RTI Message Service

Getting Started Guide

Version 5.2.3

Your systems. Working as one.

Technical Support
Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089

Phone:	(408) 990-7444
Email:	support@rti.com
Website:	https://support.rti.com/

Contents

4 Installing a Database (Optional)

Chapter 1 Welcome to RTI Message Service

Welcome to *RTI® Message Service*, the highest-performing JMS-compliant messaging system in the world. *RTI Message Service* makes it easy to develop, deploy and maintain distributed applications. Its core messaging technology has been proven in hundreds of unique designs for life- and mission-critical applications across a variety of industries, providing

- ❏ ultra-low latency and extremely high throughput
- ❏ with industry-leading latency determinism
- ❏ across heterogeneous systems spanning thousands of applications.

Its extensive set of real-time quality-of-service parameters allows you to fine-tune your application to meet a wide range of timeliness, reliability, fault-tolerance, and resource usage-related goals.

This chapter introduces the basic concepts within the middleware and summarizes how *Message Service* addresses the needs of high-performance systems. It also describes the documentation resources available to you and provides a road map for navigating them. Specifically, this chapter includes:

1.1 Benefits of RTI Message Service

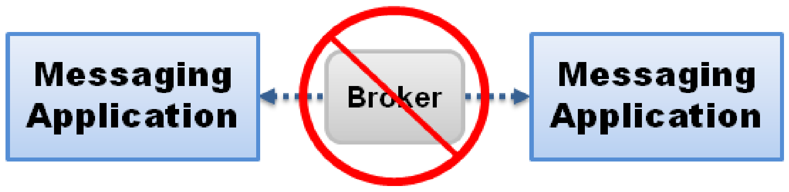
Message Service is publish/subscribe networking middleware for high-performance distributed applications. It implements the Java Message Service (JMS) specification, but it is not just another MOM (message-oriented middleware). Its unique peer-to-peer architecture and targeted high-performance and real-time capabilities extend the specification to provide unmatched value.

1.1.1 Reduced Risk Through Industry-Leading Performance and Availability

Message Service provides industry-leading performance, whether measured in terms of latency, throughput, or real-time determinism. One contributor to this superior performance is RTI's unique architecture, which is entirely peer-to-peer.

Traditional messaging middleware implementations require dedicated servers to broker message flows, crippling application performance, increasing latency, and introducing time nondeterminism. These brokers increase system administration costs and can represent single points of failure within a distributed application, putting data reliability and availability at risk.

RTI eliminates broker overhead by allowing messages to flow directly from a publisher to each of its subscribers in a strictly peer-to-peer fashion. At the same time, it provides a variety of powerful capabilities to ensure high availability.



Traditional message-oriented middleware implementations require a broker to forward every message, increasing latency and decreasing determinism and fault tolerance. RTI's unique peer-to-peer architecture eliminates bottlenecks and single points of failure.

Redundancy and high availability can optionally be layered onto the peer-to-peer data fabric by transparently inserting instances of *RTI Persistence Service*. These instances can distribute the load across topics and can also be arbitrarily redundant to provide the level of data availability your application requires. See Chapter 7, "Scalable High-Performance Applications: Durability and Persistence for High Availability," in the User's Manual for more information about this capability.

Publishers and subscribers can enter and leave the network at any time, and the middleware will connect and disconnect them automatically. *Message Service* provides fine-grained control over fail-over among publishers, as well as detailed status notifications to allow applications to detect missed delivery deadlines, dropped connections, and other potential failure conditions. See Chapter 6, "Fault Tolerance," in the Configuration and Operation Manual for more information about these capabilities.

1.1.2 Reduced Cost through Ease of Use and Simplified Deployment

- ❑ **Increased developer productivity**—Easy-to-use, well-understood JMS APIs get developers productive quickly. (Take an opportunity to go through the tutorial in the *Getting Started Guide* if you haven't already.) Outside of the product documentation itself, a wide array of third-party JMS resources exist on the web and on the shelves of your local book store.
- ❑ **Simplified deployment**—Because *Message Service* consists only of dynamic libraries, you don't need to configure or manage server machines or processes. That translates into faster turnaround and lower overhead for your team.
- ❑ **Reduced hardware costs**—Some traditional messaging products require you to purchase specialized acceleration hardware in order to achieve high performance. The extreme efficiency and reduced overhead of RTI's implementation, on the other hand, allows you to see strong performance even on commodity hardware.

1.1.3 Unmatched Power and Flexibility to Meet Unique Requirements

When you need it, RTI provides a high degree of fine-grained, low-level control over the operation of the middleware, including, but not limited to:

- ❑ The volume of meta-traffic sent to assure reliability.
- ❑ The frequencies and timeouts associated with all events within the middleware.
- ❑ The amount of memory consumed, including the policies under which additional memory may be allocated by the middleware.

These quality-of-service (QoS) policies can be specified in configuration files so that they can be tested and validated independently of the application logic. When they are not specified, the middleware will use default values chosen to provide good performance for a wide range of applications.

For specific information about the parameters available to you, consult the Configuration and Operation Manual.

1.1.4 Interoperability with OMG Data Distribution Service-Based Systems

The Data Distribution Service (DDS) specification from the Object Management Group (OMG) has become the standard for real-time data distribution and publish/subscribe messaging for high performance real-time systems, especially in the aerospace and defense industries. *Message Service* is the only JMS implementation to directly interoperate at the wire-protocol level with *RTI Connext™ DDS*, the leading DDS implementation.

Connext DDS is available not only in Java but also in several other managed and unmanaged languages. It is supported on a wide variety of platforms, including embedded hardware running real-time operating systems. For more information, consult your RTI account representative. If you are already using *Connext DDS* and are interested in DDS/JMS interoperability, consult the Interoperability Guide that accompanies this documentation.

1.2 Features of RTI Message Service

Under the hood, *Message Service* goes beyond the basic JMS publish-subscribe model to target the needs of applications with high-performance, real-time, and/or low-overhead requirements and provide the following:

- ❑ **Peer-to-peer publish-subscribe communications** Simplifies distributed application programming and provides time-critical data flow with minimal latency.
  - Clear semantics for managing multiple sources of the same data.
  - Efficient data transfer, customizable Quality of Service, and error notification.
  - Guaranteed periodic messages, with minimum and maximum rates set by subscriptions, including notifications when applications fail to meet their deadlines.
  - Synchronous or asynchronous message delivery to allow applications control over the degree of concurrency.
  - Ability to send the same message to multiple subscribers efficiently, including support for reliable multicast with customizable levels of positive and negative message acknowledgement.
- ❑ **Reliable messaging**—Enables subscribing applications to not only specify reliable delivery of messages, but to customize the degree of reliability required. Data flows can be configured for (1) guaranteed delivery at any cost, at one extreme, (2) the lowest possible latency and highest possible determinism, even if it means that some messages will be lost, at the other extreme, or (3) many points in between.

❑ **Multiple communication networks**—Multiple independent communication networks (*domains*), each using *Message Service,* can be used over the same physical network to isolate unrelated systems and subsystems. Individual applications can be configured to participate in one or multiple domains.

❑ **Symmetric architecture**—Makes your application robust:

- No central server or privileged nodes, so the system is robust to application and/or node failures.
- Topics, subscriptions, and publications can be dynamically added and removed from the system at any time.

Multiple network transports—*Message Service* includes support for UDP/IP (v4 and v6)—including, for example, Ethernet, wireless, and Infiniband networks—and shared memory transports. It also includes the ability to dynamically plug in support for additional network transports and route messages over them. It can optionally be configured to operate over a variety of transport mechanisms, including backplanes, switched fabrics, and other networking technologies.

Multi-platform and heterogeneous system support—Applications based on *Message Service* can communicate transparently with each other regardless of the underlying operating system or hardware. Consult the Release Notes to see which platforms are supported in this release.

Vendor neutrality and standards compliance—The *Message Service* API complies with the JMS specification. Unlike other JMS implementations, it also supports a wire protocol that is open and standards-based: the Real-Time Publish/Subscribe (RTPS) protocol specification from the Object Management Group (OMG), which extends the International Engineering Consortium's (IEC's) publicly available RTPS specification. This protocol also enables interoperability between *Message Service* and *Connext DDS* and between various DDS implementations. See Interoperability with OMG Data Distribution Service-Based Systems (Section 1.1.4).

1.3 JMS Conformance

Message Service is a high-performance messaging platform for demanding applications, including applications with real-time requirements. Not all portions of the JMS specification are relevant or appropriate for this domain, and some required features are not included in the specification. For more information about JMS conformance, including both limitations and significant extensions, see Appendix A, "JMS Conformance," in the User's Manual.

1.4 Understanding and Navigating the Documentation

To get you from your download to running software as quickly as possible, we have divided this documentation into several parts.

❑ Release Notes—Provides system-level requirements and other platform-specific information about the product. *Those responsible for installing Message Service should read this document first.*

❑ Getting Started Guide—Describes how to download and install *Message Service*. It also lays out the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application. *Developers should read this document first.*

- ❑ User's Manual—Describes the features of the product, their purpose and value, and how to use them. It is aimed at developers who are responsible for implementing the functional requirements of a distributed system, and is organized around the structure of the JMS APIs and certain common high-level scenarios.
- ❑ Configuration and Operation Manual—Provides lower-level, more in-depth configuration information and focuses on system-level concerns. It is aimed at engineers who are responsible for configuring, optimizing, and administering *Message Service*-based distributed systems.

Many readers will also want to consult additional documentation available online. In particular, RTI recommends the following:

- ❑ **RTI Customer Portal**—http://www.rti.com/support. Select the **Find Solution** link to see sample code, general information on *Message Service*, performance information, troubleshooting tips, and other technical details.
- ❑ **RTI Example Performance Test**—This recommended download includes example code and configuration files for testing and optimizing the performance of a simple *Message Service*-based application on your system. The program will test both throughput and latency under a wide variety of middleware configurations. It also includes documentation on tuning the middleware and the underlying operating system.

  To download this test, first log into your self-service support portal as described above. Click **Find Solution** in the menu bar at the top of the page then click **Performance** under **All Solutions** in the resulting page. Finally, click on or search for **Example Performance Test** to download the test.

  You can also review the data from several performance benchmarks here: http://www.rti.com/products/jms/latency-throughput-benchmarks.html.
- ❑ **Java Message Service (JMS) API Documentation**—*Message Service* APIs are compliant with the JMS specification. This specification is a part of the broader Java Enterprise Edition (Java EE) product from Sun Microsystems; Java EE 5 is documented at http://java.sun.com/javaee/5/docs/api/. In particular, see the **javax.jms** package.
- ❑ **Java Standard Edition API Documentation**—Java EE is an extension to, and relies on types imported from, the Java Standard Edition (Java SE) product. Java SE 6 is documented online at http://java.sun.com/javase/6/docs/api/.
- ❑ **Whitepapers and other articles** are available from http://www.rti.com/resources/.

1.5 Paths Mentioned in Documentation

The documentation refers to:

- ❑ **<RTIJMSHOME>**

  This refers to the installation directory for *Message Service*.

  Wherever you see <RTIJMSHOME> used in a path, replace it with your installation path.

  The default installation paths are:

  - Mac OS X systems:

    /Applications/rti_connext_dds-*version*

- UNIX-based systems, non-*root* user:

  /home/*your user name*/rti_connext_dds-*version*

- UNIX-based systems, *root* user:

  /opt/rti_connext_dds-*version*

- Windows systems, user without Administrator privileges:

  <your home directory>*\rti_connext_dds-*version

- Windows systems, user with Administrator privileges:

  C:\Program Files\rti_connext_dds-*version* (for 64-bits machines) or
  C:\Program Files (x86)\rti_connext_dds-*version* (for 32-bit machines)

You may also see $RTIJMSHOME or %RTIJMSHOME%, which refers to an environment variable set to the installation path.

❑ RTI Workspace directory, **rti_workspace**

The RTI Workspace is where all configuration files for the applications and example files are located. All configuration files and examples are copied here the first time you run *RTI Launcher* or any script in **<RTIJMSHOME>/bin**. The default path to the RTI Workspace directory is:

- Mac OS X systems:

  /Users/*your user name*/rti_workspace

- UNIX-based systems:

  /home/*your user name*/rti_workspace

- Windows systems:

  ***your Windows documents folder*\rti_workspace**

  Note: '*your Windows documents folder*' depends on your version of Windows.
  For example, on Windows 7, the folder is **C:\Users\your user name\Documents**; on Windows Server 2003, the folder is **C:\Documents and Settings\your user name\Documents**.

You can specify a different location for the **rti_workspace** directory. See the *RTI Connext DDS Core Libraries Getting Started Guide* for instructions.

❑ **<path to examples>**

Examples are copied into your home directory the first time you run *RTI Launcher* or any script in **<RTIJMSHOME>/bin**. This document refers to the location of these examples as **<path to examples>.** Wherever you see <path to examples>, replace it with the appropriate path.

By default, the examples are copied to **rti_workspace/*version*/examples**

So the paths are:

- Mac OS X systems:

  /Users/*your user name*/rti_workspace/*version*/examples

- UNIX-based systems:

  /home/*your user name*/rti_workspace/*version*/examples

- Windows systems:

  ***your Windows documents folder*\rti_workspace\version\examples**

Note: '*your Windows documents folder*' is described above.

You can specify that you do not want the examples copied to the workspace. See Controlling Location for RTI Workspace and Copying of Examples (Section 1.6).

1.6 Controlling Location for RTI Workspace and Copying of Examples

Here's how to specify a different location for **rti_workspace** and/or disable copying the examples into the workspace.

UNIX-based systems:

- ❑ To configure the behavior for all users, edit **<RTIJMSHOME>/resource/scripts/rticommon_config.sh**.
- ❑ To configure the behavior for the current user (takes precedence), create **$HOME/.rti/rticommon_config.sh** as follows:

```
# Customize RTI Workspace Directory and Copy Behavior
#####################################################
# copy_workspace=false
# workspace_dir=$HOME/rti_workspace
```

Uncomment the line(s) that you want to use (remove the #).

Windows systems:

- ❑ To configure the behavior for all users, edit **<RTIJMSHOME>\resource\scripts\rticommon_config.sh**.
- ❑ To configure the behavior for the current user (takes precedence), create **<your home directory>\.rti\rticommon_config.bat** as follows:

```
@REM Customize RTI Workspace Directory and Copy Behavior
@REM ###################################################
@REM set copyWorkspace=false
@REM set "workspaceDir=C:\Users\%USERNAME%\Documents\rti_workspace"
```

Uncomment the line(s) that you want to use (delete "@REM" at the start of the line).

Chapter 2 Tutorial

This chapter will teach you, with hands-on examples, how to write your first *Message Service* applications. It's divided into two parts:

- ❏ A simple "Hello, World" example gets you compiling and running your code as quickly as possible. It does not discuss or configure quality of service (QoS) parameters; it simply publishes and subscribes to simple text messages.
- ❏ A more-complex financial market data tutorial introduces you applications with richer functionality. It includes multiple publishers and subscribers and uses a number of QoS parameters.

RTI recommends that you complete both of these tutorials. The "Hello, World" tutorial should be completed before reading further in the product documentation in order to get you minimally acquainted with the product and its use. You may complete the market data tutorial either immediately following the "Hello, World" tutorial or later, at your convenience.

You may notice that these two examples use different methods to receive messages:

- ❏ The "Hello, World" example uses the synchronous **receive()** method of the *MessageConsumer* in a loop. This method is appropriate in cases, like this one, where simplicity is the overriding goal. However, because it requires memory allocation and a context switch, this method is not appropriate for applications with the most demanding performance requirements.
- ❏ The market data example uses a *MessageListener* to receive messages asynchronously. This mechanism is more complex, because it involves multiple threads. However, it requires no critical-path memory allocation, and the message is delivered in the same thread from which it was read from the network, so it provides higher performance than the synchronous method used above for Hello, World.

If you encounter problems building or running the applications you create in these tutorials, please consult Chapter 3: Building and Running Applications in this guide or the Release Notes.

2.1 Simple Tutorial: "Hello, World"

In this simple application, a publisher repeatedly writes the string "Hello, World" to any number of identical subscribers. We'll start with the completed code that is included in your *Message Service* distribution.

2.1.1 Inspect the Files

Start by examining the files and folders included in this example. Change to the **<path to examples>/jms/examples** directory.

In that directory, you should see the configuration file **ExampleQosConfig.xml**. You should also see a **com/** directory, which is the root of the package **com.rti.jms.example.hello**. That package directory includes the Java source files **HelloWorldPublisher.java** and **HelloWorldSubscriber.java**. (The **<path to examples>/jms** directory also contains source packages for other example programs; you can ignore them for now.)

2.1.2 Set Up the Environment

There are a few things to take care of before you start working with the example code.

1. Set the **RTIJMSHOME** environment variable.

   Set the environment variable **RTIJMSHOME** to the *RTI Message Distribution Service* installation directory. The defaults paths are in Paths Mentioned in Documentation (Section 1.5). This step is optional, but this variable is used below; if you choose not to set it, simply "expand" the variable as you type.

2. Update your path.

   Add *RTI Message Service's* <**RTIJMSHOME>/bin** directory to your path. This will allow you to run some of the simple command-line utilities included in your distribution without typing the full path to the executable.

   On UNIX-based systems: Add the <**RTIJMSHOME>/bin** directory to your **PATH** environment variable.

   On Windows systems: Add the <**RTIJMSHOME>/bin** directory to your **Path** environment variable.

3. Make sure Java is available.

   Make sure that a supported version of the JDK is on your path. The Release Notes list the Java versions that are supported.

   On Linux systems: Note that GNU **java** (from the GNU Classpath project) is *not* supported—and will typically not work—but is on the path by default on many Linux systems.

4. Update your class path.

   The following jar files must be on the class path:

   - **<RTIJMSHOME>/lib/java/rtijms.jar**
   - **<RTIJMSHOME>/lib/java/nddsjava.jar**

   Also make sure that your Java installation includes Java Enterprise Edition (EE). Java Standard Edition (SE) does not ship with the **javaee.jar** file that contains the definitions of the JMS-standard interfaces that *Message Service* requires. This file can be found in the **lib** directory of your Java EE installation.

5. Make sure the native libraries are available.

   Make sure that your application can load the *RTI Message Distribution Service* native libraries. These libraries are in the directory **<RTIJMSHOME>/lib/<*architecture*>**, where <*architecture*> may be **i86Win32VS2010**, **i86Linux2.6gcc3.4.3**, or another name in your **lib** directory. (The **gcc** part—only present on UNIX-based architectures—identifies the corresponding native architecture that relies on the same version of the C runtime library.)

- **On UNIX-based systems**: Add this directory to your **LD_LIBRARY_PATH** environment variable.
- **On Windows systems**: Add this directory to your **Path** environment variable. For more information about where Windows looks for dynamic libraries, see http://msdn.microsoft.com/en-us/library/ms682586.aspx.

2.1.3 Build and Run the Applications

Now we will compile and run the applications. If you have trouble with this section, consult Chapter 3: Building and Running Applications; it contains more detailed information.

❑ Both applications accept the same command line argument, which is the number of messages to process before terminating. A value of 0 means an infinite number of messages; this is also the default if you specify nothing.

❑ You need to know the name of your target architecture (look in your **<RTIJMSHOME>/lib** directory). Use it in place of `<arch>` in the commands below. For example, your architecture might be i86Win32VS2008.

❑ If you're running on a Windows system and your computer is *not* connected to a network, you must configure *Message Service* to only use shared memory. Modify the **ExampleQosConfig.xml** file as follows:

```
<connection_factory name="helloConnectionFactory">
    <transport_builtin>
        <mask>TRANSPORTBUILTIN_SHMEM</mask>
    </transport_builtin>
    <discovery>
       <initial_peers>
            <element>shmem://</element>
       </initial_peers>
       <multicast_receive_addresses></multicast_receive_addresses>
    </discovery>
</connection_factory>
```

1. Make sure the following JAR files are on the class path; they can be found in the **<RTIJMSHOME>/lib/java** directory of your installation:

   On Windows systems:

   ```
   > set CLASSPATH=%RTIJMSHOME%\lib\java\rtijms.jar;%CLASSPATH%
   > set CLASSPATH=<Java EE install dir>\lib\javaee.jar;%CLASSPATH%
   > set CLASSPATH=.;%CLASSPATH%
   ```

   On UNIX/Linux systems: (exact syntax depends on your shell)

   ```
   > export CLASSPATH=$RTIJMSHOME/lib/java/rtijms.jar:$CLASSPATH
   > export CLASSPATH=<Java EE install dir>/lib/javaee.jar:$CLASSPATH
   > export CLASSPATH=.:$CLASSPATH
   ```

   Also, you must ensure that your Java installation includes Java Enterprise Edition (EE). Java Standard Edition (SE) does not ship with the **javaee.jar** file that contains the definitions of the JMS-standard interfaces that *Message Service* requires.

2. Use the Java compiler (**javac**) to compile. Go to the **<path to examples>/jms** directory and enter:

   ```
   > javac com/rti/jms/example/hello/*.java
   ```

3. Add the **nddsjava.jar** library to your class path if you have not already done so:

   On Windows systems:

   ```
   > set CLASSPATH=%RTIJMSHOME%\lib\java\nddsjava.jar;%CLASSPATH%
   ```

   On UNIX/Linux systems: (the exact syntax depends on your shell)

   ```
   > export CLASSPATH=$RTIJMSHOME/lib/java/nddsjava.jar:$CLASSPATH
   ```

   If you have not already done so as part of your installation, you must also add the native libraries required by *Message Service* to your **Path** (on Windows systems) or **LD_LIBRARY_PATH** (on UNIX/Linux systems) environment variable. (In the following examples, replace *<arch>* with your architecture):

   On Windows systems:

   ```
   > set Path=%RTIJMSHOME%\lib\<arch>;%Path%
   ```

   For example:

   ```
   > set Path=%RTIJMSHOME%\lib\i86Win32VS2010;%Path%
   ```

   On UNIX/Linux Systems: (exact syntax depends on your shell)

   ```
   > export LD_LIBRARY_PATH=$RTIJMSHOME/lib/<arch>:$LD_LIBRARY_PATH
   ```

   For example:

   ```
   > export LD_LIBRARY_PATH=\
   >       $RTIJMSHOME/lib/i86Linux2.6gcc4.1.1:$LD_LIBRARY_PATH
   ```

4. Start **HelloWorldPublisher**.

   ```
   > java com.rti.jms.example.hello.HelloWorldPublisher
   ```

5. Verify that the **HelloWorldPublisher** starts and that it displays the following output on the console:

   ```
   Sending, count 0
   Sending, count 1
   Sending, count 2
   ...
   ```

6. Open a new command prompt. Start the **HelloWorldSubscriber**:

   ```
   > java com.rti.jms.example.hello.HelloWorldSubscriber
   ```

   You should see the subscribing application start.

   ```
   Received:
       Text: Hello, World
   Received:
       Text: Hello, World
   Received:
       Text: Hello, World
   ...
   ```

7. Stop both applications.

2.1.4 Understanding the Code

Now that you've successfully compiled and run the code, take this opportunity to inspect the source files and configuration file to better understand what they do. The comments in each file explain the logic and configuration there step by step. The next tutorial, which is more involved than this one, will also explain many of these concepts in more detail.

Once you have completed both tutorials, you will be ready to continue reading with the User's Manual.

2.2 Advanced Tutorial: Market Data

This tutorial uses a simple market data application to illustrate the mechanics of using *Message Service* to build a distributed application with specific data delivery requirements.

The example's requirements illustrate the basics of how to implement these common use cases:

- ❑ Reliable delivery to late-joining subscribers
- ❑ Minimum separation to control how quickly messages are received
- ❑ Seamless failover to a backup publisher

In Design Phase (Section 2.2.1), we'll describe the basic requirements of our example application, map out the structure of the data to be published, and decide what "Topics" to use. Then we'll list the required network-related behavior and see which Quality of Service (QoS) parameters to change to implement those behaviors.

In the final sections, we'll walk through the 'hands on' process of creating and running the applications. You will see how to modify the code to implement each required behavior.

- ❑ The exercises show how to create the example application from scratch.
- ❑ Your completed code should match the source code provided with your *Message Service* installation (in **<path to examples>/jms**).

In this tutorial, you will learn how to:

- ❑ Design the Topics and keys to use for the example.
- ❑ Analyze the example's communication requirements and map them to QoS Policies.
- ❑ Write a configuration file to specify these policies.
- ❑ Write code to supply hypothetical data to be passed from publisher to subscriber.

This tutorial will not cover all the concepts of *Message Service*. The User's Manual covers all the middleware topics in detail.

2.2.1 Design Phase

This chapter describes the example stock application, with particular focus on the structure of the data to be published, as well as the Quality of Service (QoS) parameters that will be used to implement the application's communications requirements. We'll also introduce some important RTI terms such as keys, instances, and Topics.

This chapter includes the following sections:

- ❑ Example Stock Application (Section 2.2.1.1)
- ❑ Designing Data Structures and Topics (Section 2.2.1.2)

2.2.1.1 Example Stock Application

We will use a stock application to illustrate some of the key concepts of *Message Service*. The application will have publishers and subscribers exchanging data on stock news and stock prices.

Requirement I: When you start the subscribing application, you want to receive the last 20 news items for the stock and the most recent update for the stock price.

Requirement II: Like some real world stock applications, we will have two levels of service, Guest and Premium. The service level, entered as a command-line parameter, will determine how often you get a stock price update:

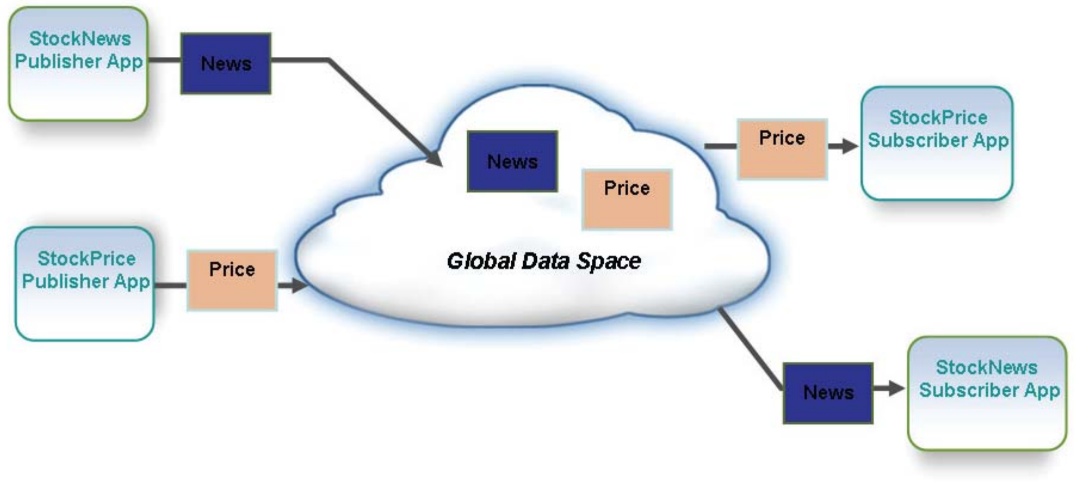
- ❑ Guest Level: Stock price updates every 15 seconds.
- ❑ Premium Level: Stock price updates every 4 seconds.

Requirement III: You only want to receive stock price updates from the 'most trusted' source. If that source disappears, you want to automatically get data from the next-most-trusted source. We'll use the same service level (guest or premium) to determine how long to wait before switching to the next most trusted source (that is, the 'deadline' for hearing from the most trusted source).

To keep the example focused on the middleware logic, we will not cover other aspects of building the application, such as user authentication, the user interface, or acquiring news and stock data from a database.

To keep things simple, we will create 4 separate applications:

- ❑ Stock News Publisher
- ❑ Stock News Subscriber
- ❑ Stock Price Publisher
- ❑ Stock Price Subscriber



2.2.1.2 Designing Data Structures and Topics

The first step in designing an *Message Service* application is to consider the structure of your data. For our example, we'll use the predefined JMS BytesMessages[1] type to efficiently represent the following data:

Table 2.1 **Stock News**

Type	Description
String	Stock ticker symbol
String	Stock news URL
String	Updated by
long	Updated date

Table 2.2 **Stock Price**

Type	Description
String	Stock ticker symbol
double	Stock price
String	Updated by
long	Updated date

When designing your own applications, consider your data types carefully—the design has a significant impact on the performance of the distributed application. To preserve network bandwidth, applications should send no more data than is required. For example, while we could combine the data pertaining to Stock Price and Stock News in every message, that would imply that we want to send stock news with every update to the stock price. This would not be desirable, since we plan to send stock price updates every 4 seconds, but only want to send stock news every 15 seconds.

At this point, it is helpful to have a basic understanding of the term, **Topics**. A **Topic** has a unique name. Applications can subscribe to a **Topic** and receive **Messages**. For the publishing and subscribing applications to communicate, they need to use the same **topic**. We will define two topics: “Example Stock News” and “Example Stock Price,” which will use the **Stock News** and **Stock Price** messages described above.

2.2.1.3 Selecting Quality of Service (QoS) Parameters

Quality of Service (QoS) policies allow you to control the behavior of application communications.

QoS policies give you the flexibility to change the behavior of the distributed application with zero or minimal code changes. The QoS policies you choose are picked up from a configuration file when you start the application. This allows you to test different QoS settings without even recompiling. We’ll see an example of this later in the tutorial.

Many QoS policies have to be set compatibly in both the publishing and subscribing applications. Other QoS policies apply to one side only. We will see examples of both cases.

For the sake of brevity, we will not cover all the QoS policies in this document. See the Configuration and Operation Manual for a detailed review of all the QoS policies.

Let’s look at the specific QoS changes needed for our application.

2.2.1.3.1 Requirement I: Late-Joiners Need Last Price and Last 20 News Messages

When the Stock News Subscriber application starts up, we want it to receive the last 20 news items.

When the Stock Price Subscriber application starts up, we want it to receive the last price.

1. See http://java.sun.com/javaee/5/docs/api/javax/jms/BytesMessage.html.

To meet this requirement, we will change the following QoS policies in the Stock News Publisher and Subscriber applications.

- ❑ **Reliability QoS Policy**

  The Reliability QoS Policy determines whether or not a producer's data will be reliably delivered by *Message Service* to matching consumers.

  By default, *Message Service* is configured for 'best-effort' delivery for the lowest possible latency and the greatest possible determinism.

  The **kind** field in the Reliability QoS Policy can be either:

  - **BEST_EFFORT**—*Message Service* will send messages only once to MessageConsumers. No effort or resources are spent to track whether or not sent messages are received. Messages may be lost.
  - **RELIABLE**—*Message Service* will send messages reliably—buffering published messages until they have been acknowledged by the consumers and resending any messages that may have been lost during transport.

  To ensure that messages are delivered reliably in our example, we'll change the Reliability QoS's **kind** field to RELIABLE.

- ❑ **History QoS Policy**

  The History QoS Policy controls whether *Message Service* should deliver only the most recent value of an instance, attempt to deliver all intermediate values, or do something in between. The History QoS Policy's **depth** field specifies the number of messages to be kept.

  By default, *Message Service* only keeps the last message (depth = 1).

  To change the behavior so that the last 20 messages are saved for late-joining subscribers, we'll change the History QoS's **depth** field to 20 in the Stock News Publisher.

  Since we only need to keep the last stock price, we can use the default depth (1) in the Stock Price Publisher and Subscriber.

- ❑ **Durability QoS**

  The Durability QoS Policy controls whether or not, and how, published messages are stored for subscribing applications that are found after the messages were initially sent (late joiners). A consumer uses this QoS Policy to request messages that were published before the consumer was created. These are known as 'historical' messages.

  By default, *Message Service* does not need to keep any messages for possible delivery to future subscribers.

  These are the possible settings for this QoS:

  - **VOLATILE** (default)—*Message Service* will not deliver any messages to *MessageConsumers* that are discovered after the messages were initially published.
  - **TRANSIENT_LOCAL**—*Message Service* will store and send previously published messages for delivery to newly discovered *MessageConsumers*. The History QoS Policy determines exactly how many messages are saved or delivered (see below).

    To change the behavior so that already sent messages are saved, we'll change the Durability QoS's **kind** field to TRANSIENT_LOCAL. We must make this change in all four applications (Stock News Publisher and Subscriber, Stock Price Publisher and Subscriber).

- **TRANSIENT, PERSISTENT** *Message Service* will store sent messages in a database external to the publishing process itself. These levels of durability require one or more instances of the *RTI Persistence Service* on your network.

2.2.1.3.2 Requirement II: Service Level Controls How Often Stock Prices are Received

When you start the Stock Price Subscriber application with the Guest service level, you will get stock price updates every 15 seconds. When you start the application with the Premium service level, you will get stock price updates every 4 seconds.

To meet this requirement, we will change the following QoS policy in the Stock Price Subscriber application.

❏ Time-Based Filter QoS Policy

The Time-Based Filter QoS Policy allows you to specify that a consumer does not want to receive more than one message during each **minimum_separation** time period, regardless of how fast producers are sending new messages. This QoS Policy only applies to consumers, not producers.

By default, *Message Service* does not impose any 'minimum separation' time between messages.

The service level command-line parameter will determine the **minimum_separation** value of the Stock Price Subscriber:

- Guest Level: 15 seconds.
- Premium Level: 4 seconds.

In either case, the networking code remains the same!

The Time-Based Filter QoS Policy is only set on the *MessageConsumer*, so no changes are needed in the Stock Price Publisher.

2.2.1.3.3 Requirement III: Stock Price From the Most 'Trusted' Publisher

We want the Stock Price Subscriber application to get updates from the most trusted source. If that source disappears for some reason, then data from the next-most-trusted source will be delivered to all the subscribing applications.

To meet this requirement, we will change the following QoS policies in the Stock Price Publisher and Stock Price Subscriber applications.

❏ Ownership QoS Policy

The Ownership QoS Policy specifies whether multiple producers can update the same instance.

By default, a subscribing application can receive messages from multiple producers. That is, the data instance is shared among all its producers—there is no exclusive owner.

The **kind** field in the Ownership QoS Policy can be either:

- SHARED (default)—In effect, there is no "owner." The subscribing application will receive modifications from all producers.
- EXCLUSIVE—The producer with the highest ownership strength is considered the exclusive owner. Only messages from the 'owner' are passed on to the consumers. The owner of an instance can change dynamically.

To ensure that messages are from the single most trusted source, we'll change the Ownership QoS's **kind** field to EXCLUSIVE.

Both the Stock Price Publisher and Stock Price Subscriber applications must use the same settings for the Ownership QoS Policy, so we must make this change in both applications.

❑ Ownership Strength QoS Policy

You can use the Ownership Strength QoS Policy to indicate which producer has the highest strength. When the Ownership QoS Policy is set to EXCLUSIVE, the producer with the highest strength owns the instance.

By default, all producers have equal strength (0).

Rather than hard-code a strength value, we'll use a command-line parameter. This will allow us to start multiple copies of the publishing application with various strengths.

Since this QoS Policy only applies to producers, we will only need to make this change in the Stock Price Publisher application.

❑ Deadline QoS Policy

For a producer, the Deadline QoS Policy sets the maximum amount of time that may pass between sending new messages. That is, the producer is offering to send new messages within this deadline period.

For a consumer, the Deadline QoS Policy sets the maximum amount of time that is acceptable between receipt of new messages. That is, the consumer is requesting that messages are received by this deadline period.

By default, there is no 'deadline' period by which a producer must publish new messages. (The Deadline QoS Policy has a value of 'infinite').

The deadline offered by the producer cannot be greater than the deadline requested by the consumer.

For the producer, we will use a deadline of 8 seconds.

For the consumer, we will use a different deadline for each service level:

- Guest Level: deadline = 30 seconds
- Premium Level: deadline = 8 seconds

For our example, we'll set the above deadlines in both the Stock Price Publisher and Subscriber applications.

2.2.1.4 Summary

The example's requirements, summarized below, illustrate how to achieve these common use cases:

Use Case	Requirement in Stock Example	Related QoS Policies
Late-joining subscribers	I. Late-Joiners Need Last Price and Last 20 News Messages	Durability, History, Reliability
Minimum separation to control how quickly messages are received	II. Service Level Controls How Often Stock Prices are Received	Time-Based Filter
Seamless failover to a backup publisher	III. Stock Price From the Most 'Trusted' Publisher; Service Level Determines the Deadline	Ownership, Ownership Strength, Deadline

2.2.2 Implementing the Design

The exercises show you how to create the example application from scratch. Your completed code should match the source code provided with your *Message Service* installation (in **<path to examples>/jms/com/rti/jms/example/stock**).

This section describes how to:

- ❑ Create Files (Section 2.2.2.1)
- ❑ Publish and Subscribe to Data (Section 2.2.2.2)
- ❑ Implement the Requirements in StockNews (Section 2.2.2.3)
- ❑ Implement the Requirements in StockPrice (Section 2.2.2.4)

2.2.2.1 Create Files

Start by creating the files and folders you'll need for this example.

1. Create a new directory, such as **MyStockTutorial**, and change to that directory.
2. Create the class StockNewsPublisher in the file **StockNewsPublisher.java**:

```
public class StockNewsPublisher {
    // TODO: fill in later
}
```

3. Create the class StockNewsSubscriber in the file **StockNewsSubscriber.java**:

```
public class StockNewsSubscriber {
    // TODO: fill in later
}
```

4. Create the class StockPricePublisher in the file **StockPricePublisher.java**:

```
public class StockPricePublisher {
    // TODO: fill in later
}
```

5. Create the class StockPriceSubscriber in the file **StockPriceSubscriber.java**:

```
public class StockPriceSubscriber {
    // TODO: fill in later
}
```

6. Create the XML configuration file, **ExampleQosConfig.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<jms>
    <!-- TODO: fill in later -->
</jms>
```

2.2.2.2 Publish and Subscribe to Data

First, we'll make code changes that are not strictly related to any single requirement. These changes will supply example data for the StockNews and StockPrice messages and add command-line parameters for the service level (Guest or Premium), data source, and strength.

When we are done adding the command-line parameters, the syntax for the applications will be:

```
StockNewsPublisher [#messages]
StockNewsSubscriber [#messages]
StockPricePublisher [#messages] [provider] [strength]
StockPriceSubscriber [#messages] [guest|premium]
```

The first parameter is included in every application:

- ❑ **# messages** In the publishing application, this is the number of messages to publish before terminating. In the subscribing application, this is the number of 4-second intervals for which it will sleep/awake before terminating. In both applications, a value of 0 means an infinite number of messages. Default: 0 (infinite).

We will add the following parameters for our example:

- ❑ **provider** You can enter any string here; it will be displayed in the "updated by" field. Default: "Message Board."
- ❑ **strength** The OwnershipStrength QoS for the StockPrice Publisher. Default: 0. Valid range: 0 to 1 million.
- ❑ **guest | premium** This is the service level. (For simplicity in this example, we have not provided any error checking or allowance for mixed upper/lower case.) Default: guest.

2.2.2.2.1 StockNews Publisher

Changes to the example code are shown in **`bold blue`**.

1. Make the following changes in **StockNewsPublisher.java**.

   Note: If you copy/paste the above code, you may need to correct the quotation marks in your file.

```
import java.util.Properties;
import javax.jms.*;
import javax.naming.Context;
import com.rti.naming.InitialContext;

public class StockNewsPublisher {
    public static void main(String[] args) throws Exception {
// 1. Get number of messages to send from command line; 0 means publish indefinitely
    int messageCount = 0;
     if (args.length >= 1) {
         messageCount = Integer.valueOf(args[0]).intValue();
     }
// 2. Initialize administered objects from XML file; the code won't run until we fill in that file
     Properties prop = new Properties();
     prop.setProperty(Context.PROVIDER_URL, "ExampleQosConfig.xml");
     InitialContext ctx = new InitialContext(prop);
// 3. Look up administered ConnectionFactory from file. Use the factory to create
//    other entities:
    ConnectionFactory factory = (ConnectionFactory) ctx.lookup(
"stock/connectionFactory");
    Connection conn = factory.createConnection();
     try {
        Session session = conn.createSession(
false, Session.DUPS_OK_ACKNOWLEDGE);
        Topic topic = (Topic) ctx.lookup("stock/Example Stock News");
        MessageProducer producer = session.createProducer(topic);
```

```
            conn.start();

    // 4. Send messages, filling in hypothetical data:
            BytesMessage instance = session.createBytesMessage();
            final long sendPeriodMillis = 4 * 1000; // 4 seconds
            for (int count = 0;
                (messageCount == 0) || (count < messageCount);
                 ++count) {
                    System.out.println("Sending Stock News, count " + count);

                    instance.writeUTF("RTII");  // stock ticker symbol
                    instance.writeUTF("http://www.rti.com"); // stock news URL
                    instance.writeUTF("reuters");   // updated by
                    instance.writeLong(System.currentTimeMillis()); // date
                    producer.send(instance);

                    // Prepare message for reuse, then pause for four
                    // seconds before sending again:
                    instance.clearBody();

                    Thread.sleep(sendPeriodMillis);
                }
            // Close down communication:
            } finally {
                if (conn != null) {
                    conn.close();
                }
            }
        }
    }
```

2. Next, add the following **bold blue lines** to **ExampleQosConfig.xml**:

```
<jms>

// 1. Define a library to store the managed object (i.e. ConnectionFactory and Topic)
//    configurations. All managed object configurations are stored in libraries:
    <library name="stock">

// 2. Add a simple configuration for the ConnectionFactory that is used in the code to creation
//    the MessageProducer and MessageConsumer.:
        <connection_factory name="connectionFactory">
        </connection_factory>

// 3. Define the "Example Stock News" topic and prepare a placeholder for configuring the
//    QoS of producers to that topic:
        <topic name="Example Stock News">
            <producer_defaults>
                <!-- TO DO: QoS policies will be filled in later. -->
            </producer_defaults>
        </topic>
    </library>
</jms>
```

2.2.2.2.2 StockNews Subscriber

1. Make the following changes in **StockNewsSubscriber.java**:

```
import java.util.*;
import javax.jms.*;
import javax.naming.Context;
import com.rti.naming.InitialContext;

public class StockNewsSubscriber {
    public static void main(String[] args) throws Exception {
```

// 1. Get number of messages to expect from command line; 0 means wait indefinitely:

```
        int messageCount = 0;
        if (args.length >= 1) {
            messageCount = Integer.valueOf(args[0]).intValue();
        }
```

// 2. Initialize administered objects from XML file; the code won't run until files is filled in:

```
        Properties prop = new Properties();
        prop.setProperty(Context.PROVIDER_URL, "ExampleQosConfig.xml");
        InitialContext ctx = new InitialContext(prop);
```

*// 3. Look up administered **ConnectionFactory** from file. Use factory to create other entities.*

```
        ConnectionFactory factory = (ConnectionFactory) ctx.lookup(
                "stock/connectionFactory");
        Connection conn = factory.createConnection();
        try {
            Session session = conn.createSession(
                    false, Session.DUPS_OK_ACKNOWLEDGE);
            Topic topic = (Topic) ctx.lookup("stock/Example Stock News");
            MessageConsumer consumer = session.createConsumer(topic);
```

// 4. Set listener to handle received messages:

```
            consumer.setMessageListener(new MessageListener() {
                public void onMessage(Message message) {
                    System.out.println("Received message: ");
                    if (message instanceof BytesMessage) {
                        try {
                            BytesMessage bytes = (BytesMessage) message;
                            System.out.println(
                                "    Symbol: " + bytes.readUTF());
                            System.out.println(
                                "    URL: " + bytes.readUTF());
                            System.out.println(
                                "    By: " + bytes.readUTF());
                            System.out.println(
                                "    Time: " + new Date(bytes.readLong()));
                        } catch (JMSException jx) {
                              System.err.println("Error reading message:");
                              jx.printStackTrace();
                        }
                    } else {
                        System.out.println("    " + message);
                    }
                }
            });

            conn.start();
```

// 5. Wait for messages to arrive:

```
            final long receivePeriodSec = 4;
            for (int count = 0;
                (messageCount == 0) || (count < messageCount);
                  ++count) {
                System.out.println("Stock News subscriber sleeping for " +
                                    receivePeriodSec +
                                    " sec...");
                 Thread.sleep(receivePeriodSec * 1000);  // in millisec
            }
```

// 6. Close down communication:

```
        } finally {
            if (conn != null) {
                conn.close();
            }
            }
        }
    }
```

2. Next, add the following to **ExampleQosConfig.xml**:

```
<library name="stock">
```

// 1. Reuse the same ConnectionFactory:

```
    <connection_factory name="connectionFactory">
    </connection_factory>
    <topic name="Example Stock News">
            ...
```

// 2. Prepare a placeholder for configuring the QoS of consumers of that topic:

```
        <consumer_defaults>
            <!-- TO DO: QoS policies will be filled in later. -->
        </consumer_defaults>
    </topic>
</library>
```

2.2.2.2.3 StockPrice Publisher

1. Make the following changes in **StockPricePublisher.java**:

```
import java.util.Properties;
import javax.jms.*;
import javax.naming.Context;
import com.rti.naming.*;

public class StockPricePublisher {
    public static void main(String[] args) throws Exception {
```

// 1. Get number of messages to send from command line; 0 means publish indefinitely:

```
        int messageCount = 0;
        if (args.length >= 1) {
            messageCount = Integer.valueOf(args[0]).intValue();
        }
```

// 2. Get price provider from command line. If none is specified, assume the default
//   "Message Board." This string will be used for filling in the "updated by" field.

```
        String price_provider = "Message Board"; // Default provider
        if (args.length >= 2) {
            price_provider = args[1];
        }
```

// 3. Initialize administered objects from XML file; the code won't run until we fill in
//   that file. Use properties to programmatically override QoS values that were
//   loaded from the file.

```
        Properties prop = new Properties();
        prop.setProperty(Context.PROVIDER_URL, "ExampleQosConfig.xml");
        InitialContext ctx = new InitialContext(prop);
```

// 4. Look up administered ConnectionFactory from file. Use the factory to create
//   the other entities.

```
        ConnectionFactory factory = (ConnectionFactory) ctx.lookup(
                "stock/connectionFactory");
        Connection conn = factory.createConnection();
        try {
            Session session = conn.createSession(
                    false, Session.DUPS_OK_ACKNOWLEDGE);
            Topic topic = (Topic) ctx.lookup("stock/Example Stock Price");
            MessageProducer producer = session.createProducer(topic);
```

```
                conn.start();
```

// 5. Send messages, filling in hypothetical data:

```
                BytesMessage instance = session.createBytesMessage();
                final long sendPeriodMillis = 4 * 1000; // 4 seconds
                for (int count = 0;
                    (messageCount == 0) || (count < messageCount);
                    ++count) {
                  System.out.println("Sending Stock Price, count " + count);

                  instance.writeUTF("RTII");      // stock ticker symbol
                  instance.writeDouble(32.0 + (0.1 * count)); // stock price
                  instance.writeUTF(price_provider);  // updated by date/time
                  instance.writeLong(System.currentTimeMillis());
                  producer.send(instance);
```

// 6. Prepare message for reuse, then pause for four seconds before sending again:

```
                  instance.clearBody();
                  Thread.sleep(sendPeriodMillis);
                }
```

// 7. Close down communication:

```
            } finally {
                if (conn != null) {
                    conn.close();
                }
            }
        }
}
```

2. Open **ExampleQosConfig.xml** and add another topic as follows:

```
<library name="stock">
    ...
```

<!-- Define the "Example Stock Price" topic and prepare a placeholder for configuring -->

<!-- the QoS of producers to that topic -->

```
    <topic name="Example Stock Price">
        <producer_defaults>
            <!-- TO DO: QoS policies will be filled in later. -->
        </producer_defaults>
    </topic>
</library>...
```

2.2.2.2.4 StockPrice Subscriber

1. Make the following changes in **StockPriceSubscriber.java**:

```
import java.util.*;
import javax.jms.*;
import javax.naming.Context;
import com.rti.naming.*;

public class StockPriceSubscriber {
    public static void main(String[] args) throws Exception {
```

// 1. Get number of messages to expect from command line; 0 means wait indefinitely:

```
        int messageCount = 0;
        if (args.length >= 1) {
            messageCount = Integer.valueOf(args[0]).intValue();
        }
```

// 2. Initialize administered objects from XML file; the code won't run until we fill in

//   that file. Use properties to programmatically override QoS values loaded from the file.

```
        Properties prop = new Properties();
        prop.setProperty(Context.PROVIDER_URL, "ExampleQosConfig.xml");
        InitialContext ctx = new InitialContext(prop);
```

*// 3. Look up administered **ConnectionFactory** from file. Use the factory to create the*
//   other entities.

```
        ConnectionFactory factory = (ConnectionFactory) ctx.lookup(
                "stock/connectionFactory");
        Connection conn = factory.createConnection();
        try {
            Session session = conn.createSession(
                    false, Session.DUPS_OK_ACKNOWLEDGE);
            Topic topic = (Topic) ctx.lookup("stock/Example Stock Price");
            MessageConsumer consumer = session.createConsumer(topic);
```

// 4. Set listener to handle received messages:

```
            consumer.setMessageListener(new MessageListener() {
                public void onMessage(Message message) {
                    System.out.println("Received message: ");
                    if (message instanceof BytesMessage) {
                        try {
                            BytesMessage bytes = (BytesMessage) message;
                            System.out.println(
                              "    Symbol: " + bytes.readUTF());
                            System.out.printf(
                              "    Price: $%1$.2f\n",
                              bytes.readDouble());
                            System.out.println(
                              "    Updated By: " + bytes.readUTF());
                            System.out.println(
                              "    Time: " + new Date(bytes.readLong()));
                        } catch (JMSException jx) {
                            System.err.println("Error reading message:");
                            jx.printStackTrace();
                        }
                    } else {
                        System.out.println("    " + message);
                    }
                }
            });
            conn.start();
```

// 5. Wait for messages to arrive:

```
            final long receivePeriodSec = 4;
            for (int count = 0;
                 (messageCount == 0) || (count < messageCount);
                 ++count) {
               System.out.println(
                   "Stock Price subscriber sleeping for " +
                   receivePeriodSec +
                   " sec...");
                   Thread.sleep(receivePeriodSec * 1000); // in msec
            }
```

// 6. Close down communication:

```
        } finally {
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

2. Open **ExampleQosConfig.xml** and add another topic as follows:

```
<topic name="Example Stock Price">
    ...
```

```
<!-- Prepare a placeholder for configuring the QoS of consumers of that topic -->
    <consumer_defaults>
        <!-- TO DO: QoS policies will be filled in later. -->
    </consumer_defaults>
</topic>
```

2.2.2.3 Implement the Requirements in StockNews

In Design Phase (Section 2.2.1), we identified the specific behavior we want in the application. Now we'll change the basic code you wrote previously—and fill in the configuration file—to implement each requirement. Table 2.3 summarizes how the requirements map to QoS Policies in the StockNews application.

Table 2.3 **Mapping Requirements to QoS in StockNews Publisher**

Requirement	StockNews Publisher (MessageProducer) Changes	StockNews Subscriber (MessageConsumer) Changes
I. Late-Joiners Need Last 20 News Messages	producer's reliability kind = RELIABLE	consumer's reliability kind = RELIABLE
	producer's history depth = 20	consumer's history depth = 20
	producer's durability kind = TRANSIENT_LOCAL	consumer's durability kind = TRANSIENT_LOCAL
II. Service Level Controls How Often Stock Prices are Received	no changes	no changes
III. Stock Price from the Most Trusted Publisher; Service Level Determines Deadline	no changes	no changes

1. Open **ExampleQosConfig.xml** and set the *MessageProducer* QoS parameters to as listed in Table 2.3.

```
<topic name="Example Stock News">
    <producer_defaults>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
        <history>
            <depth>20</depth>
        </history>
        <durability>
            <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
        </durability>
    </producer_defaults>
</topic>
```

2. Make the required *MessageConsumer* QoS changes listed in Table 2.3 to keep compatibility between the publisher and subscriber QoS policies.

```
<topic name="Example Stock News">
    <consumer_defaults>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
        <history>
            <depth>20</depth>
        </history>
        <durability>
            <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
        </durability>
        </consumer_defaults>
    </topic>
```

3. Save your changes.

2.2.2.3.5 Testing the StockNews Applications

Now we will compile and run the applications.

❑ You need to know the name of your target architecture (look in your **RTIJMSHOME\lib** directory). Use it in place of `<arch>` in the commands below. For example, your architecture might be i86Win32VS2010.

❑ If you are on a Windows system, and your computer is *not* connected to a network, you must configure *Message Service* to only use shared memory. For example:

```
<connection_factory name="connectionFactory">
    <transport_builtin>
        <mask>TRANSPORTBUILTIN_SHMEM</mask>
    </transport_builtin>
    <discovery>
        <initial_peers>
            <element>shmem://</element>
        </initial_peers>
        <multicast_receive_addresses></multicast_receive_addresses>
    </discovery>
</connection_factory>
```

1. Make sure the following JAR files are on the class path; they can be found in the **<RTIJMSHOME>/lib/java** directory of your installation:

   On Windows systems:

   ```
   > set CLASSPATH=%RTIJMSHOME%\lib\java\rtijms.jar;%CLASSPATH%
   > set CLASSPATH=<Java EE install dir>\lib\javaee.jar;%CLASSPATH%
   > set CLASSPATH=.;%CLASSPATH%
   ```

   On UNIX/Linux systems: (exact syntax depends on your shell)

   ```
   > export CLASSPATH=$RTIJMSHOME/lib/java/rtijms.jar:$CLASSPATH
   > export CLASSPATH=<Java EE install dir>/lib/javaee.jar:$CLASSPATH
   > export CLASSPATH=.:$CLASSPATH
   ```

   Also, you must ensure that your Java installation includes Java Enterprise Edition (EE). Java Standard Edition (SE) does not ship with the **javaee.jar** file that contains the definitions of the JMS-standard interfaces that *Message Service* requires[1].

2. Use the Java compiler (**javac**) to compile.

   ```
   > javac *.java
   ```

3. Add an additional library to your class path.

   On Windows systems:

   ```
   > set CLASSPATH=%RTIJMSHOME%\lib\java\nddsjava.jar;%CLASSPATH%
   ```

   On UNIX/Linux systems: (exact syntax depends on your shell)

   ```
   > export CLASSPATH=$RTIJMSHOME/lib/java/nddsjava.jar:$CLASSPATH
   ```

1. If you've chosen to place the tutorial code inside a package, or you are compiling the shipped tutorial code (which is in the com.rti.jms.example.stock package), you will need to run the javac compiler from the root of the package directory hierarchy and include the fully qualified name of the class, like this:

```
> javac com/rti/jms/example/stock/*.java
```

If you have not already done so as part of your installation, you must also add the native libraries required by *Message Service* to your **Path** (on Windows) or **LD_LIBRARY_PATH** (on UNIX/Linux) environment variable (replace `<arch>` with your architecture):

On Windows systems:

```
> set Path=%RTIJMSHOME%\lib\<arch>;%Path%
```

For example:

```
> set Path=%RTIJMSHOME%\lib\i86Win32VS2010;%Path%
```

On UNIX/Linux Systems: (exact syntax depends on your shell)

```
> export LD_LIBRARY_PATH=$RTIJMSHOME/lib/<arch>:$LD_LIBRARY_PATH
```

For example:

```
> export LD_LIBRARY_PATH=\
>         $RTIJMSHOME/lib/i86Linux2.6gcc4.1.1:$LD_LIBRARY_PATH
```

4. Now you're ready to run the **StockNewsPublisher** application[1]:

```
> java StockNewsPublisher 0
```

Test Requirement I

5. Verify that the **StockNews** publisher starts, and that it displays the following output on the console:

```
Sending Stock News, count 0
Sending Stock News, count 1
Sending Stock News, count 2
...
Sending Stock News, count 20
...
```

6. Open a new command prompt. After you see at least 20 messages of **StockNews** published in the first command console, start the StockNews subscriber (you must also make the environment variable changes described above in this new prompt):

```
> java StockNewsSubscriber 0
```

You should see the subscribing application start and then retrieve and publish the last 20 messages that the producer published.

```
StockNews subscriber sleeping for 4 sec...
Received:
    Symbol: RTII
    URL: http://www.rti.com
    Updated By: reuters
    Time: Mon Aug 11 10:17:35 PDT 2008
Received:
    Symbol: RTII
    URL: http://www.rti.com
    Updated By: reuters
    Time: Mon Aug 11 10:17:39 PDT 2008
```

1. If you've chosen to place the tutorial code inside a package, or you are running the shipped tutorial code (which is in the com.rti.jms.example.stock package), you will need to start the JVM from the root of the package directory hierarchy and include the fully qualified name of the class, like this:

```
> java com.rti.jms.example.stock.StockNewsPublisher 0
```

7. Stop both applications.

2.2.2.4 Implement the Requirements in StockPrice

In Design Phase (Section 2.2.1), we identified the specific behavior we want in the application. Now we'll change the basic code you wrote as well as the configuration file to implement each requirement.

Table 2.4 summarizes how the requirements map to QoS Policies in the **StockPrice** applications.

Table 2.4 **Mapping Requirements to QoS in StockPrice Publisher**

Requirement	StockPrice Publisher (MessageProducer) Changes	StockPrice Subscriber (MessageConsumer) Changes
I. Late-Joiners Need Last Price	producer's reliability kind = RELIABLE	consumer's reliability kind = RELIABLE
	producer's durability kind = TRANSIENT_LOCAL	consumer's durability kind = TRANSIENT_LOCAL
II. Service Level Controls How Often Stock Prices are Received	no changes in Price Publisher	*Guest*: consumer's time-based filter minimum separation = 15 sec *Premium*: consumer's time-based filter minimum separation = 4 sec
III. Stock Price from Most Trusted Publisher; Service Level Determines Deadline	producer's ownership kind = EXCLUSIVE	consumer's ownership kind = EXCLUSIVE
	producer's ownership strength = [*set via command-line*]	
	producer's deadline period = 8 sec	*Guest*: consumer's deadline period = 30 sec *Premium*: consumer's deadline period = 10 sec

Changes to the code are shown in **bold blue**.

1. Open **ExampleQosConfig.xml**. Configure the producer QoS as follows:

```
<topic name="Example Stock Price">
    <producer_defaults>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
        <history>
            <depth>1</depth>
        </history>
        <durability>
            <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
        </durability>
        <ownership>
            <kind>EXCLUSIVE_OWNERSHIP_QOS</kind>
        </ownership>
        <ownership_strength>
            <!-- Will be overridden in code -->
            <value>0</value>
        </ownership_strength>
        <deadline>
            <period>
```

```
                <sec>8</sec>
                <nanosec>0</nanosec>
            </period>
        </deadline>
    </producer_defaults>
</topic>
```

2. Open **StockPricePublisher.java**.
3. Override the ownership strength specified in the configuration file:

```
public static void main(String[] args) throws Exception {
    int messageCount = 0;
    if (args.length >= 1) {
        messageCount = Integer.valueOf(args[0]).intValue();
    }
    String price_provider = "Message Board"; // Default provider
    if (args.length >= 2) {
        price_provider = args[1];
    }
    int ownership_strength = 0;
    if (args.length >= 3) {
        ownership_strength = Integer.valueOf(args[2]).intValue();
    }
    Properties prop = new Properties();
    prop.setProperty(Context.PROVIDER_URL,
                                  "ExampleQosConfig.xml");
    prop.setProperty(
        RTIContext.QOS_FIELD_PREFIX +
        ":stock/Example Stock Price/producer_defaults/" +
        "ownership_strength/value",
        String.valueOf(ownership_strength));
    InitialContext ctx = new InitialContext(prop);
```

4. Return to **ExampleQosConfig.xml** and update the consumer configuration.

```
<topic name="Example Stock Price">
    <consumer_defaults>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
        <history>
            <depth>1</depth>
        </history>
        <durability>
            <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
        </durability>
        <ownership>
            <kind>EXCLUSIVE_OWNERSHIP_QOS</kind>
        </ownership>
        <time_based_filter>
            <minimum_separation>
                <!-- Will be overridden in code -->
                <sec>0</sec>
                <nanosec>0</nanosec>
            </minimum_separation>
        </time_based_filter>
        <deadline>
            <period>
                <!-- Will be overridden in code -->
                <sec>8</sec>
                <nanosec>0</nanosec>
            </period>
        </deadline>
      </consumer_defaults>
  </topic>
```

5. Open **StockPriceSubscriber.java**.
6. Override the QoS for the MessageConsumer as follows:

```
public static void main(String[] args) throws Exception {
    int messageCount = 0;
    if (args.length >= 1) {
        messageCount = Integer.valueOf(args[0]).intValue();
    }
    String service_level = "guest";
    if (args.length >= 2) {
        service_level= args[1];
    }
    Properties prop = new Properties();
    prop.setProperty(Context.PROVIDER_URL, "ExampleQosConfig.xml");
    if ("guest".equals(service_level)) {
        // Guest service level:
        prop.setProperty(
            RTIContext.QOS_FIELD_PREFIX +
            ":stock/Example Stock Price/consumer_defaults/" +
            "time_based_filter/minimum_separation/sec",
            "15");
        prop.setProperty(
            RTIContext.QOS_FIELD_PREFIX +
            ":stock/Example Stock Price/consumer_defaults/" +
            "deadline/period/sec",
            "30");
    } else {
        // Premium service level:
        prop.setProperty(
            RTIContext.QOS_FIELD_PREFIX +
            ":stock/Example Stock Price/consumer_defaults/" +
            "time_based_filter/minimum_separation/sec",
            "4");
        prop.setProperty(
            RTIContext.QOS_FIELD_PREFIX +
            ":stock/Example Stock Price/consumer_defaults/" +
            "deadline/period/sec",
            "8");
    }
    InitialContext ctx = new InitialContext(prop);
```

2.2.2.4.6 Testing the StockPrice Applications

Now we will compile and run the applications. Note the classpath and other environment variable settings described above; for the sake of brevity, they are not repeated here.

1. Use the Java compiler (**javac**) to compile[1].

```
> javac *.java
```

1. If you've chosen to place the tutorial code inside a package, or you are compiling the shipped tutorial code (which is in the com.rti.jms.example.stock package), you will need to run the javac compiler from the root of the package directory hierarchy and include the fully qualified name of the class, like this:

```
> javac com/rti/jms/example/stock/*.java
```

Test Requirement I

First we will test our Requirement I changes with the following scenario, which starts a **StockPrice** publishing application, waits until at least one message has been sent, then starts a **StockPrice** subscribing application—you'll see that the late-joining subscriber gets the last published message.

2. Start StockPricePublisher[1]:

```
> java StockPricePublisher 0
```

You should see the **StockPrice** publisher start and then display the following output on the console:

```
Sending Stock Price, count: 0
Sending Stock Price, count: 1
Sending Stock Price, count: 2
...
```

3. After you see a few messages of **StockPrice** published on the console, start the **StockPrice** subscriber from a different command prompt:

```
> java StockPriceSubscriber 0
```

You should see the subscribing application start and then retrieve and publish the last stock price message that the producer published.

```
Received:
    Symbol: RTII
    Price: $32.50
    Updated By: Message Board
    Time: Mon Aug 11 11:38:53 PDT 2008
```

4. Stop both applications.

Test Requirement II

Now we will test our Requirement II changes with the following scenario, which has a **StockPrice** publishing application sending at the default rate of every 4 seconds, and a **StockPrice** subscribing application that requires a minimum separation between messages of 15 seconds.

5. Open two command prompt windows.
6. In the first command prompt window, start the **StockPricePublisher**:

```
> java StockPricePublisher 0 reuters
```

7. In the second command prompt window, start **StockPriceSubscriber**, using the Guest service level:

```
> java StockPriceSubscriber 0 guest
```

8. Verify that the **StockPriceSubscriber** is reading messages every 15 seconds, even though the Publisher is sending messages every 4 seconds:

```
StockPrice subscriber sleeping for 4 sec...
Received:
    Symbol: RTII
    Price: $32.70
    Updated By: reuters
    Time: Mon Aug 11 11:45:51 PDT 2008
StockPrice subscriber sleeping for 4 sec...
StockPrice subscriber sleeping for 4 sec...
```

1. If you've chosen to place the tutorial code inside a package, or you are running the shipped tutorial code (which is in the com.rti.jms.example.stock package), you will need to start the JVM from the root of the package directory hierarchy and include the fully qualified name of the class, like this:

```
> java com.rti.jms.example.stock.StockPricePublisher 0
```

```
StockPrice subscriber sleeping for 4 sec...
StockPrice subscriber sleeping for 4 sec...
Received:
    Symbol: RTII
    Price: $33.10
    Updated By: reuters
    Time: Mon Aug 11 11:46:07 PDT 2008
StockPrice subscriber sleeping for 4 sec...
StockPrice subscriber sleeping for 4 sec...
StockPrice subscriber sleeping for 4 sec...
StockPrice subscriber sleeping for 4 sec...
Received:
    Symbol: RTII
    Price: $33.50
    Updated By: reuters
    Time: Mon Aug 11 11:46:23 PDT 2008
```

9. Stop both applications.

Test Requirement III

Now we will test our Requirement III changes by starting a **StockPrice** subscribing application and two **StockPrice** publishing applications with different strengths. You will see the subscriber get data only from the stronger publisher; then the stronger publisher will quit and messages from the weaker publisher will be used instead.

10. Open three separate command prompts.
11. In the first command prompt, start **StockPricePublisher** with an additional argument: the strength (100).

```
> java StockPricePublisher 0 reuters 100
```

12. In the second command prompt, start **StockPricePublisher** with 'prwire' as the stock price source and strength 10.

```
> java StockPricePublisher 0 prwire 10
```

13. In the third command prompt, Start **StockPriceSubscriber**.

```
> java StockPriceSubscriber 0 premium
```

14. Verify that **StockPriceSubscriber** is reporting data from 'reuters,' since it has the higher strength:

```
Received:
    Symbol: RTII
    Price: $32.50
    Updated By: reuters
    Time: Wed Mar 11 10:05:30 PDT 2009
Stock Price subscriber sleeping for 4 sec...
```

You may see data from **prwire** at first, if it discovers that Publisher first. Once both Publishers are discovered, the higher strength **reuters** data will prevail.

15. Terminate the first **StockPricePublisher** (from 'reuters' with strength 100) by typing `Ctrl-C`.
16. Verify that after the Subscriber's deadline expires, the subscriber seamlessly picks up the **prwire** source:

Notice that the "reuters" publisher stops. After the deadline, the subscriber accepts samples from "prwire."

```
StockPrice subscriber sleeping for 4 sec...
```

```
Received:
    Symbol: RTII
    Price: $37.00
    Updated By: reuters
    Time: Tue Aug 12 09:22:53 PDT 2008
StockPrice subscriber sleeping for 4 sec...
StockPrice subscriber sleeping for 4 sec...
Received:
    Symbol: RTII
    Price: $36.10
    Updated By: prwire
    Time: Tue Aug 12 09:23:01 PDT 2008
```

17. Stop the two remaining applications. This completes the Tutorial!

Chapter 3 Building and Running Applications

In the Tutorial chapter, you learned how to build and run a sample application. This chapter codifies the requirements—which were summarized in the Tutorial—for building and running applications.

This chapter is organized as follows:

3.1 Building Your Application

To compile an application that uses *Message Service*, the JAR files in Table 3.1 must be on the class path. RTI provides libraries in both debug and release formats. With respect to the API, both are equivalent; you may compile against whichever you prefer.

Table 3.1 **Java Libraries Required for Building**

Description	File Name (release)	File Name (debug)
Message Service API	rtijms.jar	rtijmsd.jar
J2EE SDK 1.4 or Java EE SDK 5[a]	j2ee.jar or javaee.jar [b]	

a. *Message Service* will not build against an installation of Java Standard Edition (SE) because the javaee.jar file, which contains the JMS APIs on which it depends, is not included in that edition. That jar file can be found in your Java EE installation.

b. The name of this jar file may vary depending on the version of Java EE you have installed.

You can modify the class path on the command line of the **javac** compiler or modify the **CLASSPATH** environment variable.

3.2 Running Your Application

Message Service requires a number of Java and native libraries.

3.2.1 Requirements when Using Microsoft Visual Studio

You must have the appropriate Visual Studio service pack or redistributable package installed on the machine where you are *running* an application linked with dynamic libraries. See the *RTI Message Service Release Notes* for details.

3.2.2 Java Libraries

When running an application that uses *Message Service*, the JAR files in Table 3.2 must be on the class path.

RTI provides libraries in both debug and release formats and recommends that you use one or the other consistently. For example, if you have **rtijms.jar** on your class path, you should also have **nddsjava.jar**; you should not have **nddsjavad.jar** on your class path.

Table 3.2 **Java Libraries Required to Run**

Description	File Name (release)	File Name (debug)
Message Service API	rtijms.jar	rtijmsd.jar
Core messaging engine	nddsjava.jar	nddsjavad.jar
Java Enterprise Edition 1.5 or 1.6[a]	j2ee.jar or javaee.jar[b]	

a. *Message Service* will not build against an installation of Java Standard Edition (SE) because the javaee.jar file, which contains the JMS APIs on which it depends, is not included in that edition. That jar file can be found in your Java EE installation.

b. The name of this jar file may vary depending on the version of Java EE you have installed.

You can modify the class path on the command line of the java virtual machine (VM) or by modifying the **CLASSPATH** environment variable.

3.2.3 Native Libraries

Message Service depends on a number of native libraries to provide the high performance and strict determinism RTI customers require.

RTI provides libraries in both debug and release formats and recommends that you use one or the other consistently; do not use the release version of some libraries and the debug version of others. The release and debug Java libraries, documented above, will attempt to load the corresponding versions, release or debug, of the native libraries on which they depend.

3.2.3.1 Native Libraries on Windows Systems

The native libraries in Table 3.3 are required on Windows systems.

Table 3.3 **Native Libraries Required to Run on Windows Systems**

Description	File Name (release)	File Name (debug)
Core messaging engine	nddsc.dll	nddscd.dll
	nddscore.dll	nddscored.dll
	nddsjava.dll	nddsjavad.dll

To use these libraries, add them to your **Path** environment variable.

3.2.3.2 Native Libraries on Linux Systems

The native libraries in Table 3.4 are required on Linux systems.

Table 3.4 **Native Libraries Required to Run on Linux Systems**

Description	File Name (release)	File Name (debug)
Core messaging engine	libnddsc.so	libnddscd.so
	libnddscore.so	libnddscored.so
	libnddsjava.so	libnddsjavad.so

To use these libraries, add them to your **LD_LIBRARY_PATH** environment variable.

3.2.4 Running on Linux: GCJ

Some Linux distributions include the GNU Compiler for Java (GCJ) by default and place the **java** and **javac** executables on the path, where they can be picked up in preference to a JDK from Sun Microsystems. RTI does not test or support this configuration; the Sun JDK with Java EE is the recommended Java distribution.

If you experience errors while building or running the tutorial, and you see the string '**gcj**' in the output, you are likely encountering this configuration. To resolve it, make sure that the directories containing your Sun JDK binaries appear before those containing the GCJ binaries in your **PATH** environment variable, or start the **java** and **javac** binaries using their full paths.

Chapter 4 Installing a Database (Optional)

Advanced

This chapter only applies if you want to use the Durable Producer History and/or Durable Consumer State features. For more information about these features, see the chapter on Durability and Persistence for High Availability in the User's Manual.

- **Durable Producer History** This feature allows a *MessageProducer* to locally persist its local history cache so that it can survive shutdowns, crashes and restarts. When an application restarts, each *MessageProducer* that has been configured to have durable producer history automatically loads all the data in its history cache from disk and can carry on sending data as if it had never stopped executing. To the rest of the system, it will appear as if the MessageProducer was temporarily disconnected from the network and then reappeared.
- **Durable Consumer State** This feature allows a *MessageConsumer* to locally persists its state and remember the data it has already received. When an application restarts, each *MessageConsumer* that has been configured to have durable consumer state automatically loads its state from disk and can carry on receiving data as if it had never stopped executing. Data that was received by the MessageConsumer before the restart will be suppressed, so it is not sent over the network.

To use either of these features, you need a relational database, such as MySQL®.

We have tested the following driver:

- MySQL ODBC 5.0.45

To use MYSQL as the database, you will need:

- MySQL 5.0.45 or higher

  The installation of MySQL is beyond the scope of this document. Please refer to the MySQL Reference Manual for the process to install and configure MySQL.
- MySQL ODBC 3.51 driver

  The driver is not bundled with the MySQL server and must be installed separately.

  The ODBC connector can be downloaded from:

  http://dev.mysql.com/downloads/connector/odbc/3.51.html

  The installation guide can be found here:

  http://dev.mysql.com/doc/refman/5.0/en/myodbc-installation-binary.html
- UnixODBC

  The MySQL ODBC driver requires an ODBC driver manager. We recommend using UnixODBC, a complete, free/open ODBC solution for Unix and Linux systems. The driver manager can be downloaded from http://www.unixodbc.org.

4.1 Creating a Data Source for MySQL

4.1.1 Linux/Solaris Systems

Message Service uses the MySQL ODBC driver to access data sources.

The connection information for each data source is stored in the ".odbc.ini" file. The stored information describes each data source in detail, specifying the driver name, a description, and any additional information the driver needs to connect to the data source.

To create the ".odbc.ini" file with a DSN named "Example", follow these steps:

1. Create a new text file named ".odbc.ini" in your home directory.
2. Insert these lines in the file:

```
[ODBC Data Source]
Example=MySQL Driver

[Example]
DRIVER=/usr/lib/libmyodbc3.so
Database=test
```

Note: Make sure that DRIVER points to the valid location of the MySQL ODBC driver on you system.

4.1.2 Windows Systems

Message Service uses the MySQL ODBC driver to access data sources.

The connection information for each data source is stored in the Windows registry. The stored information describes each data source in detail, specifying the driver name, a description, and any additional information the driver needs to connect to the data source.

To add a data source named "Example", follow these steps:

1. Open the ODBC Data Source Administrator:
   - On Windows 2000 systems: choose **Start, Control Panel, Performance and Maintenance, Administrative Tools, Data Sources (ODBC)**.
   - On Windows XP systems: choose **Start, Settings, Control Panel, Administrative Tools, Data Sources (ODBC)**.
2. Select the **System DSN** tab.
3. Click **Add**; the Create New Data Source dialog appears.
4. Select the MySQL driver from the list of drivers.
5. Click **Finish**; the MySQL ODBC Driver Configuration dialog appears.
6. Fill in the fields in the dialog.
   a. Enter "Example" as the Data Source Name (DSN).
   b. Enter a valid username as the user and a valid password as the password.
   c. Select a database (for example, "test").
   d. All other fields can be left empty.
7. Click **OK**.