y: 1
RequestMsg:
    x: 2
    y: 2
Sleeping for 10.000000 sec...
RequestMsg:
    x: 3
    y: 3
...
```

Chapter 3

Advanced Exercises

Read the Overview and perform the Basic Exercises before you attempt the Advanced Exercises. This chapter includes the following lessons on the advanced capabilities of *NDDS*:

- ❏ Lesson 6: Publish and Subscribe Reliably (Section 3.1)
- ❏ Lesson 7: Publish and Subscribe Using Multicast (Section 3.2)
- ❏ Lesson 8: Subscribe Using Patterns (Section 3.3)

These advanced lessons do not require you to make any code changes. You will be reviewing, and reading the explanation of, the supplied example code. The steps to follow for C++ are included in this chapter. For C, read the tutorial information in this chapter and then refer to Chapter 5 to complete each lesson.

3.1 Lesson 6: Publish and Subscribe Reliably

Goal

Create a reliable publication-subscription pair to send and receive the "Hello World!" message of the HelloMsg *NDDS* type created in Lesson 1.

In this exercise, you will review the publication and the subscription code in the **<NDDSTutorialDir>/reliable** directory and discover the calls necessary to send issues reliably. You will learn that the code is similar to the unreliable publication and sub-

scription created in Lesson 2. The *NDDS* type and the message content is exactly the same. There are no code changes for you to make in this lesson.

3.1.1 Reliability and Time-Determinism

Most communication architectures provide one or more reliability models. Not all models are equally suitable for real-time applications. For instance, a transactional model (e.g. banking application) is highly reliable but has extremely poor determinism (its time behavior is widely varying and unbounded). The transactional model is therefore unsuitable for real-time applications.

The optimal balance between determinism (time-reliability) and data-delivery reliability varies between real-time applications and among different subscriptions within the same application. *NDDS* provides a mechanism for the application to *customize* the determinism/reliability trade-off on a per-subscription basis. Moreover, reliability impacts memory usage because extra memory buffers have to be maintained for retries, time-outs, etc. *NDDS* provides the mechanism necessary for the application to predict and control memory usage.

NDDS's reliability model is subscription-driven: a publication does not directly specify whether the data should be sent reliably. However, services are provided to allocate publication memory and control issue flow. If there is a matching subscription that wishes to receive the issues reliably from this publication, *NDDS* will send issues reliably.

This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the hands-on part of the lesson. If you are using C, refer to Section 5.1 to complete the lesson.

3.1.2 Learn How to Create a Reliable Publication

Open and review **Hello_publisher.cxx** in the **<NDDSTutorialDir>/reliable** directory. The main body of code is shown in Figure 3.1. There are no code changes required.

Figure 3.1 **Reliable Publication Code**

```
1   extern "C" int publisherMain(int nddsDomain, int nddsVerbosity)
2   {
3       int                      count           = 0;
4       RTINtpTime               send_period_sec = {0,0};
5       NDDSPublicationProperties properties;
6       NDDSPublicationDerivable *publication     = NULL;
7       NDDSDomainDerivable      *domain          = NULL;
8       HelloMsg                 *instance        = NULL;
9       RTINtpTime heartBeatTimeout = {0,0};
10      RTINtpTime waitTime         = {0,0};
11
12      RtiNtpTimePackFromNanosec(send_period_sec, 4, 0); /* 4 seconds */
13      RtiNtpTimePackFromNanosec(waitTime, 2, 0);        /* 2 seconds */
14
15      NddsVerbositySet(nddsVerbosity);
16
17      if(!(domain = new NDDSDomainDerivable(nddsDomain, NULL, NULL)) ||
18          !domain->IsValid()) {
19          return 0;
20      }
21
22      if(!(instance = new HelloMsg())) {
23          return 0;
24      }
25
26      domain->PublicationPropertiesGet(&properties);
27      properties.sendQueueSize = 5;
28      properties.lowWaterMark = 1;
29      properties.highWaterMark = 4;
30      RtiNtpTimePackFromMillisec(properties.heartBeatTimeout, 0, 500);
31      RtiNtpTimePackFromNanosec(properties.sendMaxWait, 2, 0);
32      RtiNtpTimePackFromNanosec(properties.persistence, 15, 0);
33      properties.strength        = 1;
34
35      if(!(publication = new NDDSPublicationDerivable(domain,
36                                          "Reliable HelloMsg",
37                                           instance, &properties)) ||
38           !publication->IsValid()) {
39          return 0;
40      }
41
```

```
42      if(publication->SubscriptionWait(waitTime, 5, 1) !=
43                                                  NDDS_WAIT_SUCCESS) {
44          printf("There is no subscription to the topic. "
45                 "Might as well exit.\n");
46          return 0;
47      }
48
49      for (count=0;;count++) {
50          printf("Sampling publication, count %d\n", count);
51
52          /* modify the data to be sent here */
53          sprintf(instance->msg, "Hello World! (%d)", count);
54
55          if (publication->Send() != NDDS_PUBLICATION_SUCCESS) {
56              printf("Issue %d not sent\n", count);
57              return 0;
58          }
59          NddsUtilitySleep(send_period_sec);
60      }
61      return 1;
62  }
```

Line 5 We create an **NDDSPublicationProperties** structure on the stack to tailor the publication behavior for reliability.

Lines 12-13 These lines convert time values from seconds and nanoseconds into **RTINtpTime** structures. This is our operating system-independent time representation. It is based on the NTP time format. Time is not represented in a human readable form. Instead it is organized in a way that makes 64-bit arithmetic easy. We have provided a set of functions to convert to and from **RTINtpTime**. See Appendix A for details.

Lines 15-23 The steps to create a reliable publication are identical to the best-effort publication, including creating a domain and allocating a publication instance.

Lines 25-33 Tailor reliable behavior of the publication through publication properties. When a publication has reliable subscriptions, it is functionally different than a regular (best-effort, unreliable) publication in the following ways:

- ❏ It checks for acknowledgements from the reliable subscriptions, and resends previously published issues if a subscription reports that issue(s) as missing. The *sendQueueSize* field of the publication's property specifies the maximum number of issues the publication remembers.
- ❏ In addition to passively checking the acknowledgement messages from the reliable subscriptions, the publication can (optionally) actively poll for the remote subscription's receipt status periodically. We set the *heartBeatTimeout* field to 0.5

second to poll for the subscription status twice a second. This feature is not necessary for this simple example, but we show it here in case you want to use it later in your application.

- If a subscription falls behind despite the publication's efforts, the publication's queue will eventually fill up, and the send call will block for the *sendMaxWait* seconds, and then return error. We want to block only a finite time: 2.0 seconds here.

Lines 35-40 We create the publication with the Derivable API.

Lines 42-47 Even though a reliable publication is allowed to send the issues as soon as it changes the queue size, there is no guarantee that a subscription exists to receive the issues. If you start publishing anyway without ensuring the existence of a subscription, it somewhat defeats the purpose of using reliable subscriptions. **SubscriptionWait()** is a convenience function for ensuring the existence of a specified number of subscriptions to the publication's *NDDS* topic. It is also a deterministic call because you will wait no more than *waitTime* (2 seconds) times *retries* (10 times), which is 20 seconds in this exercise.

Lines 55-58 If the send queue becomes full, a reliable publication blocks waiting for a free space, up to *sendMaxWait*, at the end of which the send call returns an error. You are free to ignore it and simply note that the issue was not sent. Here, we take the most dramatic action of quitting the program. You can also wait for the queue level to drop to a value you specify with **Wait()**, and ensure that all published issues have been acknowledged by the reliable subscriptions.

3.1.3 Learn How to Create a Reliable Subscription

Open and review **Hello_subscriber.cxx** in the **<NDDSTutorialDir>/reliable** directory. The main body of code is shown in Figure 3.2. There are no code changes required.

Figure 3.2 **Reliable Subscription Code**

```
1   extern "C" int subscriberMain(int nddsDomain, int nddsVerbosity)
2   {
3       NDDSSubscriptionProperties properties;
4       NDDSSubscriptionDerivable *subscription   = NULL;
5       MyListener            *listener       = NULL;
6       NDDSDomainDerivable   *domain         = NULL;
7       HelloMsg          *instance       = NULL;
8       char deadlineString[RTI_NTP_TIME_STRING_LEN];
9
10      NddsVerbositySet(nddsVerbosity);
11
```

```
12      if(!(domain = new NDDSDomainDerivable(nddsDomain, NULL, NULL)) ||
13          !domain->IsValid()) {
14          return 0;
15      }
16
17      /*
18        Set subscription properties.
19       */
20      domain->SubscriptionPropertiesGet(&properties);
21      RtiNtpTimePackFromNanosec(properties.deadline,10,0);
22      properties.mode = NDDS_SUBSCRIPTION_IMMEDIATE;
23      properties.receiveQueueSize = 5;
24
25      /*
26       * Allocate the object. Nddsgen generates allocation code for
27       * internal structures.
28       */
29      if(!(instance = new HelloMsg())) {
30          return 0;
31      }
32
33      if(!(listener = new MyListener())) {
34          return 0;
35      }
36
37      if(!(subscription =
38          new NDDSSubscriptionReliableDerivable(domain,
39                                                "Reliable HelloMsg",
40                                                instance, listener,
41                                                &properties)) ||
42          !subscription->IsValid()) {
43          return 0;
44      }
45
46      while (1) {
47          /*subscription->Poll() only needed if NDDS_SUBSCRIPTION_POLLED*/
48
49          /*
50           * We sleep only to kill time.  Nothing need be done here
51           * for an NDDS_SUBSCRIPTION_IMMEDIATE subscription.
52           */
53          printf("Sleeping for %s sec...\n",
54                 RtiNtpTimeToString(properties.deadline, deadlineString));
55          NddsUtilitySleep(properties.deadline);
56      }
57      return 1;
58  }
```

Line 3 We create an **NDDSSubscriptionProperties** structure on the stack to tailor the subscription behavior for reliability.

Lines 10-35 The steps to create a reliable subscription are identical to the best-effort subscription, including creating a domain, allocating a subscription instance, and creating an issue listener. There are differences in how the properties are initialized, however.

Lines 17-23 Tailor reliable behavior of the subscription through subscription properties.

A reliable subscription is different than a best-effort subscription in the following ways:

- ❑ It checks the sequence number of the received issues and puts them in order before handing them off to the application. Upon discovery of lost issue, it reserves space for the missing issue and requests retransmission. The *receiveQueueSize* field of the subscription's property specifies the maximum number of issues the subscription can cache while it retries lost issues.
- ❑ You cannot specify minimum separation for a reliable subscription. If you want to receive issues in order, you must accept all issues. Hence we specify the deadline but not the minimum separation.

Lines 37-42 We create the reliable subscription with the Derivable API. Since we do not want to be notified of any reliable-subscription-related events, we simply omit setting up a reliable listener in this example. Note that we are still specifying an issue listener, as we must to create any subscription.

3.1.4 Build the Reliable Subscription and Publication Programs

To build the reliable subscription and publication programs:

1. Copy the contents of the **<NDDSTutorialDir>/reliable** directory into a new directory of your own, such as **myreliable**.
2. Follow the build steps listed in Section 2.3.6, but use your new **myreliable** directory.

3.1.5 Run the Reliable Subscription and Publication Programs

To run the reliable subscription and publication programs:

1. Start the publication first. Since there is no subscription, the publication should wait for 20 seconds then quit.
2. Start the subscription.

3. Start the publication again.

Note: The output from the reliable subscription is the same as what you had seen from the best-effort publication in Lesson 2. The *NDDS* reliability mechanism is transparent to the application.

4. Now kill the subscription with <**Ctrl-c**>. The publication keeps publishing until its send queue fill up, blocks for 2 seconds, and finally quits the program because the send call returned an error because it timed out on the full queue. For UNIX-based operating systems you can test the reliability by suspending <**Ctrl-z**> the subscriber process for a couple of seconds and then restoring it <**bg**>. The subscriber gets the missed issues (in the right order!).

3.2 Lesson 7: Publish and Subscribe Using Multicast

Goal

Create a multicast publication-subscription pair to send and receive the "Hello World!" message of the HelloMsg *NDDS* type created in Lesson 1.

In this exercise, you will review the publication and the subscription code in the **<NDDSTutorialDir>/multicast** directory and discover the calls necessary to send and receive issues on a multicast address. You will learn that the code is very similar to the unicast publication and subscription code in the **<NDDSTutorialDir>/hello** directory, created in Lesson 2. The *NDDS* type and the message content will be exactly same. There are no code changes for you to make in this lesson.

3.2.1 Using Multicast in NDDS

NDDS takes advantage of multicasting on systems that support it. With multicasting, applications can send data to multiple subscribers efficiently. This is useful for users who want to send a large amount of data to multiple nodes simultaneously.

NDDS integrates multicasting seamlessly. The API and behavior is transparent. All *NDDS* features such as reliability and transparent redundancy are the same whether you are using unicast or multicast.

To use multicast an *NDDS* application must:

1. Enable multicasting and set the *Time-To-Live* (TTL) value appropriate for the multicast data for the application during initialization of *NDDS*.

   TTL is the number of hops or gateways a message may take before it is discarded, thus controlling the extent of the multicast data.

2. If the application is a subscriber, specify a multicast address at creation time.

   We recommend using a multicast address in the range of 225.0.0.0 and higher.

Note: A publication does not specify a multicast address. A subscription specifies whether and on which multicast address to receive issues. This is consistent with other aspects of receiving messages, such as reliability. If a subscription specifies a multicast address, the matching publication(s) will send the issues on the requested multicast address.

This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the hands-on part of the lesson. If you are using C, refer to Section 5.2 to complete the lesson.

3.2.2 Learn How to Create a Multicast Publication

Open **Hello_publisher.cxx** in the **<NDDSTutorialDir>/multicast** directory. The main body of code is shown in Figure 3.3.

Line 8 Create an **NDDSDomainProperties** structure variable to turn on multicasting and set the TTL.

Lines 13-23 Copy the default application properties into the **NDDSDomainProperties** structure before changing the multicast properties structure. Multicasting is enabled and the TTL is set to restrict the messages to within the same subnet. The application is finally initialized with these multicast properties during **new NDDSDomainDerivable()**.

Lines 24-47 The rest of this example is the same as the unicast publication code in the **<NDDSTutorialDir>/hello** directory (see Lesson 2).

Figure 3.3 **Multicast Publication Code**

```
1   extern "C" int publisherMain(int nddsDomain, int nddsVerbosity)
2   {
3       int                       count           = 0;
4       RTINtpTime                send_period_sec = {0,0};
5       NDDSPublicationDerivable *publication     = NULL;
6       NDDSDomainDerivable      *domain          = NULL;
7       HelloMsg                 *instance        = NULL;
8       NDDSDomainProperties      domainProperties;
9       NDDSPublicationProperties publProperties;
10
11      RtiNtpTimePackFromNanosec(send_period_sec, 4,0); /* 4 seconds */
12
13      NddsDomainPropertiesDefaultGet(&domainProperties);
14      domainProperties.multicast.enabled = RTI_TRUE;
15      domainProperties.multicast.ttl     = NDDSTTLSameSubnet;
16
17      NddsVerbositySet(nddsVerbosity);
18
19      if(!(domain = new NDDSDomainDerivable(nddsDomain,
20                                            &domainProperties)) ||
21          !domain->IsValid()) {
22          return 0;
23      }
24
25      domain->PublicationPropertiesGet(&publProperties);
26      RtiNtpTimePackFromNanosec(publProperties.persistence, 15, 0);
27      publProperties.strength = 1;
28
29      if(!(instance = new HelloMsg())) {
30          return 0;
31      }
32
33      if(!(publication = new NDDSPublicationDerivable(domain,
34                      "Multicast HelloMsg", instance, &publProperties)) ||
35          !publication->IsValid()) {
36          return 0;
37      }
38
39      for (count=0;;count++) {
40          printf("Sampling publication, count %d\n", count);
41          /* modify the data to be sent here */
42           sprintf(instance->msg, "Hello World! (%d)", count);
43          publication->Send();
44          NddsUtilitySleep(send_period_sec);
45      }
46      return 1;
47  }
```

3.2.3 Learn How to Create a Multicast Subscription

Open **Hello_subscriber.cxx** in the **<NDDSTutorialDir>/multicast** directory. The main body of code is shown in Figure 3.4.

Figure 3.4 **Multicast Subscription Code in C++**

```
1   extern "C" int subscriberMain(int nddsDomain, int nddsVerbosity)
2   {
3       NDDSSubscriptionDerivable *subscription   = NULL;
4       MyListener                *listener       = NULL;
5       NDDSDomainDerivable       *domain         = NULL;
6       HelloMsg                  *instance       = NULL;
7       char                   deadlineString[RTI_NTP_TIME_STRING_LEN];
8       NDDSDomainProperties       domainProperties;
9       NDDSSubscriptionProperties subsProperties;
10
11      NddsVerbositySet(nddsVerbosity);
12
13      NddsDomainPropertiesDefaultGet(&domainProperties);
14      domainProperties.multicast.enabled = RTI_TRUE;
15      domainProperties.multicast.ttl     = NDDSTTLSameSubnet;
16      if(!(domain = new NDDSDomainDerivable(nddsDomain,
17                                            &domainProperties)) ||
18          !domain->IsValid()) {
19          return 0;
20      }
21
22      domain->SubscriptionPropertiesGet(&subsProperties);
23      subsProperties.mode = NDDS_SUBSCRIPTION_IMMEDIATE;
24      RtiNtpTimePackFromNanosec(subsProperties.deadline,10,0);
25      RtiNtpTimePackFromNanosec(subsProperties.minimumSeparation,0,0);
26
27      /*
28       * Allocate the object. Nddsgen generates allocation code for
29       * internal structures.
30       */
31      if(!(instance = new HelloMsg())) {
32          return 0;
33      }
34
35      if(!(listener = new MyListener())) {
36          return 0;
37      }
38
```

```
39      if(!(subscription = new NDDSSubscriptionDerivable(domain,
40                  "Example HelloMsg", instance, listener, &subsProperties,
41                  NddsStringToAddress("225.0.0.1"))) ||
42          !subscription->IsValid()) {
43          return 0;
44      }
45
46      while (1) {
47          /*subscription->Poll() only needed if NDDS_SUBSCRIPTION_POLLED*/
48
49          /*
50           * We sleep only to kill time.  Nothing need be done here
51           * for an NDDS_SUBSCRIPTION_IMMEDIATE subscription.
52           */
53          printf("Sleeping for %s sec...\n",
54                 RtiNtpTimeToString(subsProperties.deadline,
55                                    deadlineString));
56          NddsUtilitySleep(subsProperties.deadline);
57      }
58      return 1;
59  }
```

Line 8 Similar to a publication, an **NDDSDomainProperties** structure is created to turn on multicasting and set the TTL.

Lines 13-20 The steps to turn on multicasting and set the TTL before initializing *NDDS* are the same as for the multicast publication.

Lines 22-37 The steps to create a reliable subscription are identical to the best-effort subscription, including setting properties, creating a domain and allocating a publication instance.

Lines 39-44 The only difference between a unicast subscription and a multicast subscription is in the last argument to the create call. For the unicast subscription in Lesson 2, you passed in **NDDS_USE_UNICAST** to indicate a unicast subscription. This time, you specify the host byte order integer for 225.0.0.1 with **NddsStringToAddress()**, which converts the string form of an IP address into a host byte-ordered address.

Lines 46- The rest of this example is the same as the unicast publication code in the **<NDDSTutorialDir>/hello** directory (see Lesson 2).

3.2.4 Build the Multicast Subscription and Publication Programs

To build the multicast subscription and publication programs:

1. Copy the contents of the **<NDDSTutorialDir>/multicast** directory into a new directory of your own, such as **mymulticast**.

2. Follow the build steps listed in Section 2.3.6, but use your new **mymulticast** directory.

3.2.5 Run the Multicast Subscription and Publication Programs

To run the multicast subscription and publication programs:

1. Start the subscription.
2. Start the publication.

Note: The output from the multicast subscription is the same as what you had seen from the best-effort publication in Lesson 2. The *NDDS* multicast mechanism is transparent to the application.

3.3 Lesson 8: Subscribe Using Patterns

Goal

In this exercise, you will review the publication and the subscription code in the **<NDDSTutorialDir>/pattern** directory and discover the calls necessary to implement the pattern subscription strategy shown in Figure 3.5.

There are no code changes for you to make in this lesson.

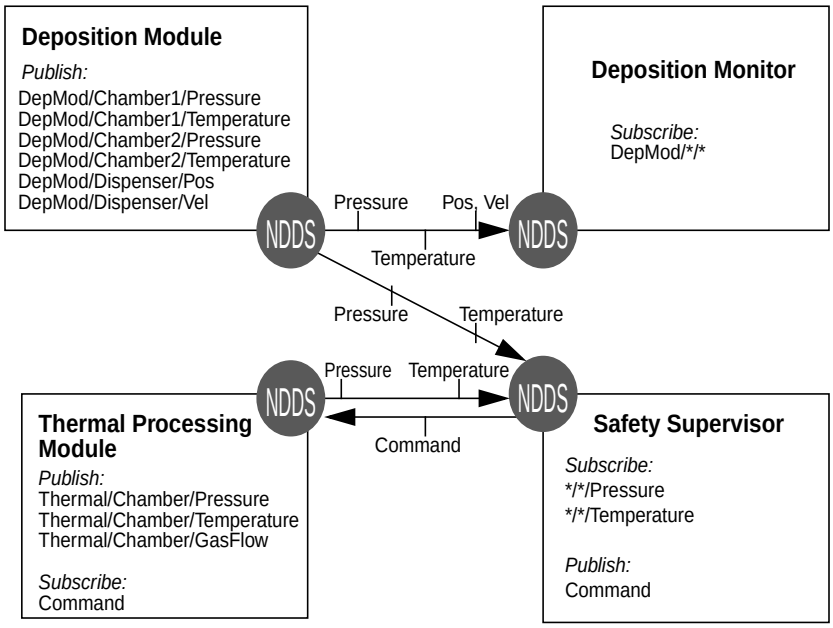
3.3.1 Pattern Subscriptions

With the pattern subscription feature, you can subscribe to sets of related *NDDS topics*. An *NDDS* subscriber may subscribe to a *topic* or *type pattern* in place of the explicit *NDDS topic* and *NDDS type*. The *NDDS topic* and *NDDS type* patterns are simple regular expressions that are matched against the full *NDDS topics* and *NDDS types* for all current and future publications. The pattern subscription feature allows you to group publications and subscribe to related *NDDS topics*. For instance, a monitoring program could subscribe to all types of alarm conditions by specifying a subscription to *NDDS topic* "Alarm*." The applications would then receive a publication of AlarmTemperature, AlarmTimeout, and AlarmError.

Hierarchical NDDS Topics

The pattern subscription feature supports a hierarchical organization for *NDDS topics*, which can help you organize your data, especially if your system is complex. Figure 3.5 illustrates the use of pattern topics in a semiconductor processing application. In the diagram, the chemical vapor deposition module is overseen by the deposition monitor node and the safety supervisor node. The safety supervisor node also monitors the ther-

Figure 3.5 **Pattern Subscription Example: Semiconductor Processing**



mal processing module and sends commands *reliably*. There are two deposition chambers and a dispenser in the deposition module.

In the above case, the use of pattern subscriptions reduces the amount of code on the subscriber side drastically, and allow flexible subscription strategy. For example, the Deposition Monitor subscribes to all of the Deposition Module publications with a single function. The flexibility derives from the fact that new publications are automatically added if they adhere to the same naming convention.

This completes the general discussion for this lesson. If you are using C++, the next sections walk you through the code for the lesson. If you are using C, refer to Section 5.3 to complete the lesson.

3.3.2 Review the Deposition Monitor Code

Open **depMonitor.cxx** in the **<NDDSTutorialDir>/pattern** directory. The main body of the code, seen in Figure 3.6, shows how to subscribe to the pattern *NDDS topic* "DepMod/*/*".

Figure 3.6 **Subscribing to Patterns in C++**

```
1   class DataPatternSubscriptionClass : public NDDSSubscriberPatternClass {
2     public:
3       virtual ~DataPatternSubscriptionClass();
4       DataPatternSubscriptionClass(class NDDSDomainDerivable *domain,
5                                    class DataListener *listener);
6       virtual NDDSSubscriptionClass *OnMatch(const char *nddsTopic,
7                                              const char *nddsType);
8     private:
9       class NDDSDomainDerivable *_domain;
10      class DataListener        *_listener;
11  };
12
13  DataPatternSubscriptionClass::~DataPatternSubscriptionClass() {}
14  DataPatternSubscriptionClass::
15  DataPatternSubscriptionClass(class NDDSDomainDerivable *domain,
16                               class DataListener *listener) :
17      _domain(domain),
18      _listener(listener)
19  {
20  }
21  NDDSSubscriptionClass *
22  DataPatternSubscriptionClass::OnMatch(const char *nddsTopic,
23                                        const char *nddsType)
24  {
25      NDDSSubscriptionProperties properties;
26      NDDSSubscriptionDerivable *subscription;
27      Data                      *instance;
28
29      _domain->SubscriptionPropertiesGet(&properties);
30      RtiNtpTimePackFromNanosec(properties.deadline, 10,0);
31      RtiNtpTimePackFromNanosec(properties.minimumSeparation, 0, 0);
32      properties.mode             = NDDS_SUBSCRIPTION_IMMEDIATE;
33
34      if(!(instance = new Data())) {
35          return 0;
36      }
37
38      subscription = new NDDSSubscriptionDerivable(_domain, nddsTopic,
39                                                   instance,_listener,
40                                                   &properties,
41                                                   NDDS_USE_UNICAST);
```

```
42      if (subscription->IsValid()) {
43          return subscription;
44      }
45
46      return NULL;
47  }
48
49  extern "C" int DepositionMonitor(int nddsDomain, int nddsVerbosity)
50  {
51      NDDSSubscriberDerivable *depMonitor     = NULL;
52      MyListener              *listener       = NULL;
53      NDDSDomainDerivable     *domain         = NULL;
54      RTINtpTime               sleepTime = {10, 0};
55      DataPatternSubscriptionClass *myPatternSubscription = NULL;
56
57      NddsVerbositySet(nddsVerbosity);
58      if(!(domain = new NDDSDomainDerivable(nddsDomain)) ||
59          !domain->IsValid()) {
60          return 0;
61      }
62
63      if(!(listener = new MyListener())) {
64          return 0;
65      }
66
67      if(!(depMonitor = new NDDSSubscriberDerivable(domain)) ||
68          !depMonitor->IsValid()) {
69          return 0;
70      }
71
72      if(!(myPatternSubscription =
73           new DataPatternSubscriptionClass(domain, listener))) {
74          return 0;
75      }
76
77      depMonitor->PatternAdd("DepMod/*/*", "Data", myPatternSubscription);
78
79      while (1) {
80          NddsUtilitySleep(sleepTime);
81      }
82
83      return 1;
84  }
```

Line 67 In Lesson 4, you learned that a subscriber manages subscriptions. Since the pattern subscription feature involves creating multiple subscriptions for each match, you specify the desired pattern *NDDS topic* and *NDDS types* to a subscriber, which can create a subscription for the match. Therefore, you need to create a subscriber first.

Line 77 The **PatternAdd()** method tells the subscriber (**depMonitor**) what pattern topic ("**DepMod/*/***") and pattern type ("**Data**") to match. If there is a match, the subscriber invokes the **OnMatch()** method of the object (**myPatternSubscription**) of a sub-class of **NDDSSubscriberPatternClass** to create a subscription for the match.

Lines 21-47 You told *NDDS* to invoke **myPatternSubscription's** (an instance of the **DataPatternSubscriptionClass**) **OnMatch()** method when it discovers a publication matching the pattern topic/type. With this method, you have the flexibility to:

- ❑ Create a subscription for all matches, as you are doing here. Note that all subscriptions will use the same deadline, minimum separation, callback routine, and data instance, but this is purely for simplification purposes. Each subscription can have different parameters.
- ❑ Do other things before returning the subscription to *NDDS*. The user parameter is convenient for passing information in either direction.

If you want to implement the **OnMatch()** method:

1. Derive your own pattern subscriber class from the **NDDSSubscriberPatternClass**.
2. Implement the **OnMatch()** method of the derived class.

3.3.3 Review the Safety Supervisor Code

Open **supervisor.cxx** in the **<NDDSTutorialDir>/pattern** directory. Even though the safety supervisor publishes as well as subscribes, you will examine only the pattern subscription part in this exercise. You already learned how to subscribe to a pattern in the deposition monitor program. The safety supervisor code in Figure 3.7 shows that you can subscribe to *multiple* pattern topics with one subscriber: in this example, the Safety Supervisor subscribes to the patterns "*/*/Pressure" and "*/*/Temperature".

Figure 3.7 **Subscribing to Multiple Patterns in C++**

```
1   if(!(safetySubscriber = new NDDSSubscriberDerivable(domain) ||
1      !safetySubscriber->IsValid()) {
2       return 0;
3   }
4   if(!(myPatternSubscription =
5      new DataPatternSubscriptionClass(domain,listener))) {
6       return 0;
7   }
8   safetySubscriber->PatternAdd("*/*/Pressure", "Data",
9                            myPatternSubscription);
10  safetySubscriber->PatternAdd("*/*/Temperature", "Data",
11                           myPatternSubscription);
```

3.3.4 Review the Deposition Module

Review the code in **depModule.cxx** in the **<NDDSTutorialDir>/pattern** directory. The deposition module creates six publications with a hierarchical naming scheme. If you need a refresher on the concept of creating publications, see Lesson 2.

3.3.5 Review the Thermal Processing Module Code

Review the code in **thermProcess.cxx** in the **<NDDSTutorialDir>/pattern** directory. The thermal processing module creates three publications with a hierarchical naming scheme, and creates a reliable subscription to the command. If you need a refresher on the concept of creating publications and subscriptions, see Lesson 2.

3.4 Congratulations!

You can now develop any distributed real-time application. Your next steps are:

- ❑ Develop your application. See Chapter 3 in the *NDDS User's Manual* for guidance.
- ❑ Refer to the other advanced examples in the **examples/** directory. To test *NDDS*'s performance on your system, work in the **examples/performance/** directory.
- ❑ Consult with RTI on using *NDDS* to meet your design goals. For consulting rates and scheduling, contact your sales representative or send e-mail to RTI at **info@rti.com**.

Chapter 4

Basic C Exercises

This chapter provides instructions on how to perform the same exercises presented in Chapter 2 if you are using C language.

- ❏ Lesson 1: Use nddsgen to Auto-Create NDDS Types (Section 4.1)
- ❏ Lesson 2: Create a Publication and a Subscription (Section 4.2)
- ❏ Lesson 3: Create a Polled Subscription (Section 4.3)
- ❏ Lesson 4: Create a Publisher and a Subscriber (Section 4.4)
- ❏ Lesson 5: Create a Client and a Server (Section 4.5)

You should read the general discussion sections for each lesson presented in Chapter 2, before using the instructions in this chapter.

Each exercise is available in complete source code form in the <***NDDS*TutorialDir/C**> directory of your *NDDS* installation. For convenience, we will leave off "/C" and simply refer to this directory as <***NDDS*TutorialDir**>.

The same platforms that are supported with C++ are also supported with C. Refer to the setup checklist for your platform before proceeding with the lessons:

- ❏ Compiling UNIX Applications (Section 3.1) in the *NDDS Getting Started Guide*
- ❏ Compiling Windows Applications (Section 3.2) in the *NDDS Getting Started Guide*
- ❏ Compiling VxWorks Applications (Section 3.3) in the *NDDS Getting Started Guide*

4.1 Lesson 1: Use nddsgen to Auto-Create NDDS Types

Goal Create an *NDDS* type called "HelloMsg" to send a short message to other applications.

The resulting source code should be similar to what is in the <**NDDSTutorialDir>/hello**> directory.

4.1.1 Create an NDDS Type

To create an NDDS Type called HelloMsg:

1. Create a directory called **myhello**.
2. In the **myhello** directory, create a file called **Hello.x** that contains:

```
/* nddsgen.C.NDDSType HelloMsg; */

const MAX_MSG_LEN = 128;

struct HelloMsg {
    string msg<MAX_MSG_LEN>;
};
```

The keyword *"string"* within the **HelloMsg** structure is the type used in the *NDDS* exchange language for NULL-terminated strings. The value in the angle brackets, < >, specifies a maximum size for the message, which is **MAX_MSG_LEN** here.

The line:

```
/* nddsgen.C.NDDSType HelloMsg; */
```

- Tells **nddsgen** to generate code in C.
- Declares an *NDDSType* called HelloMsg.
- Tells **nddsgen** to look for a structure called **HelloMsg** to build the HelloMsg *NDDS* type.

3. Type at the command prompt:

```
nddsgen Hello.x
```

4.1.2 Review the Generated Files

nddsgen generates all of the serialize / deserialize code for the structure **HelloMsg** in the file **Hello.x** and puts all of the code in the files **Hello.h** and **Hello.c**. The **Hello.c** file contains all of the exchange language code and *NDDS* wrapper functions, such as the serialize / deserialize and print functions. The **Hello.h** file contains the structure that you will use throughout your code.

To correct mistakes or modify the *NDDS* type, use the **-replace** argument to overwrite the current files.

The generated **Hello.h** file contains the actual data structure for the new *NDDS* type and declares the functions in **Hello.c**.

The following are the relevant lines from **Hello.h**.

```
#define MAX_MSG_LEN 128

typedef struct HelloMsg {
    char *msg;
} HelloMsg;

#define HelloMsgNDDSType "HelloMsg";
extern RTIBool HelloMsgSerialize(NDDSXDRStream nddsds,
                                 HelloMsg *instance,
                                 int options);
extern RTIBool HelloMsgNddsRegister();
extern HelloMsg *HelloMsgAllocate();
extern HelloMsg *HelloMsgDeserialize(NDDSXDRStream nddsds,
                                     HelloMsg *instance);
extern RTIBool HelloMsgPrint(HelloMsg *nddsHelloMsg,
                             unsigned int indent);
```

HelloMsg is defined to be a structure that contains a single data member: the character pointer *msg*. Although **MAX_MSG_LEN** was used to specify the maximum size of the string in **Hello.x** and defined with a **#define** in **Hello.h**, *msg* does not have a predetermined size. This gives you flexibility in allocating space for the string.

In addition to the serialize, deserialize, allocate, and print routines, **nddsgen** also creates a function that registers the type to *NDDS*: **HelloMsgNddsRegister()**. **HelloMsgNddsRegister()** and **HelloMsgAllocate()** are the two most commonly used functions in your application. The rest are internally used by *NDDS*.

4.2 Lesson 2: Create a Publication and a Subscription

Goal

Create a publication and subscription pair to send and receive “Hello World!” messages in a best-effort real-time manner. Best-effort real-time mode means that the issues will be delivered as deterministically as the underlying OS and system allows. If an issue is not received by a subscriber until the deadline expires, the subscriber is notified of the failure (the publisher continues publishing the data nevertheless). *NDDS* also supports reliable mode, where the issues are guaranteed to be delivered in-order.

The final code should be the same as in the **<NDDSTutorialDir>/hello** directory.

4.2.1 Generate Example Publication and Subscription Code

Refer to Section 2.3.3 for instructions on using **nddsgen** to create the code you need before starting this lesson.

4.2.2 Edit Hello_publisher.c

To edit the generated publication code to send the "Hello World!" message:

1. Open **Hello_publisher.c** and review the generated code.

Specify the verbosity

NDDS runs at the silent verbosity setting by default; you see only error messages. Increasing verbosity can reveal what *NDDS* is doing under the hood and is helpful for advanced debugging. To change the verbosity to 3, initialize **nddsVerbosity** to 3 in **main()**.

Initialize an NDDS application

NddsInit() initializes an application in the *NDDS* domain of your choice, and returns a handle to the new domain, for use in creating *NDDS* objects in that domain later on. **NddsInit()** must be called before you can create any publications or subscriptions.

Register the NDDSType

The generated function **HelloMsgNddsRegister()** registers the HelloMsg *NDDSType*.

Allocate space for the message

The generated function **HelloMsgAllocate()** allocates memory for an instance of the HelloMsg *NDDSType*, enough to accommodate the maximum array size specified by the **MAX_MSG_SIZE** constant specified in **Hello.x**.

The code:

```
instance = HelloMsgAllocate();
```

allocates memory for the HelloMsg structure and sets *instance* to it so you can give the pointer to *NDDS* to send issues. You change the issue value by modifying the content of this address, as shown in the next step.

Create a publication

The following creates a publication for the hello message:

```
publication = NddsPublicationCreate(domain, "Example HelloMsg",
                                    HelloMsgNDDSType, instance,
                                    persistence, strength);
```

Table 4.1 lists the arguments specified to create this publication.

Table 4.1 **Arguments to NddsPublicationCreate()**

Argument	Value	Description
domain	domain	Handle to the domain to which the publication will belong.
nddsTopic	"Example HelloMsg"	*NDDSTopic.*
nddsType	HelloMsgNDDSType	#defined as "HelloMsg" in Hello.h.
instance	address of HelloMsg structure	Pointer to HelloMsg structure containing the data to be published.
persistence	15 seconds	Refer to Section 2.3.1.
strength	1	Refer to Section 2.3.1.

Send the message.

2. You need to add one line to make the code work. Modify the message to be sent under the following line:

```
/* modify the data to be sent here */
```

by adding the following line:

```
sprintf(instance->msg, "Hello Universe! (%d)", count);
```

The *count* variable already exists and is incremented by the publication task every time it sends an issue with the call:

```
NddsPublicationSend(publication);
```

You produced the effect of sending issues at a fixed rate by sleeping for *send_period_sec* (4 seconds):

```
NddsUtilitySleep(send_period_sec).
```

Warning: Do not use **NddsUtilitySleep()** to conduct performance tests, such as throughput or latency tests. **NddsUtilitySleep()** cannot guarantee the resolution required in a performance test. Instead, use the for() loop or its equivalent, which has much finer resolution. For hints, see the performance test examples in the **examples/performance** directory.

3. Save your changes. The final code should be the same as that in the **<NDDSTutorialDir>/hello** directory.

4.2.3 Review Hello_subscriber.c

To review the subscription code for the "Hello World!" message:

1. Open **myhello/Hello_subscriber.c** and review the generated code. No changes are required to run this code.

   The generated code for **Hello_subscriber.c** parallels **Hello_publisher.c** in four ways:

   a. Sets the *NDDS* verbosity with **NddsVerbositySet()**.

   b. Initializes the application in an *NDDS* domain with **NddsInit()**.

   c. Registers the HelloMsg *NDDSType* with **HelloMsgNddsRegister()**.

   d. Allocates space for a data object of the type HelloMsg with **HelloMsgAllocate()**.

Create an immediate subscription

A subscription is created after the above four steps:

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

Table 4.2 lists the arguments specified to create this subscription.

Customize the callback routine (optional)

HelloMsgCallback() is currently set to print only the contents of the message.

2. (*Optional*) If you want to print a detailed message: Look in the **HelloMsgCallback()** routine, find the **#if 0** and the **#endif /* 0 */** lines, and remove them.

Sleep while waiting for issues

For an immediate subscription, there is nothing left to do, because *NDDS* will automatically invoke the callback when an issue is received.

3. Save your changes (if any). The final code should be the same as that in the **<NDDSTutorialDir>/hello** directory.

Table 4.2 **Arguments to NddsSubscriptionCreate()**

Argument	Value	Description
domain	domain	Handle to the subscription's domain.
mode	NDDS_SUBSCRIPTION_IMMEDIATE	This subscription will invoke the subscription callback routine as soon as a valid issue is received.
nddsTopic	"Example HelloMsg"	Must be the same as the *NDDSTopic* assigned to the publication.
nddsType	HelloMsgNDDSType	#defined as "HelloMsg" in Hello.h.
instance	Address of HelloMsg structure	Structure to be used as temporary data holder for the incoming issue.
deadline	10.0 seconds	Alert the application if no new data is received in 10 seconds. See Section 2.3.2.
minimum-Separation	0.0 seconds	The subscription will accept all issues from the publications no matter how fast they are published. See Section 2.3.2.
callBackRtn	HelloMsgCallback()	This routine is invoked when a valid issue arrives or the deadline occurs.
callback-RtnParam	NULL	User provided pointer to be parsed back to the callback routine.
multicast-Address	NDDS_USE_UNICAST	Receive issues using unicast. To learn about using multicast in *NDDS*, see Lesson 7.

4. The remaining steps in this lesson are covered in Chapter 2:

Refer to Chapter 2

- Build the Publication and Subscription Programs (Section 2.3.6)
- Run the Subscription Program (Section 2.3.7)
- Run the Publication Program (Section 2.3.8)
- Review the Screen Output on Each Side (Section 2.3.9)
- Experiment with the Programs (Section 2.3.10)

4.3 Lesson 3: Create a Polled Subscription

Goal Modify the subscription created in Lesson 2 to poll for issues from the publication created in Lesson 2. The final code should be the same as that in the **<NDDSTutorialDir>/polled** directory.

4.3.1 Edit the Subscription Source Code

Continue working in the **myhello** directory you used in the previous lessons.

There are three changes required in the subscription to poll for "Example HelloMsg":

- ❑ Change the subscription mode to **NDDS_SUBSCRIPTION_POLLED**.
- ❑ Increase the receive queue size to 8.
- ❑ Poll the subscription at a 10 second period.

Note: You will be polling slower than the send rate of the publication. Initially, the receive queue will cache the issues, but you will drop issues eventually. For the purpose of this exercise, this is fine.

To edit the Hello_subscriber.c code for polled mode:

1. Open **Hello_subscriber.c** for edit.
2. Below the NDDSSubscription declaration:

   ```
   NDDSSubscription subscription;
   ```

   Declare the structure to hold the new subscription properties information:

   ```
   NDDSSubscriptionProperties properties;
   ```

3. Change **NDDS_SUBSCRIPTION_IMMEDIATE** in **NddsSubscriptionCreate()** to **NDDS_SUBSCRIPTION_POLLED**.
4. After creating the subscription (below the **NddsSubscriptionCreate()** call), change the receive queue size by entering the following:

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

5. Uncomment the line:

```
/* NddsSubscriptionPoll(); */
```

   in the **while(1**) loop to poll the subscription at the *deadline* rate.

6. Save your changes. The final code should be the same as that in the **<NDDSTutorialDir>/polled** directory.

Note: Publication code is unaffected for the polled mode.

4.3.2 Build the Publication and Subscription Programs

See Section 2.3.6 for instructions on building the publication and subscription programs.

4.3.3 Run the Subscription and Publication Programs

There is no difference in output content from Lesson 2. But if you look closely, the subscription does not report new data right away, but in bursts every 10 seconds, which is the polling rate.

4.4 Lesson 4: Create a Publisher and a Subscriber

Goal Use a signalled publisher to send the "Hello World!" message to a subscription managed by a subscriber. The final code should be the same as in the **<NDDSTutorialDir>/publisherSubscriber** directory.

4.4.1 Edit Hello_publisher.c

Continue working in the **myhello** directory you used in the previous lessons.

To edit the Hello_publisher.c code to use a signalled publisher:

1. Open **Hello_publisher.c** for edit.
2. Add the **NDDSPublisher** declaration after the **NDDSPublication** declaration line:

```
NDDSPublication publication;
NDDSPublisher   publisher;
```

3. Create a signalled publisher *before* **NddsPublicationCreate()** by adding the following line:

```
publisher = NddsPublisherCreate(domain, NDDS_PUBLISHER_SIGNALLED);
```

4. *After* creating the publication and the publisher, add the publication to the publisher by adding the following line:

```
NddsPublisherPublicationAdd(publisher, publication);
```

5. Use the publisher to send issues. Change:

```
NddsPublicationSend(publication);
```

   to:

```
NddsPublisherSend(publisher);
```

6. Save your changes. The final code should be the same as in the **<NDDSTutorialDir>/publisherSubscriber** directory.

4.4.2 Edit Hello_subscriber.c

To edit the Hello_subscriber.c code to use a subscriber:

1. Open **Hello_subscriber.c** for edit.
2. Add the **NDDSSubscriber** declaration after the **NDDSSubscription** declaration line:

```
NDDSSubscription subscription;
NDDSSubscriber   subscriber;
```

3. Create a subscriber before calling **NddsSubscriptionCreate()**.

```
subscriber = NddsSubscriberCreate(domain);
```

4. *After* creating the subscription and changing its properties (*receiveQueueSize* in this case), add it to the subscriber. After the following lines:

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

   Add:

```
NddsSubscriberSubscriptionAdd(subscriber, subscription);
```

5. If the subscription were in the immediate mode, you could stop here. But since this is a polled subscription, you must now use the subscriber to poll.

   Change:

   ```
   NddsSubscriptionPoll(subscription);
   ```

   to:

   ```
   NddsSubscriberPoll(subscriber);
   ```

6. Save your changes. The final code should be the same as in the **<NDDSTutorialDir>/publisherSubscriber** directory.

4.4.3 Build the Publisher and Subscriber Programs

For instructions on how to build the publisher and subscriber programs, see Section 2.3.6.

4.4.4 Run the Subscription and Publication Programs

Using publishers and subscribers does not change the apparent behavior. The output should be the same as in Lesson 2.

4.5 Lesson 5: Create a Client and a Server

Goal

Create a client-server pair and implement a service that adds a pair of numbers. In this exercise, you will create a client that sends two numbers to a server, which will add them and return the result to the client. The final code should be the same as in the **<NDDSTutorialDir>/clientServer** directory.

4.5.1 Create the Request and Reply NDDS Types

To create the request and reply NDDS Types for the add service in C:

1. Create a new directory called **myclientServer/**.

2. In the new directory, create a file named **Add.x** which contains:

```
1   /* nddsgen.C.NDDSType RequestMsg; */
2   /* nddsgen.C.NDDSType ReplyMsg; */
3
4   struct RequestMsg {
5       int x;
6       int y;
7   };
8
9   struct ReplyMsg {
10      int sum;
11  };
```

Lines 1,2 As in Lesson 1, the *NDDSType* token identifies the structure that will be the basis of the *NDDS* type.

3. Type at the command prompt:

```
nddsgen Add.x -example <architecture>:client
```

Refer to Chapter 2 where **<*architecture*>** is the architecture switch for your system (see Table 2.1).

4. Verify that **nddsgen** generated **Add.h**, **Add.c**, **Add_client.c**, **Add_server.c**, and the following:
   - **makefile_Add_<*architecture*>**, if you specified a UNIX architecture**.**
   - **Add.dsw, Add_client.dsp,** and **Add_server.dsp**, if you specified a Windows architecture.

4.5.2 Edit Add_server.c

Create a server that adds two integers sent by the client, and then sends the results back to the client. Once created, the server sits in an infinite **while()** loop. The real action takes place in the server callback routine that you provide when you create the server.

Continue working in the **myClientServer** directory you created in Section 4.5.1.

To create a server for the add service:

1. Open **Add_server.c** and review the generated code.

   If you finished Lesson 2, you are familiar with the beginning part of the code, which:
   - Sets the verbosity level with NddsVerbositySet().
   - Initializes the application in an *NDDS* domain with NddsInit().

- Registers the request and reply *NDDSTypes*.
- Allocates space for the request and reply message holders.

Create a server

2. Before the **while()** loop in **serverMain()**, add the following lines to create a server for "Service1" with **NddsServerCreate()**:

```
NddsServerCreate(domain, "Service1", NDDS_SERVER_IMMEDIATE, strength,
                 ReplyMsgNDDSType, instanceReply, RequestMsgNDDSType,
                 instanceRequest, ServerRoutine, NULL);
```

Table 4.3 lists the arguments specified.

Table 4.3 **Arguments to NddsServerCreate()**

Argument	Value	Description
domain	domain	Handle to the server's domain.
serviceName	"Service1"	The name of the server to be invoked.
mode	NDDS_SERVER_IMMEDIATE	The ServerRoutine() will be invoked as soon as a valid request is received.
strength	1	Allows arbitration among multiple servers. The value does not matter if there is only one server for "Service1."
replyType	Reply-MsgNDDSType	#defined as "ReplyMsg" in Add.h
reply	instanceReply	Temporary holder for the outgoing reply to the client.
requestType	Request-MsgNDDSType	#defined as "RequestMsg" in Add.h
request	instanceRequest	Temporary holder for the incoming request from clients.
callBackRtn	ServerRoutine()	This routine is invoked by *NDDS* when a client request is received.
callBack-RtnParam	NULL	Pass any user data to the server callback routine with this pointer.

Once the server is created, there is nothing more to do in the main body of the server program but to wait for a request to come. The real action is in the server callback routine, **ServerRoutine()**, where the service is performed. In this exercise, add the two numbers in the request message.

3. In **ServerRoutine()**, find the line:

Edit the server callback routine

```
/* modify the data replied here */
```

and add below it:

```
itemReply->sum = itemRequest->x + itemRequest->y;
```

to add the two numbers in the request message.

Note: The server callback routine returns **RTI_TRUE** by default. If you return **RTI_FALSE**, no reply is sent to the client; the server will "veto" that request.

4. Save your changes. The final code should be the same as in the **<NDDSTutorialDir>/clientServer** directory.

4.5.3 Edit Add_client.c

You will create a client that will continuously send a pair of increasing integers. For each service request, it will wait at least 2.0 seconds and at most 5.0 seconds for the reply.

As illustrated in Figure 2.4, the minimum and maximum waits are the parameters governing the "best then first" semantics. If you ran multiple servers for "Service1" with one server stronger than the others, you would see that the client waits for 2.0 seconds and then comes back with the reply from the strongest server. If the client did not get a reply from any server within the 2.0 second wait, then it will report the first reply it gets from any server from that point on, until the deadline expires after 5.0 seconds.

To create a client for the *add* service:

1. Open **Add_client.c** and review the generated code.

   The beginning part of the code is the same as in the server:

   - Set the verbosity level with **NddsVerbositySet()**.
   - Initialize the application in an *NDDS* domain with **NddsInit()**.
   - Register the request and reply *NDDS* types.
   - Allocate space for the request and reply messages.

Create a client

2. Before calling **NddsClientServerWait()**, add the following lines to create a client for "Service1" with **NddsClientCreate()**:

```
client = NddsClientCreate(domain, "Service1",
                          ReplyMsgNDDSType, instanceReply,
                          RequestMsgNDDSType, instanceRequest);
```

Table 4.4 lists the arguments specified.

Table 4.4 **Arguments to NddsClientCreate()**

Argument	Value	Description
domain	domain	Handle to the client's domain.
serviceName	"Service1"	Must match the ServiceName for the server.
replyType	ReplyMsgNDDSType	#defined as "ReplyMsg" in Add.h
reply	instanceReply	Temporary holder for outgoing reply to the client.
requestType	Request-MsgNDDSType	#defined as "RequestMsg" in Add.h
request	instanceRequest	Temporary holder for incoming request from clients.

Wait for a server to appear (optional)

3. *(Optional)* Since clients and servers are anonymous, the clients do not know whether a server for the desired service exists. If there is in fact no server, or something prevents the client from receiving the server reply, the client would only find out after the maximum wait for the request expires. Optionally, you can ensure that a server exists with **NddsClientServerWait()**.

After creating the client and before sending the request, type:

```
if (!NddsClientServerWait(client, 2.0, 5, 1)) {
         printf("There is no server. Exiting...\n");
         return RTI_FALSE;
}
```

With this call, you tell the client to try to find at least one server, retrying up to 5 times and waiting 2.0 seconds between retries. If the client still can not find a server, you will need to exit the program.

Enter the request message

4. Since you already have the count variable which is incremented in the **for()** loop, you will need to send its value in the request message.

Below the line:

```
/* modify the data requested here */
```

Add the following lines:

```
instanceRequest->x = count;
instanceRequest->y = count;
```

Send the request and wait

When sending a request, clients can either:

- Block for a reply by calling **NddsClientCallAndWait()**.
- Go on with other tasks while passively waiting for the reply by calling **NddsClientCall()**.

Since you have nothing else to do (probably not true in your real application), block waiting for the reply for up to 5.0 seconds. Even if the reply comes back immediately, you will need to wait for at least 2.0 seconds.

When the reply comes back before the deadline, report the content of the reply with **ReplyMsgPrint()**, which is defined in **Add.h**.

5. Save your changes. The final code should be the same as in the **<NDDSTutorialDir>/clientServer** directory.

6. The remaining steps in this lesson are provided in:

   - Build the Client and Server Programs (Section 2.6.5)
   - Run the Client Only (Section 2.6.6)
   - Start the Server First and Then the Client (Section 2.6.7)

Refer to Chapter 2

Chapter 5

Advanced C Exercises

Read the Overview and perform the Basic Exercises before you attempt the Advanced Exercises. This chapter provides instructions on how to perform the same exercises presented in Chapter 3 if you are using C language.

These advanced lessons do not require you to make any code changes. You will be reviewing, and reading the explanation of, the supplied example code.

- ❏ Lesson 6: Publish and Subscribe Reliably (Section 5.1)
- ❏ Lesson 7: Publish and Subscribe Using Multicast (Section 5.2)
- ❏ Lesson 8: Subscribe Using Patterns (Section 5.3)

5.1 Lesson 6: Publish and Subscribe Reliably

Goal

Create a reliable publication-subscription pair to send and receive the "Hello World!" message of the HelloMsg *NDDS* type created in Lesson 1.

In this exercise, you will review the publication and the subscription code in the **<NDDSTutorialDir>/reliable** directory and discover the calls necessary to send issues reliably. You will learn that the code is similar to the unreliable publication and subscription created in Lesson 2. The *NDDS* type and the message content is exactly the same. There are no code changes for you to make in this lesson.

5.1.1 Learn How to Create a Reliable Publication

Open **Hello_publisher.c** in the **<NDDSTutorialDir>/reliable** directory. The main body of code is shown in Figure 5.1.

Figure 5.1 **Reliable Publication Code in C**

```
1   int publisherMain(int nddsDomain, int nddsVerbosity)
2   {
3       int          count           = 0;
4       RTINtpTime   send_period_sec = {0,0};
5       int          strength        = 1;
6       NDDSPublication publication;
7       NDDSPublicationProperties properties;
8       HelloMsg *instance = NULL;
9       NDDSDomain   domain;
10      RTINtpTime waitTime         = {0,0};
11
12      RtiNtpTimePackFromNanosec(send_period_sec, 4, 0);   /* 4 seconds */
13      RtiNtpTimePackFromNanosec(waitTime, 2, 0);        /* 2 seconds */
14
15      NddsVerbositySet(nddsVerbosity);
16      domain = NddsInit(nddsDomain, NULL, NULL);
17
18      HelloMsgNddsRegister();
19
20      printf("Allocate HelloMsg type.\n");
21      instance = HelloMsgAllocate();
22
23      NddsPublicationPropertiesDefaultGet(domain, &properties);
24      properties.sendQueueSize   = 5;
25      RtiNtpTimePackFromMillisec(properties.heartBeatTimeout, 0, 500);
26      RtiNtpTimePackFromNanosec(properties.sendMaxWait, 2, 0);
27      RtiNtpTimePackFromNanosec(properties.persistence, 15, 0);
28                                                          /* 15 seconds */
29      properties.strength        = strength;
30
31      publication = NddsPublicationCreateAtomic(domain,
32                                            "Reliable HelloMsg",
33                                             HelloMsgNDDSType, instance,
34                                             &properties, NULL);
35
36      if (NddsPublicationSubscriptionWait(publication, waitTime, 10, 1)
37                                                 != NDDS_WAIT_SUCCESS) {
38          printf("There is no subscription to the topic. \
39                  Might as well exit.\n");
40          return 0;
41      }
42
```

```
43      for (count=0;;count++) {
44          printf("Sampling publication, count %d\n", count);
45
46          /* modify the data to be sent here */
47          sprintf(instance->msg, "Hello World! (%d)", count);
48
49          if (NddsPublicationSend(publication) !=
50                                          NDDS_PUBLICATION_SUCCESS) {
51              printf("publication %d not sent\n", count);
52              return 0;
53          }
54          NddsUtilitySleep(send_period_sec);
55      }
56
57      return 1;
58  }
```

Line 7 We create an **NDDSPublicationProperties** structure on the stack to tailor the publication behavior for reliability.

Lines 12-13 These lines convert time values from seconds and nanoseconds into **RTINtpTime** structures. This is our operating system-independent time representation. It is based on the NTP time format. Time is not represented in a human readable form. Instead it is organized in a way that makes 64 bit arithmetic easy. We have provided a set of functions to convert to and from **RTINtpTime**. See Appendix A for details.

Lines 12-21 The steps to create a reliable publication are identical to the best-effort publication, up to the point of initializing *NDDS*, registering the *NDDS* Type, and allocating a publication instance.

Lines 23-29 Tailor reliable behavior of the publication through publication properties.

When a publication has reliable subscriptions, it is functionally different than a best-effort (unreliable) publication in the following ways:

- ❏ It checks for acknowledgements from the reliable subscriptions, and resends previously published issues if a subscription reports that issue(s) as missing. The *sendQueueSize* field of the publication's property specifies the maximum number of issues the publication remembers.
- ❏ In addition to passively checking the acknowledgement messages from the reliable subscriptions, the publication can (optionally) actively poll for the remote subscription's receipt status periodically. We set the *heartBeatTimeout* field to 0.5 second to poll for the subscription status twice a second. This fancy feature is not necessary for this simple example, but we show it here in case you want to use it later in your application.

- ❏ If a subscription falls behind despite the publication's efforts, the publication's queue will eventually fill up, and the send call will block for the *sendMaxWait* seconds, and then return error. We want to block only a finite time: 2.0 seconds here.
- ❏ We specify the strength and persistence through the properties rather than the creation API because we use the atomic creation API, as shown below.

Lines31-34 We created the publication with the atomic creation API because we want the publication to start out with the desired properties from the outset. In addition to the properties, the atomic creation API takes a publication listener as the last argument. Since we do not want to be notified of any publication-related events, we simply pass NULL in this example.

Atomically created *NDDS* objects behave as you want from the very beginning and do not take additional time to change the properties afterwards. Changing an object property after the initial creation can be an expensive operation, depending on the property being changed. Queue size is one such field, and the property change may fail for a variety of reasons, including running out of memory. In such cases, the object is in an inconsistent or unusable (at least not exactly the way you wanted) state. Strictly speaking, you have to kill that object if the property change failed, wasting the effort invested to create that object.

Lines 36-41 Even though a reliable publication is allowed to send issues immediately after creation, there is no guarantee that a subscription exists to receive the issues. If you start publishing anyway without ensuring the existence of a subscription, it defeats the purpose of using reliable subscriptions somewhat. **NddsPublicationSubscriptionWait()** is a convenience function for ensuring the existence of a specified number of subscriptions to the publication's *NDDS* topic. It is also a deterministic call because you will wait no more than *waitTime* (2.0 seconds) times *retries* (10 times), which is 20 seconds in this exercise.

Lines 49-53 If the send queue becomes full, a reliable publication blocks waiting for a free space, up to the *sendMaxWait* time you specified, at the end of which the send call returns an error. You are free to ignore it and simply note that the issue was not sent. Here, we take the most dramatic action of quitting the program. You can also wait for the queue level to drop to a value you specify with **NddsPublicationWait()**, and ensure that all published issues have been acknowledged by the reliable subscriptions.

5.1.2 Learn How to Create a Reliable Subscription

Open **Hello_subscriber.c** in the **<NDDSTutorialDir>/reliable** directory. The main body of code is shown in Figure 5.2.

Figure 5.2 **Reliable Subscription Code in C**

```
1   int subscriberMain(int nddsDomain, int nddsVerbosity)
2   {
3       NDDSSubscription subscription;
4       NDDSSubscriptionProperties properties;
5       HelloMsg    *instance = NULL;
6       NDDSDomain       domain;
7       NDDSIssueListener issueListener;
8       char             deadlineString[RTI_NTP_TIME_STRING_LEN];
9
10      NddsVerbositySet(nddsVerbosity);
11      domain = NddsInit(nddsDomain, NULL, NULL);
12
13      HelloMsgNddsRegister();
14
15      printf("Allocate HelloMsg type.\n");
16      instance = HelloMsgAllocate();
17
18      NddsSubscriptionPropertiesDefaultGet(domain, &properties);
19      properties.receiveQueueSize = 5;
20      properties.mode = NDDS_SUBSCRIPTION_IMMEDIATE;
21      RtiNtpTimePackFromNanosec(properties.deadline, 10, 0);
22
23      NddsSubscriptionIssueListenerDefaultGet(&issueListener);
24      issueListener.recvCallBackRtn = HelloMsgCallback;
25
26      subscription =
27          NddsSubscriptionReliableCreateAtomic(domain, "Reliable HelloMsg",
28                                               HelloMsgNDDSType, instance,
29                                               &properties, &issueListener,
30                                               NULL, NDDS_USE_UNICAST);
31
32      while (1) {
33          /* Only needed if NDDS_SUBSCRIPTION_POLLED */
34          /* NddsSubscriptionPoll(subscription); */
35
36          /*
37           * We sleep only to kill time.  Nothing need be done here
38           *  for an NDDS_SUBSCRIPTION_IMMEDIATE subscription.
39           */
40          printf("Sleeping for %s sec...\n",
41                 RtiNtpTimeToString(properties.deadline, deadlineString));
42          NddsUtilitySleep(properties.deadline);
43      }
44      return 1;
45  }
```

Line 4 We create an **NDDSSubscriptionProperties** structure on the stack to tailor the subscription behavior for reliability.

Lines 10-16 The steps to create a reliable subscription are identical to the best-effort subscription, up to the point of initializing *NDDS*, registering the *NDDS* type, and allocating a subscription instance.

Lines 18-21 Tailor reliable behavior of the subscription through subscription properties.

A reliable subscription is different than a best-effort subscription in the following ways:

- ❏ It checks the sequence number of the received issues and puts them in order before handing them off to the application. Upon discovery of lost issue, it reserves space for the missing issue and requests retransmission. The *receiveQueueSize* field of the subscription's property specifies the maximum number of issues the subscription can cache while it retries lost issues.
- ❏ You cannot specify minimum separation to a reliable subscription. If you want to receive issues in-order, you must accept all issues. Hence we specify the deadline but not the minimum separation in properties.
- ❏ We specify the mode and the deadline through the properties rather than passing them directly to the creation API because we use the atomic creation API, as shown below. The creation API itself is different than for the best-effort subscriptions: namely, it uses the keyword "Reliable" as part of the function name.

Lines 23-30 We create the reliable subscription with the atomic creation API for the same reasons we had used the atomic API for the publication creation. In addition to the properties and issue listener, the atomic creation API takes the reliable subscription listener as the second-to-last argument. Since we do not want to be notified of any reliable subscription-related events, we simply pass NULL in this example. Note that we are still specifying an issue listener, as we must to create any subscription.

5.1.3 Build the Reliable Subscription and Publication Programs

To build the reliable subscription and publication programs:

1. Copy the contents of the **<NDDSTutorialDir>/reliable** directory into a new directory of your own, such as **myreliable**.
2. Follow the build steps listed in Section 2.3.6, but use your new **myreliable** directory.

5.1.4 Run the Reliable Subscription and Publication Programs

To run the reliable subscription and publication programs:

1. Start the publication first. Since there is no subscription, the publication should wait for 20 seconds then quit.
2. Start the subscription.
3. Start the publication again.

Note: The output from the reliable subscription is the same as what you saw for the best-effort publication in Lesson 2. The *NDDS* reliability mechanism is transparent to the application.

4. Now kill the subscription with <**Ctrl-C**>. The publication keeps publishing until its send queue fill up, blocks for 2 seconds, and finally quits the program because send call returned error due to the send call timed out on the full queue.

5.2 Lesson 7: Publish and Subscribe Using Multicast

Goal

Create a multicast publication-subscription pair to send and receive the "Hello World!" message of the HelloMsg *NDDS* type created in Lesson 1.

In this exercise, you will review the publication and the subscription code in the **<NDDSTutorialDir>/multicast** directory and discover the calls necessary to send and receive issues on a multicast address. You will learn that the code is very similar to the unicast publication and subscription created in Lesson 2. The *NDDS* type and the message content will be exactly same.

There are no code changes for you to make in this lesson.

5.2.1 Learn How to Create a Multicast Publication

Open **Hello_publisher.c** in the **<NDDSTutorialDir>/multicast** directory. The main body of code is shown in Figure 5.3.

Figure 5.3 **Multicast Publication Code in C**

```
1   int publisherMain(int nddsDomain, int nddsVerbosity)
2   {
3       int          count           = 0;
4       RTINtpTime   send_period_sec = {0,0};
5       RTINtpTime   persistence     = {0,0};
6       int          strength        = 1;
7       NDDSPublication publication;
8       HelloMsg *instance = NULL;
9       NDDSDomain   domain;
10      NDDSDomainProperties domainProperties;
11
12      RtiNtpTimePackFromNanosec(send_period_sec, 4, 0);   /* 4 seconds */
13      RtiNtpTimePackFromNanosec(persistence, 15, 0);      /* 15 seconds */
14
15      NddsVerbositySet(nddsVerbosity);
16
17      NddsDomainPropertiesDefaultGet(&domainProperties);
18      domainProperties.multicast.enabled = RTI_TRUE;
19      domainProperties.multicast.ttl     = NDDSTTLSameSubnet;
20      domain = NddsInit(nddsDomain, &domainProperties, NULL);
21
22      HelloMsgNddsRegister();
23
24      printf("Allocate HelloMsg type.\n");
25      instance = HelloMsgAllocate();
26
27      publication = NddsPublicationCreate(domain, "Multicast HelloMsg",
28                                          HelloMsgNDDSType, instance,
29                                          persistence, strength);
30
31      for (count=0;;count++) {
32          printf("Sampling publication, count %d\n", count);
33
34          /* modify the data to be sent here */
35          sprintf(instance->msg, "Hello World! (%d)", count);
36
37          NddsPublicationSend(publication);
38          NddsUtilitySleep(send_period_sec);
39      }
40
41      return 1;
42  }
```

Line 9 This code creates an **NDDSDomainProperties** structure variable to enable multicasting and set the TTL.

Lines 17-19 First copy the default application properties into the **NDDSDomainProperties** structure before changing the fields you are interested in. In this case, we are only interested in multicast properties.

In this exercise, multicasting is enabled and the TTL is set to restrict the messages to within the same subnet. The application is finally initialized with these multicast properties during **NddsInit()**.

Lines 20-42 The rest of the code is the same as the unicast publication in Lesson 2 (Section 4.2).

5.2.2 Learn How to Create a Multicast Subscription

Open **Hello_subscriber.c** in the **<NDDSTutorialDir>/multicast** directory. The main body of code is shown in Figure 5.4.

Figure 5.4 **Multicast Subscription Code in C**

```
1   int subscriberMain(int nddsDomain, int nddsVerbosity)
2   {
3       RTINtpTime       deadline       = {0,0};
4       RTINtpTime       min_separation = {0,0};
5       NDDSSubscription subscription;
6       HelloMsg        *instance = NULL;
7       NDDSDomain       domain;
8       char             deadlineString[RTI_NTP_TIME_STRING_LEN];
9       NDDSDomainProperties domainProperties;
10
11      RtiNtpTimePackFromNanosec(deadline, 10, 0);
12      RtiNtpTimePackFromNanosec(min_separation, 0, 0);
13
14      NddsVerbositySet(nddsVerbosity);
15
16      NddsDomainPropertiesDefaultGet(&domainProperties);
17      domainProperties.multicast.enabled = RTI_TRUE;
18      domainProperties.multicast.ttl     = NDDSTTLSameSubnet;
19      domain = NddsInit(nddsDomain, &domainProperties, NULL);
20      HelloMsgNddsRegister();
21
22      printf("Allocate HelloMsg type.\n");
23
24      instance = HelloMsgAllocate();
25      subscription = NddsSubscriptionCreate(domain,
26                NDDS_SUBSCRIPTION_IMMEDIATE,"Multicast HelloMsg",
27                HelloMsgNDDSType, instance,deadline, min_separation,
28                HelloMsgCallback, NULL,NddsStringToAddress("225.0.0.1"));
```

```
29
30      while (1) {
31          /* Only needed if NDDS_SUBSCRIPTION_POLLED */
32          /* NddsSubscriptionPoll(subscription); */
33
34          /*
35           * We sleep only to kill time.  Nothing need be done here
36           *  for an NDDS_SUBSCRIPTION_IMMEDIATE subscription.
37           */
38          printf("Sleeping for %s sec...\n",
39                 RtiNtpTimeToString(deadline, deadlineString));
40          NddsUtilitySleep(deadline);
41      }
42      return 1;
43  }
```

Line 9 As in the publication code, this code creates an **NDDSDomainProperties** structure variable to enable multicasting and set the TTL.

Lines 16-18 Enabling multicasting and setting the TTL is the same as for the publication.

Lines 25-28 The only difference between a unicast subscription and a multicast subscription is in the last argument to the create call. For the unicast subscription in Lesson 2 (Section 4.2), you just passed in **NDDS_USE_UNICAST** to indicate a unicast subscription. This time, specify the host byte order integer for 225.0.0.1 with **NddsStringToAddress()**, which converts a string form of an IP address into a host byte ordered address.

5.2.3 Build the Multicast Subscription and Publication Programs

To build the multicast subscription and publication programs:

1. Copy the contents of the **<NDDSTutorialDir>/multicast** directory into a new directory of your own, such as **mymulticast**.
2. Follow the build steps listed in Section 2.3.6, but use your new **mymulticast** directory.

5.2.4 Run the Multicast Subscription and Publication Programs

To run the multicast subscription and publication programs:

1. Start the subscription.
2. Start the publication.

Note: The output from the multicast subscription is the same as what you saw for the best-effort publication in Lesson 2. The *NDDS* multicast mechanism is transparent to the application.

5.3 Lesson 8: Subscribe Using Patterns

Goal

In this exercise, you will review the publication and the subscription code in the **<NDDSTutorialDir>/pattern** directory and discover the calls necessary to implement the pattern subscription strategy shown in Figure 3.5.

5.3.1 Review the Deposition Monitor Code

Open **depMonitor.c** in the **<NDDSTutorialDir>/pattern** directory. The main body of the code, provided in Figure 5.5, shows how to subscribe to the pattern *NDDS* topic "DepMod/*/*".

Figure 5.5 **Subscribing to Patterns in C**

```
1   NDDSSubscription SubscriptionPatternCreate(const char *nddsTopic,
2                                              const char *nddsType,
3                                              void *callBackRtnParam)
4   {
5       Data      *genericData    = NULL;
6       RTINtpTime deadline       = {10,0};
7       RTINtpTime min_separation = {0,0};
8
9       genericData = DataAllocate();
10
11      return NddsSubscriptionCreate(NULL, NDDS_SUBSCRIPTION_IMMEDIATE,
12                                    nddsTopic, nddsType, genericData,
13                                    deadline, min_separation,
14                                    DataCallback, NULL, NDDS_USE_UNICAST);
15  }
16
17  int DepositionMonitor(int nddsDomain, int nddsVerbosity)
18  {
19      RTINtpTime     sleepTime   = {10,0};
20      NDDSSubscriber depMonitor;
21      NDDSDomain     domain;
22      char           sleepString[RTI_NTP_TIME_STRING_LEN];
23
```

```
24      NddsVerbositySet(nddsVerbosity);
25      domain = NddsInit(nddsDomain, NULL, NULL);
26      DataNddsRegister();
27
28      depMonitor = NddsSubscriberCreate(domain);
29      NddsSubscriberPatternAdd(depMonitor, "DepMod/*/*", DataNDDSType,
30                               SubscriptionPatternCreate, NULL);
31
32      while (1) {
33          /* NddsSubscriptionPoll(subscription);
34             Only needed if NDDS_SUBSCRIBER_POLLED*/
35
36          /*
37           * We sleep only to kill time.  Nothing need be done here
38           *  for an NDDS_SUBSCRIBER_IMMEDIATE subscription.
39           */
40          printf("Sleeping for %s sec...\n",
41                 RtiNtpTimeToString(sleepTime, sleepString));
42          NddsUtilitySleep(sleepTime);
43      }
44      return 1;
45  }
```

Line 28 From Lesson 4, you learned that a subscriber manages subscriptions. Since the pattern subscription feature involves creating multiple subscriptions for each match, you specify the desired pattern *NDDS* topic and *NDDS* types to a subscriber, which can create a subscription for the match. Therefore, you need to create a subscriber first.

Lines 29-30 **NddsSubscriberPatternAdd()** tells the subscriber (**depMonitor**) what pattern topic ("**DepMod/*/***") and pattern type ("**Data**") to match, and if there is a match, what routine to use to create a subscription for the match (**SubscriptionPatternCreate()**). The last argument is an optional user parameter, which can be used to pass any variable you want to present to the subscription create routine. In this exercise, you use it to pass the pointer to the temporary holder for the Data (**genericData**) so that you can use the same instance to hold the incoming issues for all matching publications.

Lines 1-15 You told *NDDS* to invoke **SubscriptionPatternCreate()** when it discovers a publication matching the pattern topic/type. With **SubscriptionPatternCreate()**, you have the flexibility to:

- Create a subscription for all matches, as you are doing here. Note that all subscriptions will use the same deadline, minimum separation, callback routine, but this is purely for simplification purposes. Each subscription can have different parameters.

- ❑ Do other things before returning the subscription to *NDDS*. The user parameter is convenient for passing information in either direction.

5.3.2 Review the Safety Supervisor Code

Open **supervisor.c** in the **<NDDSTutorialDir>/pattern** directory. Even though the safety supervisor publishes as well as subscribes, you will examine only the pattern subscription part in this exercise. You already learned how to subscribe to a pattern in the deposition monitor program. The safety supervisor code in Figure 5.6 shows that you can subscribe to *multiple* pattern topics with one subscriber: in this example, the Safety Supervisor subscribes to the patterns "*/*/Pressure" and "*/*/Temperature".

Figure 5.6 **Subscribing to Multiple Patterns in C**

```
1   safetySubscriber = NddsSubscriberCreate(NULL);
2   NddsSubscriberPatternAdd(safetySubscriber,
3                            "*/*/Pressure", DataNDDSType,
4                            SubscriptionPatternCreate, NULL);
5   NddsSubscriberPatternAdd(safetySubscriber,
6                            "*/*/Temperature", DataNDDSType,
7                            SubscriptionPatternCreate, NULL);
```

5.3.3 Review the Deposition Module

Review the code in **depModule.c** in the **<NDDSTutorialDir>/pattern** directory. The deposition module creates six publications with a hierarchical naming scheme. If you need a refresher on the concept of creating publications, see Lesson 2.

5.3.4 Review the Thermal Processing Module Code

Review the code in **thermProcess.c** in the **<NDDSTutorialDir>/pattern** directory. The thermal processing module creates three publications with a hierarchical naming scheme, and creates a reliable subscription to the command. If you need a refresher on the concept of creating publications and subscriptions, see Lesson 2.

5.4 Congratulations!

You can now develop any distributed real-time application. Your next steps are:

- ❑ Develop your application. See Chapter 3 in the *NDDS User's Manual* for guidance.
- ❑ Refer to the other advanced examples in the **examples/** directory. To test *NDDS*'s performance on your system, work in the **examples/performance/** directory.
- ❑ Consult with RTI on using *NDDS* to meet your design goals. For consulting rates and scheduling, contact your sales representative or send e-mail to RTI at **info@rti.com**.

Index