SUN JAVA™ MESSAGE SERVICE GRID USER'S GUIDE

Release 5.1.3

Part Number: 820-1005-10

Version 20070417103524

Contents

List of Figures

List of Tables

Chapter 1

Sun Java Message Service Grid

1.1 Introducing Sun Java Message Service Grid

This chapter introduces Sun Java™ Message Service Grid (JMS Grid) and introduces its key features.

JMS Grid is a message oriented middleware product which provides a complete implementation of the Java Message Service (JMS) API, offering full JMS1.1 compliance for both publish/subscribe (via topics) and point to point (via queues) messaging.

JMS is a strategic element of the Java2 Enterprise Edition (J2EE) platform from Sun Microsystems. JMS is used in conjunction with the other technologies of J2EE to provide reliable asynchronous communication between components in a distributed computing environment.

A JMS Grid system consists of one or more JMS Grid server processes (daemons) and one or more JMS Grid client processes.

1.1.1 Architectural Features

Here are some of the key architectural features of JMS Grid:

- JMS 1.1 Compliance
- J2EE 1.4 Application Server support via a J2CA 1.5 resource adaptor (see release notes for a list of supported application servers)
- Dynamic subscription architecture for scalability
- Automatic flow control including producer throttling
- Dynamic queue and topic creation
- Daemon (server) clusters for fault-tolerance and high availability
- Networks of clusters for efficient and reliable message distribution across a WAN
- Load balancing of clients across clusters
- Load balancing of queue messages across multiple receivers
- Automatic failover of client connections, cluster connections and network connections in the event of failure
- Automatic recovery of restarted daemons
- JMX based management and performance monitoring

- Destination and message encryption
- Access control using users and groups
- SSL support
- HTTP and HTTPS tunneling through proxy servers and firewalls

These features are described in the remainder of this user's guide.

1.1.2 Client Features

JMS Grid provides a full implementation of the JMS 1.1 API. In addition, it provides additional client functionality:

- Wildcard destinations: allows messages to be sent to, or consumed from, multiple destinations using a hierarchical notation.
- Content based message selectors which apply to the body as well as the message header:
  - XPath message selectors on XML messages.
  - Bean message selectors on Object messages.
- Subscription notification: allows a client to listen for subscription events on a particular destination.
- Session inbox: allows a message producer to send messages to a new queue receiver or topic subscriber which will be received by that consumer and no others.
- Slow consumer notification: allows the server to notify a client that it is not processing messages quickly enough.

These features are all described in the section **Additional Programming Features** on page 179.

1.1.3 What's in This Chapter

1.1.4 Intended Audience

This document is intended for Java programmers, Java CAPS developers, and administrators who use distributed software systems.

1.1.5 Text Conventions

The following conventions are observed throughout this document.

Table 1 Text Conventions

Text Convention	Used For	Examples
Bold	Names of buttons, files, icons, parameters, variables, methods, menus, and objects	▪ Click **OK**. ▪ On the **File** menu, click **Exit**. ▪ Select the **eGate.sar** file.
`Monospaced`	Command line arguments, code samples; variables are shown in ***bold italic***	`java -jar filename.jar`
Blue bold	Hypertext links within document	See **Text Conventions** on page 21
Blue underlined	Hypertext links for Web addresses (URLs) or email addresses	http://www.sun.com

1.1.6 Screenshots Used in this Document

Depending on what products you have installed, and how they are configured, the screenshots in this document may differ from what you see on your system.

1.2 Related Documents

The following Sun Microsystems documents provide additional information about eGate Integrator and the Composite Application Platform Suite:

- *Composite Application Platform Suite Installation Guide*
- *eGate Integrator System Administration Guide*
- *eGate Integrator User's Guide*
- *Composite Application Platform Suite Primer*

1.3 Hardware and Software Requirements

1.3.1 Java Runtime Environment

JMS Grid now contains a suitable Java Runtime Environment (JRE) as part of the product installation. This JRE will be used to run JMS Grid daemons and all tools. This is normally a Java 1.5 JRE except in certain cases where one is not available and a Java 1.4 JRE is provided instead.

JMS Grid clients can use the bundled JRE or another JRE of your choice. JMS Grid supports both Java 1.5 and Java 1.5 JREs for client applications though we recommend use of the JRE supplied with JMS Grid. JMS Grid does not support JDKs earlier than version 1.4.

1.3.2 Platform Support

- JMS Grid 5.1.2 supports the following OS/processor platforms:
- Microsoft Windows XP, 2003 on Intel x86 processor
- Linux on Intel x86 processor
- Red Hat AS3 Linux on AMD 64 processor
- Solaris on Sparc processor
- Solaris on AMD 64 processor
- AIX (32-bit) on p-series 32-bit processor
- AIX (64-bit) on p-series 64-bit processor
- HPUX on Intel Itanium chip
- HPUX on HP PA-RISC 64-bit processor

1.3.3 Application Server Support

JMS Grid can be used with the following application servers in conjunction with the supplied J2CA resource adaptor:

- Sun Java CAPS Integration Server 5.1.2 (by a Sun Java CAPS 5.1.2 application)
- Sun Java System Application Server version Sun AS 8.1 EE (by a Sun Java CAPS 5.1.2 application)
- JBoss version 4 (by a non-CAPS application)

Note: *This release does not support BEA Weblogic or IBM Websphere.*

1.3.4 Compatibility with SpiritWave Versions

- This document describes how to upgrade a SpiritWave 6.1.3 daemon installation to use JMS Grid 5.1.2.
- A JMS Grid 5.1.2 server will support SpiritWave 6.1.3 clients.
- It will not be possible to use JMS Grid 5.1.2 clients with a SpiritWave 6.1.3 server.

1.3.5 Compatibility with Sun Java CAPS

JMS Grid 5.1.3 can be used in conjunction with Sun Java Composite Application Platform Suite 5.1.2. Previous versions of the Sun Java CAPS are not compatible.

1.3.6 LDAP Provider Support

In order to use LDAP for JNDI configuration, ensue that your LDAP server is configured to use Java object extensions. Refer to LDAP provider documentation for more information.

JMS Grid supports the following LDAP providers for the storage of administration data:

- Sun Java System Directory Server
- Open LDAP

1.4 Installing JMS Grid

After you have unpacked the compressed archive to a suitable directory you will need to run a new installer tool to prepare your JMS Grid installation for use.

The installation tool invites you to accept the licence conditions and then asks you to nominate the TCP and SSL ports to be used when a default daemon is started.

In addition to the instructions included in this section, also see the *Java™ Composite Application Platform Suite Installation Guide*.

1.4.1 Upload the JMS Grid Sar Files to the Sun Java CAPS Repository

If you are using JMS Grid on its own without the Sun Java Composite Application Platform Suite (CAPS) you can skip this step.

If you are using JMS Grid in conjunction with the Sun Java CAPS then the first step you need to take is to upload the appropriate JMS Grid sar files to your CAPS repository if this has not already been done.

For more information on the CAPS repository upload/download mechanism please see the Sun Java CAPS Repository Users Guide.

Upload the JMS Grid Runtime for the Required Platform(s)

The JMS Grid distribution CDs contains a separate JMS Grid runtime sar file for each supported platform. This contains the files necessary to run a JMS Grid daemon on that platform together with tools, examples and client components. The following JMS Grid runtime sar files are available in this release:

Table 2 JMS Grid Runtime Sar Files

Microsoft Windows XP, 2003 on Intel x86 processor	JMS_Grid-win32.sar
Linux on Intel x86 processor	JMS_Grid-Linux_x86.sar
Red Hat AS3 Linux on AMD 64 processor	JMS_Grid-Linux-RedHat-AS3_AMD64.sar
Solaris on Sparc processor	JMS_Grid-Solaris_SPARC.sar

Table 2 JMS Grid Runtime Sar Files

Solaris on AMD 64 processor	JMS_Grid-Solaris_AMD64.sar
AIX (32-bit) on p-series 32-bit processor	JMS_Grid-AIX32.sar
AIX (64-bit) on p-series 64-bit processor	JMS_Grid-AIX64.sar
HPUX on Intel Itanium chip	JMS_Grid-HPUX_Itanium.sar
HPUX on HP PA-RISC 64-bit processor	JMS_Grid-HPUX_PARISC.sar

After you have uploaded the required JMS Grid runtime sar file you can then manually download it as a compressed archive. The following section describes how to unpack and install this archive.

Upload the JMS Grid plug-in for Enterprise Designer

In addition you will need to upload the JMS Grid plug-in for Enterprise Designer. This is available in the following sar file which can also be found in the JMS Grid distribution CD:

Table 3 JMS Grid Plug-in for Enterprise Designer

All Platforms	JMS_Grid.sar

The JMS Grid plug-in does not need to be manually downloaded. Instead it you need to open Enterprise Designer and use its upload center tool to download and install the plug-in.

1.4.2 Obtain a JMS Grid Runtime Compressed Archive Suitable for your Machine

This section applies to all JMS Grid users whether you are using JMS Grid in conjunction with the Sun Java Composite Application Platform Suite (CAPS) or standalone.

You need to obtain a JMS Grid runtime compressed archive corresponding to the platform(s) on which you wish to run a JMS Grid daemon. There is a separate JMS Grid runtime compressed archive for each supported platform. This is because it contains a JRE suitable for that platform.

If you are using JMS Grid in conjunction with the CAPS you should download the compressed archive from your CAPS repository. If you are using JMS Grid on its own without Sun Java CAPS, you can simply copy it from the appropriate JMS Grid product CD.

The following distribution archives are available in this release:

Table 4 JMS Grid Runtime Compressed Archives

Microsoft Windows XP, 2003 on Intel x86 processor	JMS_Grid-win32.zip
Linux on Intel x86 processor	JMS_Grid-Linux_x86.tar.gz

Table 4 JMS Grid Runtime Compressed Archives

Red Hat AS3 Linux on AMD 64 processor	JMS_Grid-Linux-RedHat-AS3_AMD64. tar.gz
Solaris on Sparc processor	JMS_Grid-Solaris_SPARC.tar.gz
Solaris on AMD 64 processor	JMS_Grid-Solaris_AMD64.tar.gz
AIX (32-bit) on p-series 32-bit processor	JMS_Grid-AIX32.tar.gz
AIX (64-bit) on p-series 64-bit processor	JMS_Grid-AIX64.tar.gz
HPUX on Intel Itanium chip	JMS_Grid-HPUX_Itanium.tar.gz
HPUX on HP PA-RISC 64-bit processor	JMS_Grid-HPUX_PARISC.tar.gz

1.4.3 Unpack the JMS Grid Runtime Compressed Archive

A JMS Grid installation consists of a directory `JMS_Grid`. You need one JMS Grid installation on each computer on which you wish to run a JMS Grid daemon.

When deciding where to install JMS Grid, make sure not to choose a directory whose path contains a space (such as `C:\Program Files`). If you do then the various scripts and tools may not work.

The JMS Grid runtime is compressed using a format appropriate to the platform (i.e. zip format for Windows and tarball format for Unix). This file should be unpacked into your chosen installation directory using the standard tools for that archive format:

Windows

Use a zip management tool such as WinZip to unpack the archive to your chosen directory. The installation will consist of a single directory `JMS_Grid` under the directory you select.

Unix

Copy the compressed archive to where you would like the `JMS_Grid` directory to be and type:

```
gunzip JMS_Grid-{platform}.tar.gz
tar xvf JMS_Grid-{platform}.tar
```

This will create a single directory `JMS_Grid` under the current directory. You can then delete the compressed archive.

Note: On Solaris systems some versions of the tar command do not handle long filenames correctly and may generate a checksum error. If this happens then you should use either the GNU version of tar or the `pax` command:

```
pax -rvf JMS_Grid-{platform}.tar
```

1.4.4 Run the Installer

After you have unpacked the compressed archive to a suitable directory you then need to run the installer program to make JMS Grid ready to use.

To run the installer script simply execute the script `install.bat` (Windows) or `install` (UNIX) in the root directory of your JMS Grid installation and follow the instructions.

You will be asked to accept the licence conditions and nominate the TCP and SSL ports to be used when a default daemon is started. You will also be given the option of starting a default daemon as part of the installation process.

1.4.5 Set the JMSGRID Environment Variable

Once you have installed JMS Grid it is recommended that you create an environment variable JMSGRID and set it to the root directory if your JMS Grid installation. This allows you to run any of the JMS Grid commands without the need to navigate to the directory containing that command.

You don't have to set this variable, but if you don't you need to make sure that before you run any of the JMS Grid tools you should navigate to the directory containing that tool.

You should set the JMSGRID environment variable using the method appropriate for the operation system being used.

1.4.6 Configure the Management Console

The JMS Grid management console is a Java web application (war file). This can be run either in a standalone servlet container (web server) or a servlet container embedded in a JMS Grid daemon.

Configuring a Standalone Servlet Container

If you wish to use a standalone servlet container simply install the servlet container of your choice (e.g. a recent version of Apache Tomcat) and drop the war file `jmxConsole.war` into its webapps directory. The war file can be found in your JMS Grid installation in the directory `catalina/webapps`.

Configuring the Embedded Servlet Container

If you wish to use the embedded servlet container you need to perform the following steps:

- Install a Tomcat 4.0.2 servlet container and set the environment variable `CATALINA_HOME` to the directory where this is installed. Note that the embedded servlet container will only work if this specific version of Tomcat is installed.
- Set the environment variable `JMXCONSOLE_HOME` to the location of the JMS Grid management console support files. These can be found in your JMS Grid installation in the directory `catalina`. By default the value `%JMSGRID%\catalina` (Windows) or `$JMSGRID/catalina` (UNIX) will be used so you only need to set this variable if you move these files to another location.
- To start the embedded servlet container, simply supply the argument `/c` when using the `startserver` command to start a daemon. By default this uses port

8080. You can specify which port will be by using the argument /c <port>. For more details see the JMS Grid User's Guide.

1.5 Upgrading a SpiritWave 6 Installation to Work with JMS Grid 5.1.2

1.5.1 Upgrading the Product Installation

If you already have a SpiritWave installation and wish to upgrade it to use JMS Grid, first backup your SpiritWave installation and then follow these steps:

- Install JMS Grid into a suitable directory.
- Move all property files (*.properties) from the old SpiritWave installation to the new JMS Grid installation.
- Move the wdir directory from the old installation to the new installation.
- If you are using the default settings for the admin store, which is to use a local file-based JNDI store, then you will need to move the jndi directory from the old installation to the new installation.
- Upgrade the message and admin stores as described below:
  - When you are happy the new installation is working, you can delete your old SpiritWave installation.

1.5.2 Upgrading the Message Store

The way in which the subscriptions associated with network connections are named has had to be changed in JMS Grid. This is because it causes problems with the latest versions of the Java Management Extensions. If you have network connections in SpiritWave you need to upgrade your message store to rename these subscriptions.

What You Need To Do

There is a tool, called storeupgrade (UNIX) or storeupgrade.bat (Windows), in the root of your JMS Grid installation. You can use this tool to convert a SpiritWave message store so that it works with JMS Grid or revert back from a JMS Grid format to SpiritWave.

Usage:

```
storeupgrade [-S | [-J] ] <data-directory>
```

-J: convert a SpiritWave store to a JMS Grid store. This is the default direction so this flag is not obligatory.

-S: convert a JMS Grid store to a SpiritWave store

The <data-directory> value is the directory in which all the message store files reside, so for a daemon named Daemon1 this would be $SPIRITWAVE/wdir/data/Daemon1. Then, to run the update on this store you would issue the command, on Unix:

```
storeupgrade $SPIRITWAVE/wdir/data/Daemon1
```

and on Windows:

```
storeupgrade %SPIRITWAVE%\wdir\data\Daemon1
```

As the default is to convert from SpiritWave to JMS Grid you do not need to specify a flag.

1.5.3 Upgrading the Admin Store

The format in which users are stored in the admin store has changed. This means that if you are using JMS Grid security and have used SpiritWave 6.1.3 or earlier to define users in the admin store you need to upgrade the store as described here.

Why the Format of a User has Changed

Earlier versions of Sun JMS Grid, then known as SpiritWave, used serialization to store the public key attributes of users. However, we have now changed this to use the preferred key encoding method, which does not have any dependency on the concrete class which a particular JRE might use to represent the public key. We have also changed aspects of the key generation process.

When the user is updated as described in the next section it is important to note that the password is reset at the same time, to be the same as the user name. We strongly encourage you to ensure these are then changed to something more secure as soon as possible.

These changes have come about because we no longer need to support JRE version 1.1, which did not have key encoding, and also so that security would work on all supported platforms. However, note that there are still some cross platform issues: the *Sun Java Message Service Grid Release Notes*.

Upgrading Users: What You Need to Do

There is a key conversion tool, called `keyconvert` (Unix) or `keyconvert.bat` (Windows) in the root of your JMS Grid installation. You can use this tool to convert a SpiritWave admin store so that it works with JMS Grid.

Usage:

```
keyconvert [-S |[-E] ]
```

`-E`: Convert users with serialized keys, from earlier versions, to encoded keys. Change user's password to the same as the user's name.

`-S`: Convert users with encoded keys back to serialized keys. This preserves the password to the same as it was in the upgraded store.

If no arguments are given, the script will assume key encoding, i.e. as if `-E` were specified, is to be done.

Note: *The script uses the JRE specified by the `JAVA_HOME` environment variable and not the JRE which is distributed with JMS Grid. This is because you MUST use the same JRE to do the conversion as was used to create the users in the first place. If you see exceptions such as `StreamCorruptedException` being reported it is likely that you are using the wrong JRE.*

After running this tool to upgrade the admin store, you should use the `cpass` command to change the password of each user to something more secure than their user name.

1.6 Sun Microsystems, Inc. Web Site

The Sun Microsystems web site is your best source for up-to-the-minute product news and technical support information. The site's URL is:

http://www.sun.com

1.7 Documentation Feedback

We appreciate your feedback. Please send any comments or suggestions regarding this document to:

CAPS_docsfeedback@sun.com

1.8

Chapter 2

Architecture Overview

2.1 System Components

Any message-based application that uses JMS Grid will be composed of a number of fundamental components:

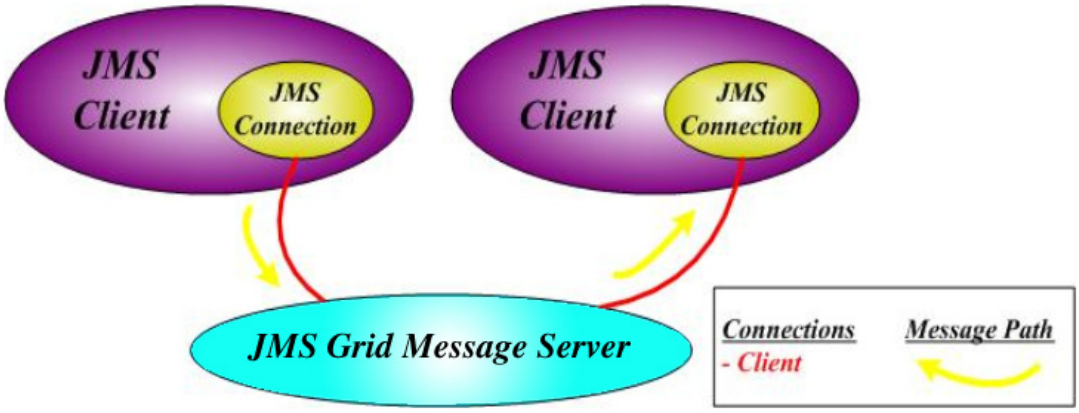
- JMS Grid clients – the physical end points from which messages are sent and received. A client could be a standalone Java application, or it could be an EJB, servlet or JSP running in an application server.
- JMS Grid server – a network of inter-connected daemon processes able to relay messages between connected client processes.
- Connections – these exist between clients and daemons, and between daemons.
- Destinations – the logical destinations from which messages are sent and received by client processes.
- JMS Grid message store – a database of messages used to guarantee the delivery of messages on failure.

These are described below.

JMS Grid clients

A JMS Grid client is simply a Java application process that creates a JMS client connection to a JMS Grid server. This is a socket connection which enables the client process to send and receive messages to/from the JMS Grid server remotely. Many client processes can connect to the JMS Grid server, enabling the sending of messages between clients via the server.

Figure 1 Two JMS Grid Clients Connected to a JMS Grid Server



JMS Grid Server

At the center of any JMS Grid messaging system is the JMS Grid Server. This consists of a network of one or more JMS Grid daemon processes. A JMS Grid daemon is the fundamental building block of the JMS Grid server, responsible for managing client connections and the delivery of messages between clients.

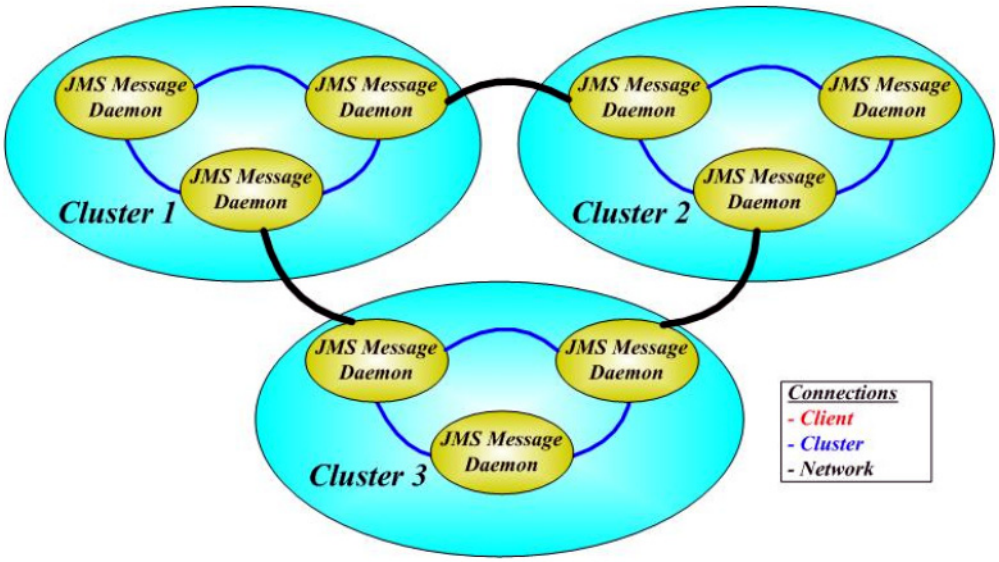
Each client is connected to a specific daemon with a socket connection. Messages are routed from the sending client, via any number of daemon processes in the server, to one or more receiving clients. Message routing is managed by JMS Grid server and is automatic. A client does not know where a recipient is located.

Networked messaging daemons can be configured into **clusters** and **networks**.

A **cluster** is a tightly coupled collection of daemon processes where all daemons are inter-connected. Client connections are spread across the available daemons and all message data is shared to provide fault tolerance.

A **network** is a loosely coupled collection of clusters where specific daemons are connected between clusters and only messages required for delivery to a client on a secondary cluster are sent between clusters. Clusters and networks together provide scalability and fault-tolerance.

Figure 2 Three Cluster - Three Daemon JMS Grid Server



Connections

Messages are transmitted from the sending client to the receiving client via a series of daemon processes and inter-process connections. There are three types of connection in a JMS Grid system:

- Client connection – connects client and demon processes
- Cluster connection – connects daemons within a cluster
- Network connection – connects daemons between clusters

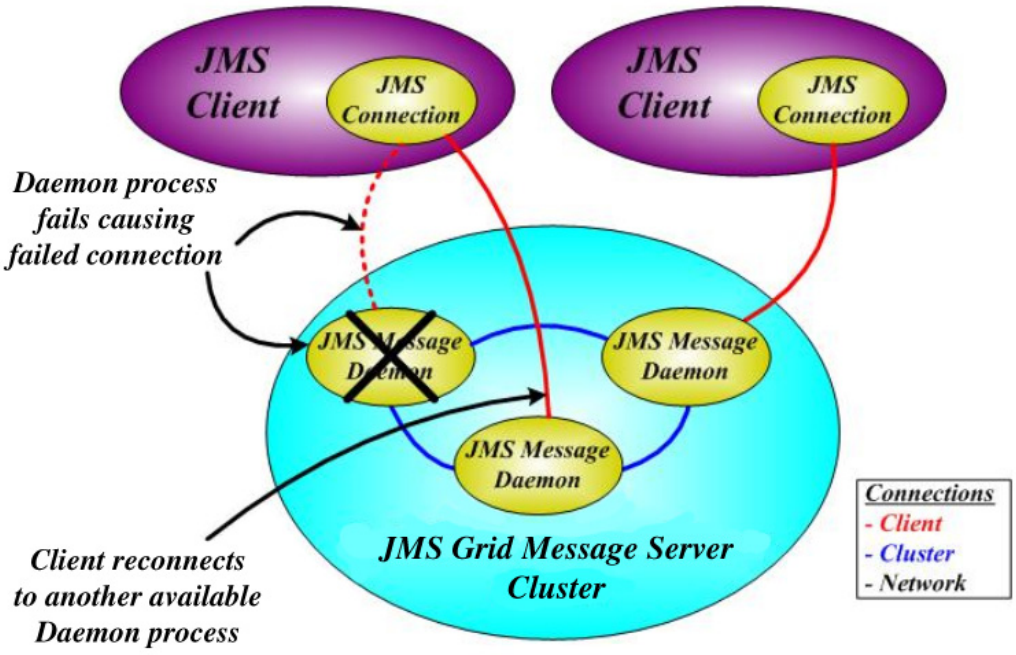
All inter-process connections are socket based and support the following protocols:

- TCP
- HTTP
- SSL
- HTTP/S

Connections have support for load balancing and failover. Load balancing spreads the client connections across the available daemon processes for performance and scalability. Failover enables any connection to automatically reconnect to another available resource when the connection is lost.

The following diagram shows a cluster of three daemons. If one of these daemons fails then any clients connected to that daemon are automatically reconnected to another daemon in the cluster. This is called **client connection failover**.

Figure 3 Client Connection Failover



Destinations

A destination a logical concept defined by the JMS specification. Clients do not send and receive messages to and from each other directly. Instead they send and receive messages to and from destinations. This allows each client to operate without knowing about the other clients.

The JMS specification defines two types of destination – queues and topics:

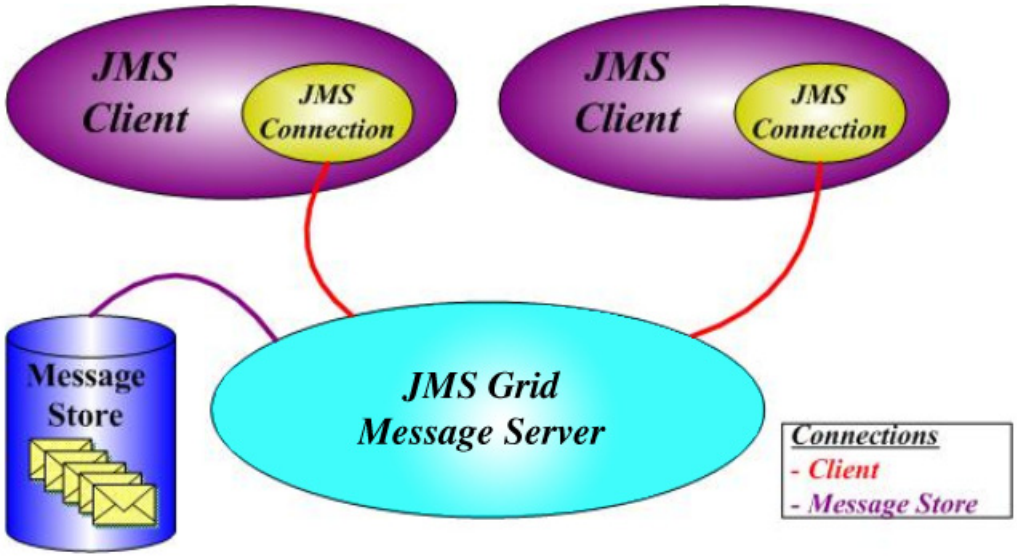
- With queues, each message is delivered to one and only one recipient.
- With topics, each message is delivered to every receiver subscribing to that topic.

The JMS Grid server is responsible for establishing the physical path between clients and destinations.

Message store

To provide guaranteed delivery of messages in the event of a system failure, messages may optionally be saved to persistent storage. JMS Grid uses a persistence mechanism known as the JMS Grid message store. This is a proprietary file-based message store optimized for use within a messaging system. It is highly scalable and can handle millions of messages.

Figure 4 JMS Grid Message Store



2.2 Distributed Topologies

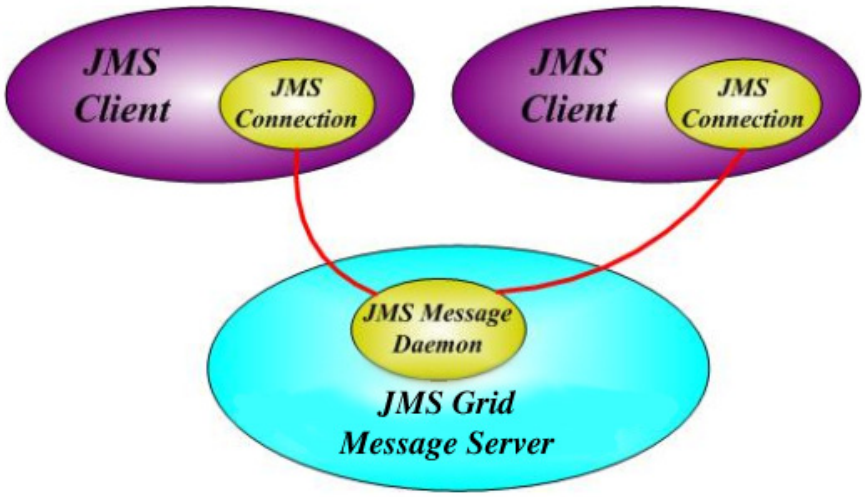
JMS Grid supports a variety of server topologies:

- A single daemon
- A cluster of multiple daemons
- A network of multiple clusters

Single Daemon

A single daemon configuration will have client connections only, with all messages routed through the daemon, as illustrated below. This configuration will exhibit minimal latency between sender and receiver, but will have limited scalability and limited fault tolerance.

Figure 5 Single Daemon

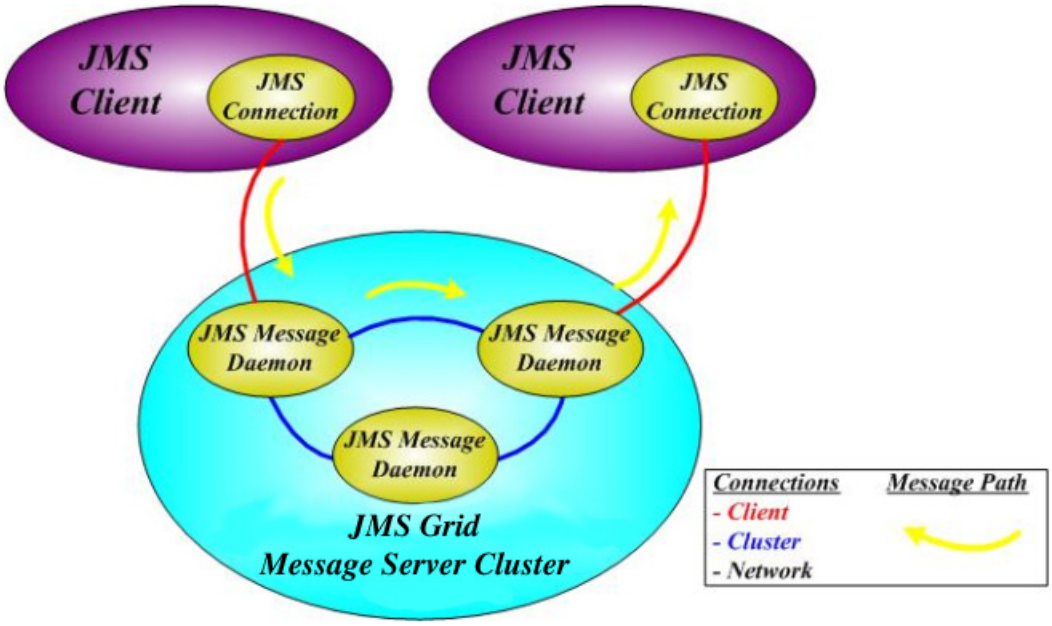


Cluster of Multiple Daemons

A cluster is a group of inter-connected JMS Grid daemons which collectively manage client connections and the routing of messages from sender to recipient. Clients can connect to any of the daemons within the cluster.

The connections between daemons are known as **cluster connections**. All daemons within a cluster are connected to all other daemons within that cluster, and so are able to route messages directly to any daemon within that cluster.

Figure 6 Cluster of Three Daemons



Daemons within a cluster collaborate to optimize message routing across the cluster. In order for the cluster to determine where messages should be routed, subscription information is shared between daemons across the cluster connections. Complete location transparency is maintained; applications need only know the logical destinations. Physical routing is established by the daemons themselves. For non-persistent messages, message routing across the cluster is optimized. A message will only pass through a maximum of two daemons before arriving at its destination. If the sending and receiving clients are connected to the same daemon, only one daemon will be involved.

For fault tolerance and resilience message replication of persistent messages occurs between daemons within a cluster. When a persistent message is received by a particular daemon from a client, the message is replicated around the cluster. In the event of failure of the receiving daemon, the remaining daemons in the cluster will assume responsibility for delivery and the client connection will failover to another daemon in the cluster. On reconnection the client will synchronize state with the cluster and resume. On restarting the failed daemon, synchronization will take place between the clustered daemon processes to establish the current state of the queues before they resume normal operation.

Clusters can be configured across multiple machines for performance and fault tolerance. A multiple machine cluster introduces greater levels of isolation into the configuration. Such a configuration will have greater resilience to severe hardware failures. As messages and subscription information are shared across the cluster, there is no single point of dependency. A machine may fail completely and the cluster will continue to function with no message loss and no interruption to service. Clients will

failover their connections to the remaining daemons on the working machines which will assume responsibility for message delivery as described above.

Clusters add resilience, fault tolerance, and scalability to the JMS Grid architecture:

- Resilience and fault-tolerance are provided through process redundancy, message and subscription replication, daemon and client synchronization and connection failover, which prevent there being any single point of failure.
- Scalability is provided by the ability to add as many message daemon processes as is necessary.

Note: *Some versions of JMS Grid may have limitations in the number of daemons that a cluster may contain. Please see the product release note for details.*

Multi Cluster Networks

Clusters can be connected to each other to form **networks** of clusters. These inter-cluster connections are known as **network connections**. Clusters can be configured in any topology, including hierarchical and full network topologies.

Networks differ from clusters in that they are loosely coupled. The daemons that form a cluster are all inter-connected; there is a connection between every pair of daemons in the cluster. All subscriptions are shared and all persistent messages are replicated between the daemons in the cluster.

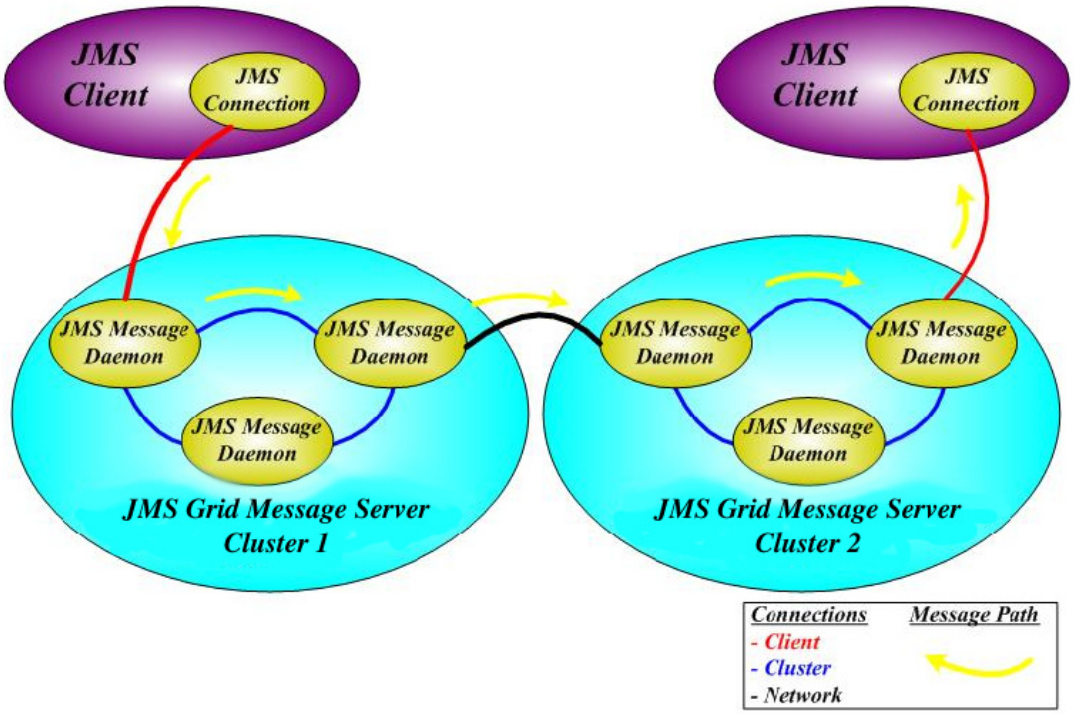
If two clusters are connected to form a network, however, a connection is only required between one daemon in the first cluster and one daemon in the second cluster.

Messages are only routed between two clusters when a message sent by a client on one cluster needs to be delivered to a client on the other cluster. Network connections, therefore, typically carry less traffic than cluster connections, making them suitable across slower infrastructure such as a WAN.

Furthermore network connections can withstand the connection going down for a period of time, as is sometimes the case with a WAN. In such situations messages are simply stored on the sending cluster, ready to be forwarded when the connection is reestablished.

Finally network connections are fault tolerant: if one of the daemons at either end of a network connection fails then the network connection will automatically failover to one of the remaining daemons in its cluster.

Figure 7 Simple Network of Two Clusters



Networks provide the means to deploy highly scalable and fault tolerant configurations. Dynamic subscription manages the physical routing of messages, preserving transparency, enabling dynamic routing and failure routing to occur while optimizing message flows across any network topology.

2.3 Architecture

Destinations and Dynamic Subscription

When a client registers interest in receiving messages from a particular destination this triggers a subscription event which is automatically propagated to the other daemons, both those within the local cluster and those in remote clusters across the network. The propagation of subscription events is known as dynamic subscription.

Similarly, when a client begins to send messages to a JMS destination (a topic or queue) this message production event is automatically propagated to all daemons in a similar way.

Message Routing

Dynamic subscription enables a JMS Grid network to dynamically create message paths across any daemon topology. As subscriptions change, so the daemon network will propagate those changes and re-configure the message routing. New message paths are established when new subscriptions are taken out.

If the topology of a network changes due to a daemon failure, the message paths will automatically adapt and find alternate routes. The failure of daemons will be transparent with no loss of service. Connection failures are also handled transparently.

When a connection breaks for any reason, the connection failover mechanism is activated. This applies for all types of connection: client connections, cluster connections and network connections. In all cases, when connection failure is detected, the client or daemon automatically detects the failure and searches for an appropriate alternative daemon to connect to. When the alternative connection has been reestablished then new message paths are established and message delivery continues as before. This occurs without any error or exception being thrown to the client.

The JMS Grid network is responsible for the delivery of messages and will determine the optimal message route across the available network of daemons. Specific message routing across networks can be configured via network filters as described below.

Subscription Propagation

When a daemon receives a subscription from a client it will be automatically forwarded to the other daemon or daemons in the cluster. If the cluster is connected to another cluster via a network connection, then the subscription will also be propagated to each connecting cluster and to all daemons within those clusters. This propagation of subscriptions enables JMS Grid to establish routing dynamically across any network topology.

Each daemon thus knows about subscriptions for every daemon in the network:

- All clients connected to it (client connections)
- All daemons in its cluster (cluster connections)
- All clusters connected to it via network connection daemons (network connections)

The only exception to this is if a network filter has been configured to control the propagation of messages across a network connection.

Network Filters

The propagation of messages across a network can be controlled via **network filters**. By default all messages are propagated to remote clusters. A network filter explicitly controls the propagation of subscriptions and hence messages across a network connection.

A network filter can either refer to a specific named destination, e.g. “Topic.SubTopic1”, or use wildcards in a hierarchical destination name, e.g. “Topic.*”, to apply filtering to a broader range of destinations.

By placing a filter on a network connection, only messages for the destination declared in the filter will be propagated between the clusters. Similarly filters with wildcards will only allow messages through that are sent to a destination within the name range specified by the wildcard. Many filters can be applied concurrently allowing explicit control of the message flows between clusters.

The ability to control inter-cluster traffic provides the means to optimize message flows across cluster networks providing greater scalability.

The ability to control the visibility of destinations across a network connection and thereby control the associated message flows provides data partitioning, enabling "Chinese walls" to be maintained between one part of a network and another.

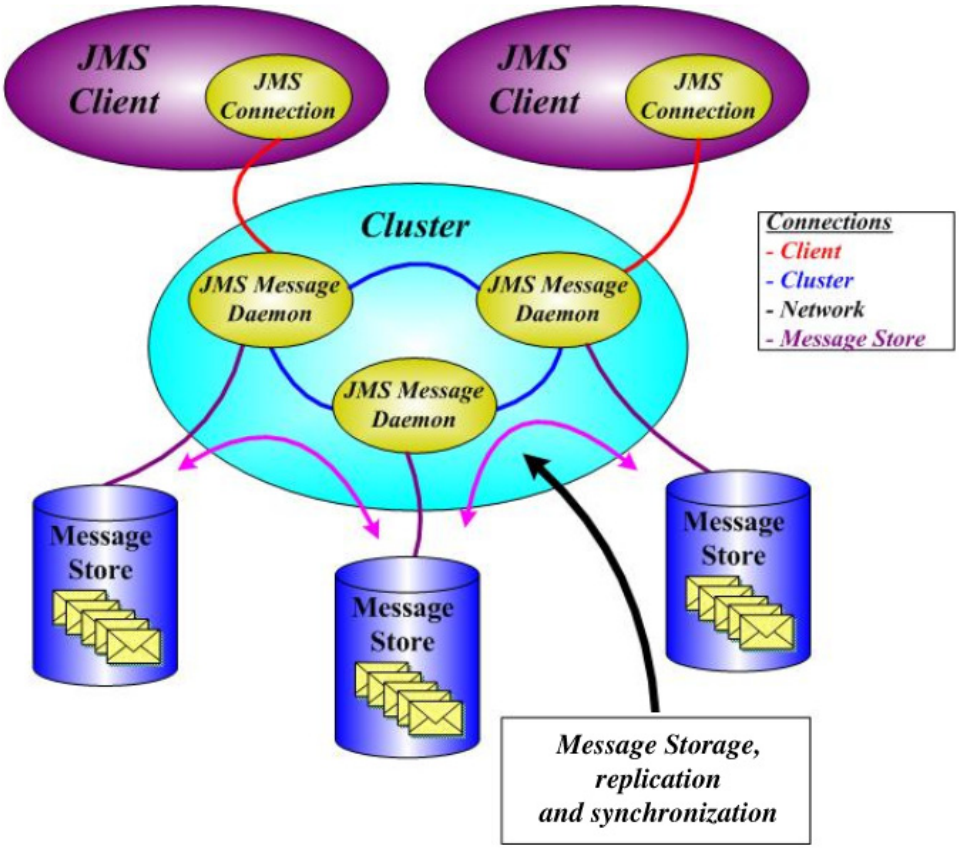
Message Persistence

Each JMS Grid daemon has a message store which is used to store persistent messages sent to queues and to topics on which there are durable subscriptions.

The message store is a file-based data store optimized for use within a messaging system. This provides better performance than would be offered by a generalized relational database. The message store is highly scalable and able to handle large volumes of stored messages.

Each daemon has its own message store. Changes made to one message store are automatically replicated across the cluster. This replication of data is what makes JMS Grid able to withstand the failure of individual daemons without any loss of service or messages.

Figure 8 Daemon Message Store - Replication and Synchronization



When a persistent message is received into the cluster from a client, the receiving daemon first persists the message in its local store. It then sends copies of the message to the other daemon or daemons in the cluster. When these other daemons receive the message they save it in their own message store and send an acknowledgement to the first daemon. A two-phase commit protocol is used to ensure that each daemon is updated consistently. When the first daemon has received acknowledgements from the other daemon or daemons it finally sends an acknowledgement back to the client.

If a daemon fails, one of the remaining daemons assumes responsibility for delivery of messages, using the messages in its local message store.

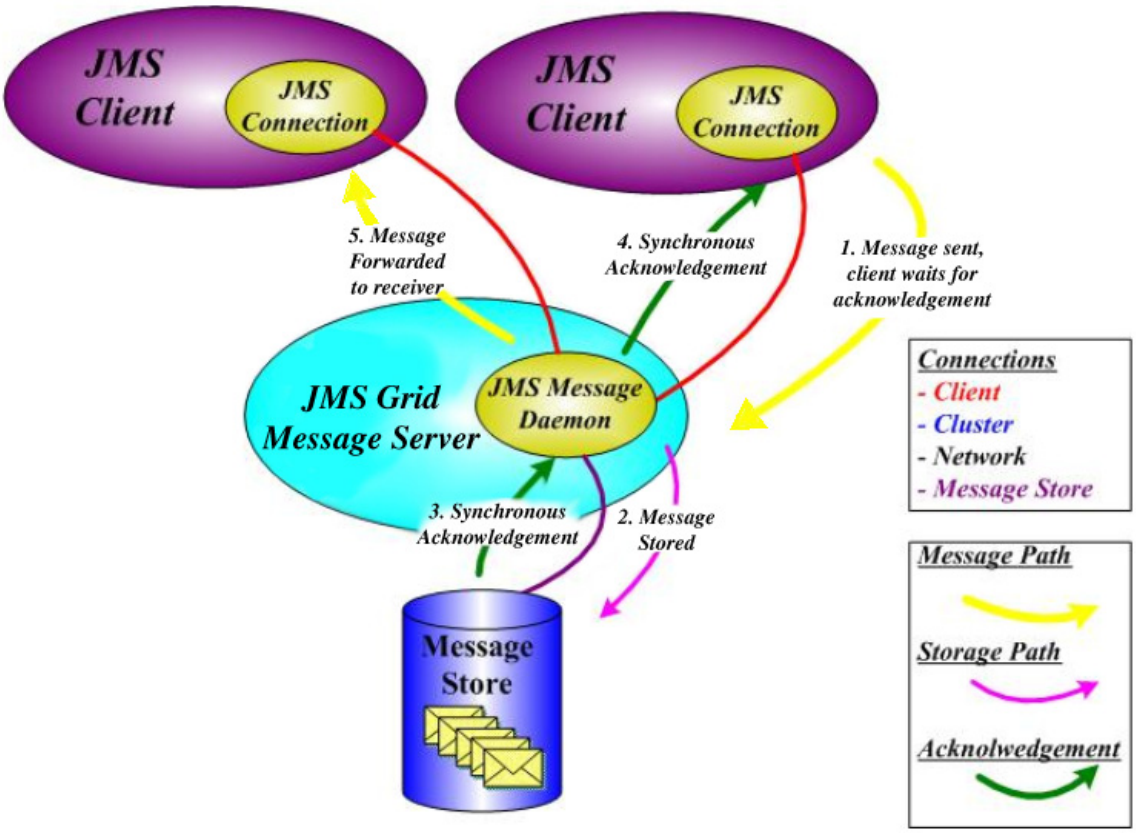
When the failed daemon re-starts and rejoins the cluster, a recovery process occurs during which the recovering daemon synchronizes its state with the rest of the cluster. Updates are sent to the recovering daemon from an up to date daemon in the cluster and stored in its message store. After recovery is complete the recovering daemon will contain exactly the same messages and subscription information as the other daemons just as it did before it originally failed.

Acknowledgement Model

JMS Grid provides guaranteed delivery of persistent messages between clients via a distributed configuration of server daemons. This is achieved via a synchronous acknowledgement mechanism between clients and daemons. When a message is sent the sending client waits for an acknowledgement from the daemon. The acknowledgement indicates the message has been persisted in the message store and is safe from system failure. At this point the client discards the message from its in-memory cache.

During the acknowledgement cycle the daemon persists the message in its recoverable message store and sends replicas out to all other daemons in the cluster. It then waits for all the other daemons to send back acknowledgements that they have received and persisted each message replica. The daemon then sends an acknowledgement back to the receiving client.

Figure 9 Acknowledgement Model



Chapter 3

Administration

3.1 Introducing the Administration Tool

This chapter describes how to configure a JMS Grid installation. It covers:

- How to use the administration tool to configure the particular system architecture of daemons and clusters that you require
- How to start and stop a JMS Grid daemon
- How to use the administration tool to configure users, groups and encrypted destinations
- How to use SSL
- How to use the administration tool to configure JMS connection factories and destinations

This chapter does not cover run-time monitoring or management. This is covered in **Chapter 5**, **Using the JMS Grid Management Console** on page 229.

This chapter does not cover how to fine-tune your system to achieve maximum performance. This is covered in**Chapter 6**, **Configuring JMS Grid for Fast Throughput** on page 259.

3.1.1 Starting the JMS Grid Admin Tool on Windows

All configurations for JMS Grid are performed through a single interface, known as the JMS Grid Admin Tool. To start the admin tool:

1. Open a command prompt and navigate to the root of your JMS Grid installation.
2. At the command prompt type: `C:\JavaCAPS51\JMS_Grid > admin`

Note: *A Windows command prompt appear and shortly afterwards, the Application Startup dialog box. After a few more seconds, the JMS Grid Admin Login dialog box will appear. If no dialogue box is seen, check under any other windows that may be open as it can easily be hidden under these.*

Figure 10 Logon Dialog

3 Enter your userName and password into the relevant text fields in the Admin Login dialog box. Click the Login button.

4 If this is the first time you have used the admin tool, or you are logging in as the default User, then enter the following values into the Username and password fields:

userName: **admin**

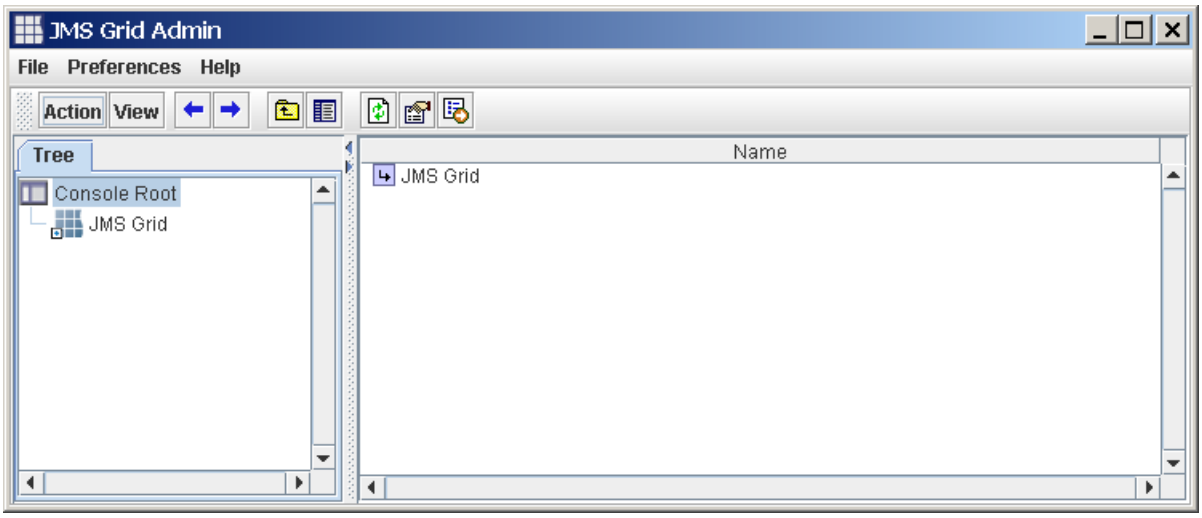
password: **admin**

You should change the default password at the earliest opportunity. This is explained in **Changing a User's Password** on page 111. You must be extremely careful not to forget the new password.

Note: *It is vitally important that you don't lose this password. All other passwords in the system can be changed if the original is forgotten. If the admin User's password is forgotten – and you have not created any other Administrators – then your position is irretrievable. You will have to re-create the admin store for your JMS Grid system and start all over again!*

The **JMS Grid Admin Tool** window appears.

Figure 11 JMS Grid Tool



If your userName was invalid, a valid userName's password was incorrect or your User did not have 'Administrator' permission, then the Error dialog shown below will appear. Click the OK button to close this Error dialog. To start the admin tool again, return to step 1 in this How To.

Figure 12 Logon Error

If you fail to Login three times using the same valid userName, then the admin tool will lock that User out. That userName will not be allowed to login again until the administrator has re-enabled the account.

3.1.2 Starting the JMS Grid Admin Tool on Unix

JMS Grid configuration tasks are performed using the JMS Grid Admin Tool. To start the admin tool from the Unix operating system:

1. Open a command shell and navigate to the root of your JMS Grid installation.
2. At the command line type: `C:\JavaCAPS51\JMS_Grid > admin`.
3. Login to the Admin Tool as described in steps 2 and 3 of **Starting the JMS Grid Admin Tool on Windows** on page 43.

3.1.3 Using the JMS Grid Admin Tool

This section gives an overview of the JMS Grid Admin Tool's Graphical User Interface. It introduces the GUI's main components and explains the principles you will need to understand in order to use it.

- Starting the JMS Grid Admin Tool in Windows
- Starting the JMS Grid Admin Tool in Unix

The figure below shows the JMS Grid Admin Tool GUI. The tree view is on the left hand side, the detail view on the right.

Menu Bar

The Admin Tool's menu bar contains three high level options that enable you to exit from the admin tool, specify various tool settings or obtain access to Help information. The majority of the Admin Tool's functionality is started from either the Action button on the Toolbar or a context sensitive pop -up menu over the Detail View.

Toolbar

The Toolbar provides a general mechanism for navigating the views, selecting display modes and providing shortcuts to the most common functions.

Tree View

The Tree View displays the hierarchy of configuration nodes and allows you to navigate around them. The navigation and manipulation of configuration nodes is central to administering JMS Grid. Detailed instructions on navigating around the Tree View are given in **Navigating the Tree View** on page 49.

Configuration Node

JMS Grid's configuration data is organized into a hierarchy of configuration nodes. JMS Grid is configured by creating configuration nodes and by setting their properties.

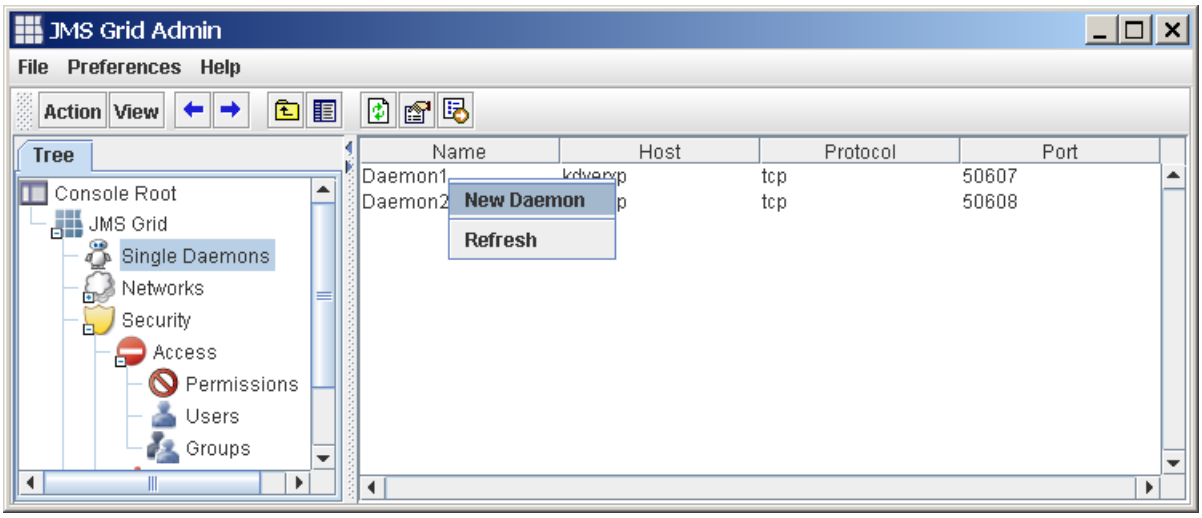
Detail View

The Detail View gives detailed information about the configuration node that is currently selected in the Tree View. The format of the information shown depends on the type of configuration node selected. For some nodes, the only information displayed in the Detail View is a list of that node's sub-nodes. For other nodes, the Detail View shows a table of the property values of that node's sub-nodes. The Detail Views allows you to query, modify, create and delete items using related context menus triggered by a right mouse button click.

Detail View Pop Up Menu

The Detail View pop up menu gives the administrator access to the majority of the Admin Tool's functionality. This pop up menu is opened by right mouse clicking in the Detail View. If an item in the Detail View is selected, then an Item Menu will pop up. The Item menu contains actions that can be performed on the item that is selected. If nothing is selected, in some cases, a Panel Menu will pop up. Panel menus allow you to perform more general tasks or create new objects in that Detail View.

Figure 13 Admin Tool GUI



3.1.4 About the Toolbar

The Toolbar is located directly under the Menu bar.

Figure 14 Toolbar

The purpose of the Toolbar is to provide a general mechanism for navigating the views, selecting display modes and to provide functional shortcuts.

Figure 15 Action Button

The Action button contains a context sensitive pull-down menu which is dependent on the admin object currently selected in the Tree View. For example, when the ‘Single Daemons’ node is selected, menu options include ‘New Daemon…’ and ‘Refresh’.

Figure 16 View Button

The View button contains a pull-down menu, which allows you to toggle between the detail and graphic viewing modes of the Detail View, such as graphics and detail. A graphic view is only available when certain nodes are selected – see How to switch between detail and graphical view.

Figure 17 Back Arrow

The Back arrow button will take the administrator from the current view to the previous view. This button will have an effect only if the administrator already clicked at least one view prior to triggering this option.

Figure 18 Forward Arrow

The Forward arrow button will take the administrator from the current view to the next view. This button will have an effect only if you already clicked at least one view prior to triggering this option.

Figure 19 Up-level Button

The Up-Level button will take the administrator from the admin object that's currently selected in the Tree View to its parent object. This button will have an effect only if the current node is not a root node in the Tree View.

Figure 20 Show-Hide Button

The Show/Hide button will hide the Tree View if it is currently showing and show the view if it is currently hidden.

Figure 21 Refresh Button

The Refresh button will trigger a refresh of any current Detail View. This is useful if the information that is being displayed was changed from the outside of this application, for example if an administered JMS object such as Destination has been removed from the JNDI directory by another User.

Figure 22 Properties Button

The Properties button will only work if there is an item selected in the Detail View. It will have the same effect as if the administrator selected an item, right-clicked a mouse and selected the Properties option from the Item Menu in the Detail View.

Figure 23 Export Button

The Export button enables you to export Detail Views in Comma Separated Values (CSV) File Format. These files may then be browsed or used for reporting in spreadsheets such as Excel. Once triggered, a file dialog will appear prompting the administrator for file name and location. For a more detailed explanation of this see **Exporting Configuration Data to a File** on page 142.

Figure 24 Help Button

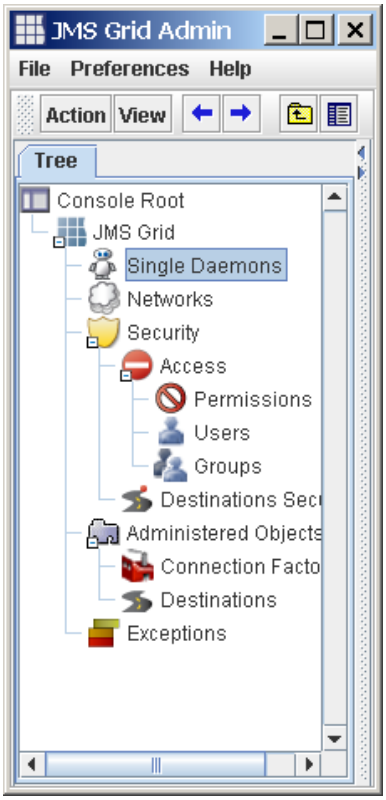
Shows this help file.

3.1.5 Navigating the Tree View

To be able to use the JMS Grid Admin Tool, you must be able to navigate around the configuration nodes in the Tree View.

The figure below shows a sample view of the configuration nodes in the Tree View.

Figure 25 Configuration Nodes in Tree View



Selecting a node

A node is selected with a left mouse click while the cursor is over either the node's icon, or the text to the right of that icon. The text of the currently selected node has a light blue background.

Types of Configuration Nodes

The Tree View displays three types of node:

Figure 26 No Node

The no node type contains no navigable sub nodes. Nodes of this type have no + or – symbol in their bottom left corner.

Figure 27 Closed Node

The closed node type has sub-nodes that are not currently displayed. Nodes of this type have a + (plus) symbol in their bottom left corner.

Figure 28 Open Node

The open node type has sub-nodes that are currently displayed. Nodes of this type have a - (minus) symbol in their bottom left corner.

Opening and Closing nodes

- A closed sub-node can be opened so that its sub nodes are displayed when you right or left-click it.
- An open node is closed when you right or left-click it.
- The right mouse button can be used for opening and closing a node without select.
- A left mouse click will both toggle the node and select it.
- Nodes with no navigable sub nodes cannot be opened or closed.

Buttons

Figure 29 Tree View

The Tree View maybe shown or hidden using the Show/Hide button, which is located on the tool bar.

You can easily re-trace your steps through the Tree View by using the Back Arrow and Forward Arrow buttons on the Toolbar.

Figure 30 Back Arrow

The Back arrow button will take you from the current view to the previous view. This button will have an effect only if you have already clicked at least one view prior to triggering this option.

Figure 31 Forward Arrow

The Forward arrow button will take you from the current view to the next view. This button will have an effect only if you have already clicked the back arrow button to take you to at least one previous view.

Figure 32 Up-level Button

You can navigate to the current node's parent by clicking on the Up-Level button. This button will have an effect only if the current node is not a root node in the Tree View.

3.1.6 Refreshing the Data that is Displayed

It is possible that the stored administrative objects are out of step with your view of them in the JMS Grid Admin Tool.

It is advisable to refresh the Admin Tool's view in the following circumstances:

- After editing network or cluster configurations.
- A Connection Factory's settings are modified from outside this tool
- The JNDI Store settings are changed.

Note: *Only the information for the admin object that is selected in the Tree View will be refreshed.*

To refresh the data

The Admin Tool provides two ways to manually refresh the view.

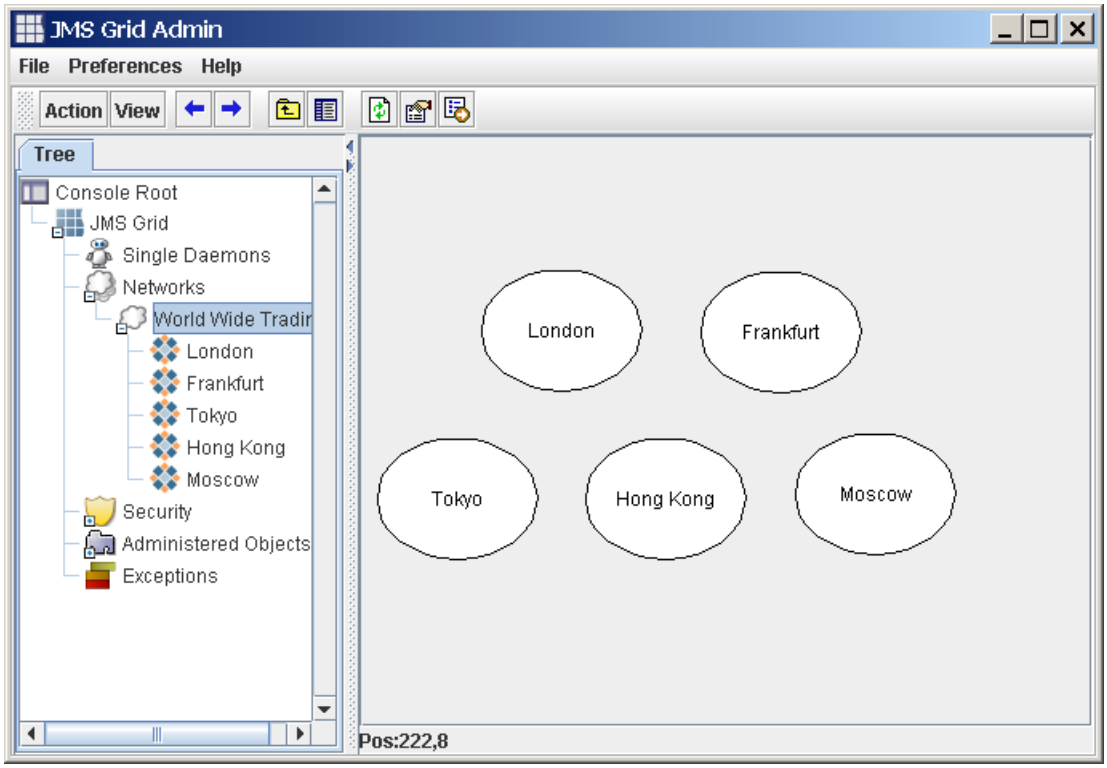
Figure 33 Refresh Button

- Click the Refresh icon on the Toolbar.
- Position your cursor over the Detail View, but not over any items in the view. Click the right mouse button to bring up the Panel Menu. For most Detail Views, the Panel Menu will contain a **Refresh** option. If the Detail View is full, right mouse click the Detail View's column headers to access the Panel Menu.

3.1.7 Toggling Between Detail and Graphical View

The JMS Grid Admin Tool allows you to view Networks and Clusters graphically as well as in tabular form.

Figure 34 View Networks or Clusters



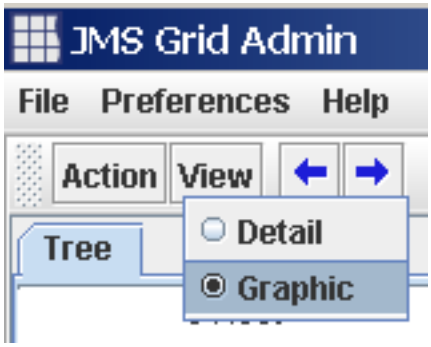
Prerequisites

- You should be familiar with **Navigating the Tree View** on page 49.
- You should be familiar with creating a network. See **Networks of Clusters of Daemons** on page 78.

To create a new cluster

1. Navigate to the network or cluster in the Tree View that you wish to view. All the networks that have been created are located under `Console Root > JMS Grid > Networks`. Cluster nodes are located under their network.

Figure 35 Detail and Graphic View



2. In the Toolbar left mouse click the View button. A pull down is displayed below the button. The pull down shows two options: Detail and Graphic.
3. From this pull down, select the view type that you want. You can only switch to a graphic view if you have a Network or Cluster admin object selected in the Tree View. When other admin object types are selected, the Graphic option is disabled.

3.2 Managing Single Daemons

3.2.1 What Is a Daemon?

A daemon is a program that runs continuously and exists for the purpose of handling periodic service requests that a computer system expects to receive.

JMS Grid messaging daemons are continually running programs that receive messages from message producing clients and perform the necessary routing to ensure that the messages are received by consuming clients. A JMS Grid server is made up of one or more such daemons.

JMS Grid daemons can be executed as single programs or grouped into clusters. This section covers the administration of single, stand-alone daemons. Single daemons offer a simple messaging solution. In this case, the JMS Grid server consists of a single process that only communicates with messaging clients and not with other daemons.

A single daemon JMS Grid server is often used for stand-alone application development and for basic evaluation.

A JMS Grid server can also consist of a number of inter-connected cluster daemons. Cluster daemons have many similarities with single daemons. Many of the How Tos in this section apply to both single and cluster daemons. Cluster daemons are explained in more detail in, **Creating a Configuration for a Cluster Daemon** on page 82.

3.2.2 Configuring a Single Daemon

This section describes the simplest way to use the JMS Grid Admin Tool to create a configuration for a single daemon.

Note: *This How To explains how you create a configuration for a single daemon. It does not tell you how to run a daemon that uses that configuration. How to run a daemon is explained in* **Starting a Daemon** *on page 56*.

Single daemons cannot be added to a cluster. If you want to create a cluster of daemons, you must create Cluster Daemons – see **Creating a Configuration for a Cluster Daemon** on page 82.

There are no fundamental differences between single daemons and cluster daemons. The only difference is that a cluster daemon has a Cluster ID. A cluster daemon then connects up with all the other daemons with the same Cluster ID.

It is also possible to start up a default single daemon that needs no configuration. This is explained in Starting a default daemon.

Prerequisite

- You should be familiar with **Navigating the Tree View** on page 49.

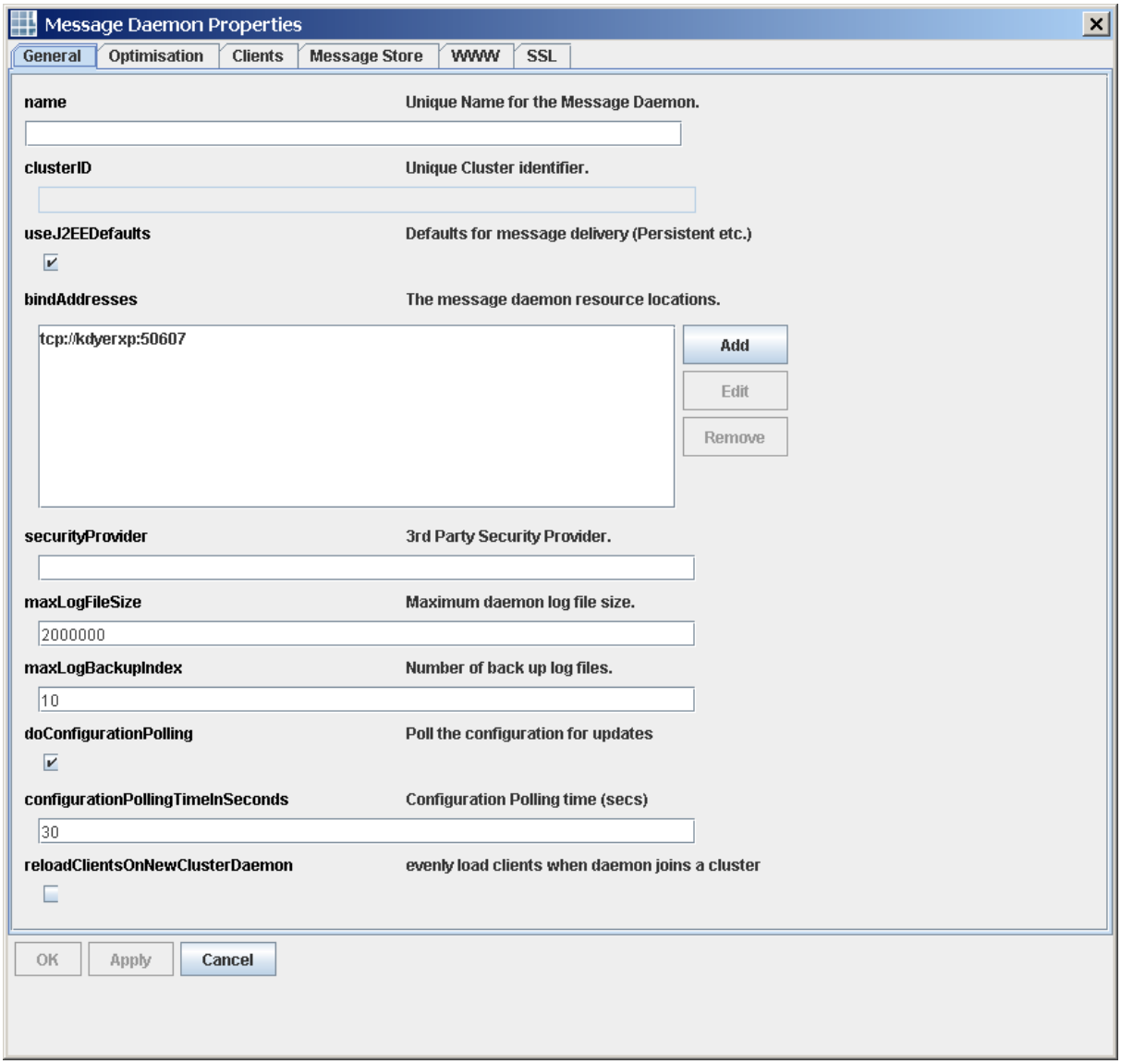
If you want to configure where your configuration data is stored, see **Specifying how Configuration Data is Stored** on page 139.

1. In the Tree View navigate to `Console Root > JMS Grid > Single Daemons`.
2. Right mouse click in the Detail View, ensuring that the cursor is not over any existing daemons that are displayed in the Detail View. This will bring up the Panel Menu. Alternatively, you can right mouse click over the Detail View column headings, or use the drop down menu off the ‘Action’ button on the Toolbar.

Note: *If the elements displayed in the Detail View fill the screen such that it is not possible to position the cursor away from them, detail, then right click the table heading in order to bring up the Panel Menu.*

3. From the Panel Menu, select New Daemon. This will open the New Message dialog box.

Figure 36 New Message Daemon



4 Enter a unique name for the daemon into the name field. You must only use alphanumeric characters for the daemon name. Do not use spaces.

Note: *The daemon name must be unique from the names of all other daemons within your configuration data store. You cannot give a single daemon the same name as one of your cluster daemons.*

5 Click the OK button. When a new daemon has been created, it is displayed in a row in the table in the Detail View.

Note: *The system will automatically generate a default network URL that the daemon will listen on. See* **Specifying a Daemon Network URL** *on page 63.*

The Cluster ID field is not used for single daemons.

See also

- **Specifying a Daemon Network URL** on page 63.
- **Creating a Configuration for a Cluster Daemon** on page 82.

3.2.3 Starting a Daemon

This section explains how to start JMS Grid daemons from the command line.

The daemon's configuration can

- match a specification defined using the Admin Tool
- match a specification in a property file
- be a default daemon that uses the default configuration.

This process is used for starting both single and cluster daemons.

The process outlined below only covers how to start a daemon that is executed on the same computer that stores the configuration data. If the daemon is being executed remotely from the configuration data machine, please read **Starting a Daemon on a Computer that is Remote From its Configuration Data** on page 70.

Prerequisites

If not starting a default daemon.

- You should be familiar with **Configuring a Single Daemon** on page 54.
- You should be familiar with how to **Creating a Configuration for a Cluster Daemon** on page 82.

The command used to start a daemon depends on which operating system you are using:

Windows

When starting up a daemon that follows a configuration specified in the JMS Grid Admin Tool, it is important to remember the daemon's unique name.

Note: *Be careful to remember the capitalization of the daemon's name.*

1. Open a Command shell window and navigate to the root of your JMS Grid installation: (`C:\JavaCAPS51\JMS_Grid`)
2. Use the startserver command to start the daemon using the following syntax:

```
startserver [/n daemonName] [/p propertiesFile] [/w workingDirectory]
[/s username password]
```

where

`/n` `daemonName` specifies the name of the daemon configuration to retrieve from configuration store – as created by following the instructions in **Configuring a Single Daemon** on page 54. Also see **Creating a Configuration for a Cluster Daemon** on page 82.

/p if specified, use the properties file `propertiesFile` rather than using the configuration in the Admin Store. Use of a property file is explained in more detail in **Configuring a Daemon from a Properties Text File** on page 75.

/w specifies the working directory where the data and logs directories are created and stored. The default is the wdir directory under your JMS Grid installation. workingDirectory is the name of the working directory.

/s if specified, enables secure operation of the daemon.

A valid username and password must be supplied (as explained in "All platforms".)

Unix

If you're starting up a daemon that follows a configuration you specified in the JMS Grid Admin Tool, you must remember the daemon's name.

Note: *Be careful to remember the capitalization of the daemon's name.*

1. Open a command shell and navigate to the root of your JMS Grid installation.
2. Use the startserver command to start the daemon using the following syntax:

   ```
   startserver [-n daemonName] [-p propertiesFile] [-w
   workingDirectory] [-s username password]
   ```

   where

   -n allows you to specify the name of the daemon.

   daemonName is the name of the daemon configuration to retrieve from configuration store – as created by following the instructions in **Configuring a Single Daemon** on page 54. Also see **Creating a Configuration for a Cluster Daemon** on page 82.

   -p if specified, use the properties file `propertiesFile` rather than using the configuration in the Admin Store. Use of a property file is explained in more detail in **Configuring a Daemon from a Properties Text File** on page 75.

   -w allows you to specify the working directory, where the data and logs are created and stored. The default is the wdir directory under your JMS Grid installation.

   -s if specified, enables secure operation of the daemon. A valid username and password must also be supplied (as explained in "All Platforms," below).

If you want to start up the daemons when the machine is booted, add the `startserver` command to the appropriate script in the `/etc/rc*.d` directories. Details of this will vary depending on the version of Unix that is being used.

All Platforms

If the daemonName you provide does not match up to a daemon configuration then coded default values will be used.

Only one daemon of a given daemon name can be started within the same JMS Grid server installation.

If you specify the –s or /s security option on the command line, it will enable security on the daemon, irrespective of the value of the 'enable security globally' property in the JMS Grid Admin Tool – see **Enabling JMS Grid Security** on page 99.

Username and Password

This username and password are used for authorizing network connections (that is connections between daemons in different clusters). In order that network connections can be established between daemons you must specify a valid existing username (and corresponding password). This username will have been previously set up in the Admin Tool – as explained in **Creating a User** on page 108. User permissions set here establish what is allowed to be done between the connected daemons. See **How to tell that a Daemon is using its Daemon Configuration** on page 72.

3.3 Starting a Default Daemon

You can start a single daemon without having to do any configuration at all.

- At the command line navigate to the root of your JMS Grid installation and enter:

```
startserver
```

If the –name option is omitted then a JMS Grid daemon is started with the default configuration. The daemon gets the name `<hostname>-50607`, where hostname is the name of the computer on which this daemon is being executed. The default daemon always binds to port 50607 on the host computer. Thus, you can only start a single default daemon per host.

3.3.1 Starting a Daemon with an Embedded Servlet Container

You can start a JMS Grid Message Daemon that includes an embedded instance of the Tomcat servlet and JSP container.

Note: *This does not work if you are using AJP or WARP connectors. This means that you cannot use an embedded Tomcat if you also wish to use it as a servlet engine for the Apache web server. In this case you must run it as a standalone server.*

Prerequisites

You need to have a Tomcat 4.0.2 installation set up. The environment variable CATALINA_HOME must be set to point to the root directory of that installation. The environment variable JMSCONSOLE_HOME must be set to the location of the JMS Grid management console support files. These can be found in your JMS Grid installation in the directory catalina. For more information see the JMS Grid installation instructions. The actual configuration of Tomcat is beyond the scope of this document: Tomcat comes with complete documentation describing the configuration process.

To start daemons with embedded servlet containers

You can start your daemon, whether configured or not, in the usual way. Only one extra argument, /c or –c, is needed to the command line.

On **Windows:**

```
startserver [/n serverName][/w workingDirectory][/s username
password]/c
```

On **Unix**:

```
startserver [-n serverName][-w workingDirectory][-s username
password]-c
```

Once you have started the daemon, it will handle servlets and JSPs in the usual way. No change is needed to the addresses used in the web browser.

3.3.2 Creating Multiple Copies of a Daemon Configuration

If you want to create many similar daemon configurations you could repeat the steps explained in **Configuring a Single Daemon** on page 54, many times over. However, this would be a laborious process.

The JMS Grid Admin Tool provides you with a much quicker way to do this – it allows you to make multiple copies an existing daemon configuration.

Prerequisites for single daemons

- You should be familiar with **Configuring a Single Daemon** on page 54.

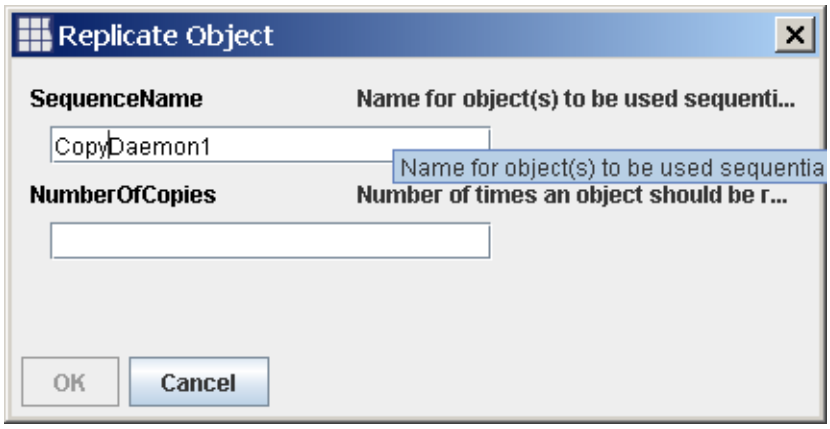
Prerequisites for cluster daemons

- You should be familiar with how to **Creating a Configuration for a Cluster Daemon** on page 82.

To create multiple copies of daemon configurations

1. Create a single configuration for a single daemon or a cluster daemon. Ensure that the configuration parameters of this configuration closely match the configurations of your copies before deciding to replicate it.
2. Select the daemon to replicate. Click the right mouse button to bring up the **Item Menu**. From the **Item Menu** select **Replicate**. This will open the **Replicate Object** dialog, as shown below.

Figure 37 Replicate Object



3 In the **SequenceName** field, enter the name to be used for the replicated objects. The Admin Tool applies a simple numbering rule to the names of each of the replicas. A sequential number, starting from 0, is appended to the end of the **SequenceName** you give to create the replica's name. For example, if you entered *MyDaemon* as the sequence name, and asked for 3 copies to be made, then the replicated daemons would be called *MyDaemon0*, *MyDaemon1* and *MyDaemon2*.

4 In the **NumberOfCopies** field enter the number of copies to make.

5 Click the OK button. In the Detail View list the sequence-named of replicas will appear, with incremented port numbers.

3.3.3 Deleting a Daemon's Configuration

This section describes how to remove a daemon's configuration from a JMS Grid configuration.

Deleting a daemon's configuration has no effect on a running daemon that is using that configuration. However, once a daemon with a deleted configuration has been stopped, it is not possible to start another daemon with that configuration.

Use the same procedure to delete both single and cluster daemons.

To delete daemon configurations

1 To delete a single daemon, in the Tree View, navigate to: `Console Root > JMS Grid >Single Daemons`.

To delete a cluster daemon, navigate to:

```
Console Root >JMS Grid >Networks <Network containing daemon's
cluster> <Cluster containing daemon>
```

2 In the Detail View, select the daemon whose configuration you want to delete.

Note: *To delete multiple daemons at the same time, you can select multiple daemons by holding down the Control key as you select each of the daemons you want to delete.*

3 With the mouse still over that daemon right click, and from the pop up Item menu that appears, select **Delete**.

4 Click **Yes** in the confirmation dialog that appears.

3.3.4 Stopping a Daemon

The recommended way to shutdown a daemon is to use either the `stopserver` or `sms` command.

The stopserver command can be found in the root directory of your JMS Grid installation:

```
C:\JavaCAPS51\JMS_Grid>stopserver
Running command: sms -connect tcp://localhost:50607,admin,admin -
context default.mymachine-50607
(logging messages omitted)
Shutdown operation complete
C:\JavaCAPS51\JMS_Grid>
```

The `stopserver` command can only be used to stop a daemon which is listening on the TCP port that was specified to be the default when JMS Grid was installed. This will be `tcp://localhost:50607` unless you specified something different.

If you wish to stop a daemon which is not listening on this protocol and port (e.g. if you are using a different port, a different protocol or if the daemon is on a remote machine) then you must use the sms (stop message server) command in the mgmt directory. The sms command allows you to specify the protocol and port to use to connect to the daemon:

For example, if the daemon is listening only on port 444 using the SSL protocol, use the following:

```
sms -connect ssl://mybox:444,admin,admin -context default.daemon1
```

For a full description of the `sms` command see **Shutdown Message Server - sms** on page 245. The `stopserver` and `sms` commands will be merged in a future version of JMS Grid.

You can also shutdown a daemon by typing Control-C in the console window or, on UNIX, by using the command `kill -TERM pid`

Note: *When a daemon is shutdown, non-persistent messages that are in transit will be lost.*

3.3.5 Editing a Daemon's Configuration

This section describes how to retrieve a daemon's configuration options for the purposes of updating specific items and not how to actually perform a specific configuration.

For a daemon to work you must give it at least a name and a network URL. See **Specifying a Daemon Network URL** on page 63.

For other configuration settings the default values need not be changed in order to configure a basic daemon.

Note: *Updating the configuration from the Admin Tool does not affect a running daemon; you must stop and restart the daemon before property changes will take effect. If you want to modify the behavior of a daemon while it is running, you must use the JMS Grid Runtime Management Console – see The JMS Grid JMX Management Guide.*

Prerequisites

- You should be familiar with **Configuring a Single Daemon** on page 54.
- You should be familiar with how to **Creating a Configuration for a Cluster Daemon** on page 82.

1. In the Tree View, navigate to the daemon you want to configure.

   Single daemons reside in `Console Root > JMS Grid > Single Daemons`

   Cluster daemons reside in:

```
Console Root > JMS Grid >Networks > <Network containing daemon's
cluster> > <Cluster containing daemon>
```

2. In the Detail View, right-click the daemon and click **Properties**. The **Message Daemon Properties** dialog box appears.
3. Perform the configurations that are required. Details of specific configurations are covered in individual sections. See the list below.
4. When finished and you want your configuration revisions accepted, click **OK** or **Apply** buttons, then click **OK** in the warning dialog that follows.

See also

- **Configuring a Single Daemon** on page 54
- **Specifying a Daemon Network URL** on page 63
- **Configuring a Daemon's Internal Queues** on page 66
- **Configuring Daemons to Actively Detect Network Outages** on page 67
- **Configuring Daemons to Automatically Close Connections to Slow or Frozen Clients** on page 67
- **Load Balancing Messages Across Cluster Daemons** on page 85
- **Enabling Auto Discovery** on page 86
- **Configuring Daemon Reconnections** on page 88
- **Configuring Message Filters on Inter-daemon Network Connections** on page 89

Note: *You can open an item's properties dialog by selecting it in the Tree View and then clicking on the Properties button in the Toolbar.*

3.3.6 Specifying a Daemon's Name

A daemon's name is used to match a JMS Grid daemon with:

1. A daemon configuration that was set up in the JMS Grid Admin Tool.
2. The name of the log file created when a given daemon is running.

Note: *The daemon name must be unique to the names of all other daemons your whole JMS Grid installation. You cannot give a single daemon the same name as one of your cluster daemons. Cluster daemon's names must be unique from daemons in all other networks and clusters in your installation.*

To specify daemon names

1. Follow steps 1 to 4 of, **Editing a Daemon's Configuration** on page 61.
2. In the Message Dialog Properties dialog, ensure that the General tab is on top.
3. Alter the name shown in the Name field. Although the input field will allow the entry of any character, you must only use alphanumeric characters for the daemon's name. Do not use spaces in the daemon's name.
4. Now complete step 4 of, **Editing a Daemon's Configuration** on page 61.

3.3.7 Specifying a Daemon Network URL

A daemon's network URL specifies the network location of a daemon and the protocol it uses to send and receive messages. Those messages could have been transferred to/from either a client or from another daemon.

A URL has the form:

```
<protocol>://<hostname>:<port>
```

It specifies the protocol to expect, the address to use (it must be an address local to the machine the daemon is running on) and the port.

Protocols

JMS Grid supports the following protocols, TCP, HTTP and SSL.

Table 5 Protocols Supported by JMS Grid

TCP Transmission Control Protocol	A set of rules used along with the internet protocol to send data in the form of packets between computers over the Internet. TCP takes care of keeping track of individual packets of data. TCP is used on the sending side to split a stream of data into packets, and on the receiving side to rebuild the packets back into the data stream.
HTTP Hypertext Transfer Protocol	A set of rules for exchanging files on the World Wide Web. HTTP is an application protocol. If a daemon is configured to use the HTTP protocol, them it will wait for HTTP requests and handle them when they arrive. Messages are given an http wrapper to enable http tunneling through firewalls.
SSL Secure Sockets Layer	A protocol for managing the security of a message transmission on the internet. It uses the program layer which is located between HTTP on the application layer and TCP on the transport layer. It uses the public-and-private key encryption system.

JMS Grid Message Daemons can support multiple network protocols concurrently. A single daemon could listen to TCP, http and SSL at the same time. You must define a Network URL for each protocol that daemon will understand. Thus, having more than one network URL enables a single daemon to talk and listen in multiple protocols. Each protocol must be associated with a unique hostname/port combination. Thus it would be invalid to have the following Network URL's in the same system:

```
TCP://localhost:3456
SSL://localhost:3456
```

Which protocol should you select?

The key consideration when selecting the protocol (s) to be used by a daemon is 'Which protocols do the messaging clients support?'. The protocols used by the daemon must match the protocol used by its clients - otherwise the clients communications will simply not be understood. If you have some flexibility over which protocols can be used by both clients and daemons, then you should take the following into account...

If you require certificate-based daemon and/or client authentication, or if you require on-the-wire encryption, then use SSL. (Note however that JMS Grid's security mechanism supports data encryption and user/password authentication and access control even across TCP and HTTP connections.)

If your firewall does not allow TCP or SSL traffic - use http.

If none of the above apply - use TCP. For point to point communication, TCP carries the lowest overhead and hence is the fastest and most efficient protocol option.

Hostname

Hostname must be an address local to the machine on which the daemon is running. It can either be the machine's physical IP address, or its name.

Port

The port is a numerical value, and represents the port that a daemon listens on for client connection requests. The port must not be in use by any other processes on that host. On Solaris and Linux, port numbers below 1024 are restricted to processes that have been started by the root User. Default values (should be in the same format as other defaults). The default network URL used by the admin tool for a JMS Grid daemon is:

```
tcp://<localhost>:50607
```

Note: *When daemons are replicated, port number of the original is incremented for each copy.*

Multihomed Machines

As mentioned above in the protocols section, a daemon can be configured to listen for client connections on more than one Network URL. As well as enabling a daemon to listen to multiple protocols, this also enables a single JMS Grid daemon to bridge more than one network domain for multihomed machines. Thus on a multihomed machine (a host that has more than one IP address) you can set up a single daemon to listen on each of that machines IP addresses.

Network URL, Bind Address, and Resource Location

Each of these terms is used in JMS Grid. They are all pseudonyms for the same thing.

Message Channels

This is the term used by a client when referring to the bind Address of a daemon running on a server.

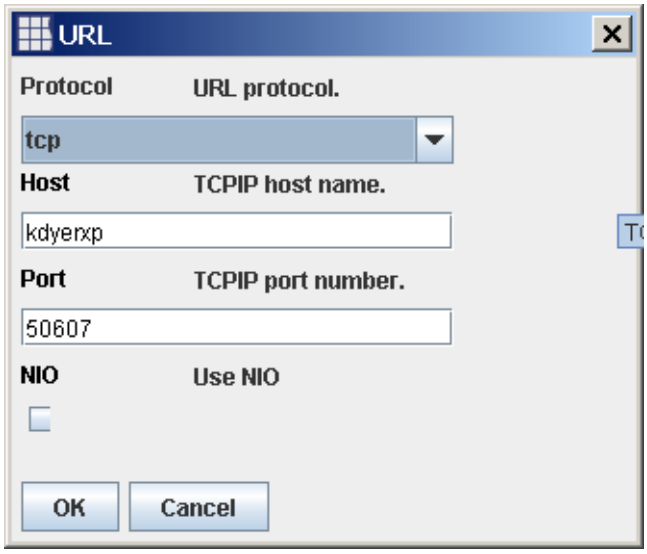
To set up daemon network URLs

1 Follow steps 1 to 4 of, **Editing a Daemon's Configuration** on page 61.

2 Click **Add** by the list of resource locations.

To edit an existing network URL, select the location you want to edit, then click the Edit button. The **NetworkURL** dialog box appears as shown below.

Figure 38 Network URL



Note: *The NIO option is not supported and the checkbox should be left unchecked.*

3 Fill in the desired protocol, host and port values before clicking the OK button. You must enter a valid host name and a port number that is not already being used by another daemon on that particular host.

4 To add multiple network URL's for a given daemon, repeat steps 2 and 3 above for each new network URL.

Systems With Multiple Network Cards

Some machines may have more than one network card, if they are being used as a bridge between two disparate network domains. By setting up more than one network URL for a daemon, it is possible for that daemon to listen for client connections on both network cards, e.g.

tcp://<Network1Name>:50607 and tcp://<Network2Name>:50607

3.3.8 Configuring a Daemon's Internal Queues

The JMS Grid Message Server uses internal dispatch queues for distributing messages to clients as efficiently as possible. This section explains how to customize the daemon's internal dispatch queues.

The following parameters are used to define the behavior of a daemon's internal queues:

Table 6 Daemon Internal Queue Parameters

Name	Description	Default Value
`maxInternalQueueSize`	The maximum memory size in bytes that this daemon's internal dispatch queues can use	33554432 (32 MB)
`maxTopicDispatchQueues`	The maximum number of dispatch queues used for distributing messages to Topic(s)	100

Internal Dispatch of Non-Persistent Messages to Queues and Topics

Messages that are non-persistent are placed on internal message queues within the daemon, awaiting dispatch to clients that have subscribed to the message's destination. When a message is received by the daemon, it is examined to determine which subscribers can receive the message. Message selection for a subscriber is done by destination name (where wild cards can be applied) and either message header or by content, depending on the type of selector used by the subscriber. Once a message has been tagged with the list of valid subscribers, it is placed to await dispatch, based on the priority of the message.

For messages sent to a Queue destination, an internal dispatch queue exists for each Queue name. For messages sent to a Topic, a dispatch queue per client is used up to the **maxTopicDispatchQueues** limit. When the numbers of clients who can receive the message on the Topic exceeds this, more than one client can share an internal dispatch queue.

A message producer can sometimes send messages at a faster pace than they can be consumed. This can result in messages having to be held by the daemon for a period of time until they are dispatched. The **maxInternalQueueSize** parameter defines the maximum amount of memory that can be allocated to all the internal dispatch queues. If you are expecting a large internal queue to build up, you may wish to increase the **maxInternalQueueSize** to make maximum use of the available physical memory on your hardware.

To Configure Internal Queues

1. Follow steps 1 to 4 **Editing a Daemon's Configuration** on page 61.
2. In the **Message Daemon Properties** dialog box, click **Optimization**.
3. Set the values of the **maxInternalQueueSize** and **maxTopicDispatchQueues** to the desired values.

3.3.9 Configuring Daemons to Actively Detect Network Outages

When using the HTTP, SSL and TCP protocols, the loss of a connection by network outage does not become apparent for some time.

A 'keep alive' or ping protocol can be used to actively detect network disconnects. This is controlled by a daemon's **pingEnabled** property.

When the **pingEnabled** property is set for a daemon, that daemon will check all of its connections for successful packet transmission.

If PingEnabled is checked and there has been no messaging traffic on a connection for the time specified in the **pingTimeout** property (milliseconds), a 'keep alive' message is passed between this daemon and its clients (a client could also be another daemon). If the message is not returned within pingTimeout * 2 msecs then the connection is deemed to have failed.

If the ping fails, the daemon assumes the client is dead and terminates the connection. If the client has not died (the connection might have been lost due to a network fault), then the client also determines that the daemon isn't available and attempts to connect to another daemon in the cluster before continuing to send messages.

Default values:

- **pingEnabled** property: False
- **pingTimeout** property: 5000

To configure daemons to actively detect network outages

1. Follow the basic steps in **Editing a Daemon's Configuration** on page 61.
2. In the Message Daemon Properties dialog box, click **Clients**.
3. To enable active network outage detection, make sure the Ping Enabled check box is ticked.
4. In the **Ping Timeout** field, specify how many milliseconds the daemon waits until it suspects that a connection has gone down, and send a 'keep alive' message.

Note: *JMS Grid Connections can be individually configured to check for network outages – see* **Editing Connection Factory Properties for a Normal JMS Grid Client Connection** *on page 131*.

3.3.10 Configuring Daemons to Automatically Close Connections to Slow or Frozen Clients

Clients that are slow consumers or frozen are referred to as blocked clients. It is important for the daemon to actively monitor blocked clients since they can have a negative impact on the performance of the whole messaging server.

Every connection is monitored for blocking. If a connection is blocked for more than `timeSocketBlockedBeforeClosing` milliseconds, the daemon will close the connection to that client. A client can be perceived as frozen due to either a client application error or a network error. If a client loses its connection to a daemon it will seamlessly fail over to another daemon in the cluster.

Default value:

`timeSocketBlockedBeforeClosing` property: 30000 milliseconds

To configure daemons to automatically close connections to slow or frozen clients

1. Follow the basic steps in **Editing a Daemon's Configuration** on page 61.
2. In the **Message Daemon Properties** dialog box, click **Clients**.
3. In the `TimeBlockedBeforeClosing` field, enter the time (in milliseconds) allowed for a connection to be blocked before it is closed by the daemon.

3.3.11 Configuring the Daemon's Message Store

Message persistence is an important part of any fault tolerant messaging system. JMS clients can specify that messages be sent in either Persistent Delivery Mode or Non Persistent Delivery Mode.

When messages are sent in Persistent Delivery Mode, the message server must ensure that messages are not lost if the server fails. The server does this by storing undelivered messages in a message store. Once a message is delivered to all intended durable subscribers, it will be removed from the message store.

To Configure the Daemon Message Store:

1. Follow the steps in configuring daemons.
2. In the Message Daemon Properties dialog, bring the **Message Store** tab to the top.
3. See below on how to fill out properties for the message store.

Table 7 Message Store Settings

Name	Description	Default Value
`marWindowSize`	Maximum number of unacknowledged messages that can have been dispatched from a particular queue or durable subscription before the daemon will cease sending further messages. (Note: A MAR is an internal "Mark As Read" message).	100
`spiritDbDispatchThreads`	The maximum number of dispatch threads for the message store	10
`SpiritDBMaxFileInMbytes`	Set the maximum size of the message store (in Mbytes).	512
`spiritDBThrottleThresholdPercentage`	When the message store reaches the Threshold percentage of the Max Size the daemon will start throttling publishers. The rate is determined at which messages are consumed, not more than the maximum (as a percentage).	80
`maxThrottleTimeout`	This will be the maximum rate publishers will be throttled (in milliseconds).	1000

Table 7 Message Store Settings (Continued)

Name	Description (Continued)	Default Value
`dataBlockSizeInBytes`	size of message blocks in the message store. Can improve performance on some platforms	10485760
`indexBlockSizeInBytes`	size of message blocks in the message store. Can improve performance on some platforms	1048576
`useDataBackup`	If true, old data blocks will be moved to a backup directory rather than be deleted	
`dataBackupDir`	Absolute path of backup directory for data blocks	unset
`useSync`	If set to true, daemon will Sync file after each message write.	false

3.3.12 Preventing JMS Grid From Failing

Some of the things that can go wrong in a computer system are:

- any piece of hardware
- any piece of software
- any communications link among hardware

That is, there is the possibility that anything in a computer system could go wrong at some point.

This section does not itself tell you how to prevent JMS Grid from failing. Instead it points you towards other sections in this user's guide that explain the steps you can take to avoid failure.

Hardware failure

Distribute daemons across hardware platforms. See **Networks of Clusters of Daemons** on page 78.

Software failure

If a daemon fails, ensure that there are other daemons that can take over. See **Creating a New Cluster** on page 80.

Ensure that if software fails, then messages are recoverable. See and **Configuring the Daemon's Message Store** on page 68.

Avoid delivering messages to slow or frozen clients.

Network failure

See **Configuring Daemons to Actively Detect Network Outages** on page 67.

3.3.13 Starting a Daemon on a Computer that is Remote From its Configuration Data

JMS Grid allows you to start JMS Grid daemon on a system that is remote from the system that is storing its configuration. Thus, for a widespread network of daemons, you can choose where the configuration data for that system is stored.

This is dependent on the level of access your remote computer has to the computer that's storing the configuration data and the type of configuration storage that you have told JMS Grid to use. It is very important that you choose a storage type that matches your requirements before you begin to store your system configuration in it.

Prerequisites

- You should be familiar with the JMS Grid Admin Tool.
- You should know how configuration data is stored.

For these basic steps, the system on which the configuration data resides is called the 'Configuration Data computer' and the machine on which the remote daemon will be running is called the 'Daemon computer'.

On the Configuration Data Computer

1. JMS Grid provides links into many different mechanisms that you can use to store the configuration data. Make sure the Admin Store mechanism that you are using allows that configuration data to be accessed from the Daemon computer. This is explained in more detail in **Deciding which Type of Configuration Data Store to use** on page 144.
2. Create a configuration for the daemon that is to run on the Daemon computer. Ensure that the Resource Location URL (also known as the daemon's network URL) is set to the name or IP address of the Daemon computer.

The following sections describe how to configure a daemon:

- **Configuring a Single Daemon** on page 54
- **Creating a New Cluster** on page 80
- **Specifying a Daemon Network URL** on page 63

On the daemon computer

1. Install JMS Grid on the Daemon system.
2. Follow the instructions given in **Configuring JMS Grid from a Remote Machine** on page 71 so that on the daemon system you can access the configuration data on the configuration data computer. When you set up the JMS Grid Admin Tool to access the configuration data on the configuration data computer some of the configuration settings on the daemon computer are altered so that any daemons that are run on the daemon computer can find the remote configuration.

Note: *If the daemon is using any JMS Administered Objects that are stored in JMS Grid's JMS Admin Object Store, then you must also set up the JMS Grid Admin Tool so that it accesses the remote JMS Admin Object Store.*

3 On the daemon computer, start the daemon using the procedure in **Starting a Daemon** on page 56. Set the value of the /n option to that of the daemon configuration you created in step 1 of this section.

4 Check to see that the daemon is using the correct configuration by following the instructions given in **How to tell that a Daemon is using its Daemon Configuration** on page 72.

3.3.14 Configuring JMS Grid from a Remote Machine

The JMS Grid Admin Tool can be run from a computer that is remote from the machine that stores your JMS Grid configuration so long as that remote computer has access to the place where the configuration data is stored.

If you require remote access to configuration data then the most common mechanisms for storing configuration data are one of the JNDI implementations (such as WebLogic or the Sun Java System Application Server) or LDAP. The pros and cons of using each storage mechanisms is covered in **Deciding which Type of Configuration Data Store to use** on page 144.

Prerequisite

Specifying how Configuration Data is Stored on page 139.

In this section refers to the system on which the configuration data is stored as the Configuration Data computer, and the system on which you are running the JMS Grid Admin Tool as the Remote computer.

On the Configuration Data Computer

1 Start up the JMS Grid Admin Tool on the Configuration Data system. You need to do this in order to complete steps 2 and 3.

2 Using the JMS Grid Admin Tool, find out which storage mechanism has been used to store the Admin Store configuration data on the Configuration Data system.

This is done by opening the Admin Settings dialog and viewing the value of the **Storage Plugin** widget.

Note: *If a file based mechanism is being used, then the remote system will need file access to the Configuration Data system before you can access the configuration data.*

You cannot remotely edit the configuration of JMS Grid if you are using XML Remote type configuration storage – unless your remote drive is mapped. With XML remote storage you can read data from a URL, but changes are output to a specified directory name. For that directory name to work a drive mapping in needed.

3 Make a note of the Admin Store configuration data store properties and values. Different properties are used by the different storage mechanisms. Details of the properties are given in **Specifying how Configuration Data is Stored** on page 139.

4 Close the Admin Tool on the Configuration Data system.

On the Remote Computer

1. Ensure that JMS Grid is installed locally on the Remote computer. The Remote system's `%JMS Grid%` environment variable should point to the JMS Grid installation on the Remote system.
2. Run the JMS Grid Admin Tool.
3. Open the Admin Settings dialog box by selecting `Preferences > Admin Settings` from the pull down menus.
4. Select the same Storage Plugin type as used by the Configuration Data computer. Set the store properties to match those that you noted in Step 2, above. You will have to make the following substitutions:

Table 8 Configuration Data

	Configuration Data System	Remote System
`Directory Path`	Drive letter – for example C:	Mapped network drive letter
`URL`	localhost	Actual machine name or IP address.

Once accepted, you should be able to see the same configuration data within the JMS Grid Admin Tool as you could see on the Configuration Data system. Depending on which Storage Plugin type is being used and on the access permissions you have, then you may or may not also be able to edit that data.

3.3.15 How to tell that a Daemon is using its Daemon Configuration

If the daemon configuration has not been found then JMS Grid will start up a daemon, with the name you've provided – but that daemon will load its most recent locally cached configuration.

When you start a JMS Grid daemon from a command prompt you are presented with the same command line output if the daemon has found its correct configuration or if it has not found it.

The command line output you will see will look something like this…

```
C:\JavaCAPS51\JMS_Grid>startserver /n myDaemon
JMS Grid Daemon: myDaemon
Initializing Daemon - logging to File: C:\JavaCAPS51\JMS_Grid\WMS-
localFileStorageDaemon.log
```

In order to find out which configuration a daemon is using, you must look in the daemon's log file.

Start the Daemon

If the daemon is running on the computer that stores the configuration data see **Starting a Daemon** on page 56.

If the daemon is running on a different computer than the one that stores the configuration data see **Starting a Daemon on a Computer that is Remote From its Configuration Data** on page 70.

Examine the Log File

1 Make a note of the name and the Resource Location of the daemon.

You can do this from the JMS Grid Admin Tool. Navigate to the daemon you are interested in (under either the Single Daemons node or the `Networks > Your Network > Your Cluster node`).

2 Open the daemon's log file.

Use the name of the daemon to work out which is its log file. If you have access to the Command Window from which the daemon was started, the daemon's log file location is output to that Window after the text 'logging to File:'. For more details see How To access a daemon's log file.

3 Examine the log file. See **Accessing a Daemon's Log File** on page 74.

If the daemon has found its configuration then in the log file it will tell you that it has bound to the Resource Location that was specified in its configuration:

The relevant part of log file would look something like the snippet shown below. The line showing which resource location the daemon has bound to is shown in bold type.

```
2002-08-08 12:43:25,083 INFO   - Initializing Message Store ....
2002-08-08 12:43:25,313 INFO   - Initializing Message Store complete.
2002-08-08 12:43:30,751 INFO   - Initializing JMX Connector ...
2002-08-08 12:43:31,041 INFO   - JMS Grid Daemon(myDaemon) now bound
to tcp://myHost:2352
2002-08-08 12:43:31,422 INFO   - Management Agent initialized
2002-08-08 12:43:31,422 INFO   - JMS Connector now ready to accept JMS
Clients at tcp://myHost:2352
```

If the daemon did not find its configuration then in the log file it will report an error, as shown in the log file snippet below:

```
2002-08-08 12:40:56,449 INFO   - Initializing Message Store ....
2002-08-08 12:40:56,559 INFO   - Initializing Message Store complete.
2002-08-08 12:40:56,559 WARN   - Failed to access admin store - trying
wave message store ...
2002-08-08 12:40:56,559 ERROR  - Failed to load configuration from
store(//admin/DefaultConfiguration/daemons/localFileStorageDaemon not
found)
2002-08-08 12:40:56,559 WARN   - Can't access configuration from
store
2002-08-08 12:41:01,867 INFO   - Initializing JMX Connector ...
2002-08-08 12:41:02,087 ERROR  - Failed to load configuration from
store(//admin/DefaultConfiguration/daemons/localFileStorageDaemon not
found)
2002-08-08 12:41:02,087 WARN   - Can't access configuration from
store
```

The log file will then report that the daemon has bound to the next available default resource location:

```
2002-08-08 12:41:02,087 INFO   - JMS Grid
Daemon(localFileStorageDaemon) now bound to tcp://mikebpc:2345
```

```
2002-08-08 12:41:02,478 INFO   - Management Agent initialized
2002-08-08 12:41:02,478 INFO   - JMS Connector now ready to accept JMS
Clients at tcp://mikebpc:2345
```

3.3.16 Accessing a Daemon's Log File

Each running daemon creates a log file in which the daemon records important events during its lifetime.

The important events that a daemon records include information:

- The times you started and stopped the daemon
- The version and build number of the daemon
- The daemon and cluster name
- The URL to which the daemon binds
- Details of clients that attempt to log into it

In this section explains how to find the log file for a given daemon.

A daemon's log files are stored in the daemon's working directory. By default the work directory will be the directory `</code>wdir</code>` under your JMS Grid installation. You can specify a different working directory using the /w (windows) or -w (unix) argument of the `<code>startserver</code>` command.

The location of a working directory depends on whether or not you specified it in the /w option (-w option on Unix) in the command line you used to start the daemon.

If you do not specify the /w option, then the default working directory is used. The default working directory is called `<courier>wdir</courier>` and can be found under the root of your JMS Grid installation (e.g. `C:\JavaCAPS51\JMS_Grid\wdir`)

Under the working directory you will find the daemon log files at:

```
log\WMS-<YourDaemonName>.log.<backupIndex>
```

where

YourDaemonName: is the name you gave to your daemon.

BackupIndex: the index number of a backup log file. The current log file does not have a backup index.

Note: *When you start a daemon from a command window, the command output tells you the location of that daemon's log file.*

For example:

```
C:\JavaCAPS51\JMS_Grid>startserver /n myDaemon
JMS Grid Daemon: myDaemon
Initializing Daemon - logging to File:
C:\JavaCAPS51\JMS_Grid\wdir\logs\WMS-localDaemon-50607.log
```

See also

- **Configuring a Daemon's Logging Properties** on page 77.
- **How to tell that a Daemon is using its Daemon Configuration** on page 72.

3.3.17 Configuring a Daemon from a Properties Text File

If your system needs to configure JMS Grid using scripts rather than using the JMS Grid Admin Tool, then you should consider using a text properties file to provide JMS Grid daemons with their configuration.

Prerequisite

- You should be familiar with **Configuring a Single Daemon** on page 54.

Create a text properties file. The table below gives details of the properties you are allowed to set, that property's description and its default value.

Table 9 Properties Text File

Property Name	Description	Default value
`autoDiscoveryAllowed`	Determines whether the daemon will register as a service for clients to automatically discover, and whether the daemon itself will use multicast discovery to locate other daemons	false
`bindAddresses`	urls the daemon will attach to	tcp://localhost:50607
`closeClientsOnNetworkConnectionFailure`	close all connections on a network connection failure	false
`clusterID`	unique id for the cluster the daemon belongs to	default
`connectionLoadBalancing`	Either RandomLoading or LeastUsedLoading	RandomLoading
`daemonClusteredConnections`	URLS of daemons in the cluster to connect with	
`daemonConnectionRetriesTimeout`	timeout in seconds before retrying to establish connection to another daemon	10000
`daemonNetworkConnections`	URLS of daemons in the hierarchy to connect with	
`exceptionOnNoQueueReceiver`	throw an exception on client of no receiver for a Queue	false
`jmxTimeout`	The maximum time a synchronous JMX client will wait for a response from this daemon. This value is defined on the daemon and passed to the client when it first connects.	30000

Table 9 Properties Text File (Continued)

Property Name	Description	Default value
marWindowSize	Maximum number of unacknowledged messages that can have been dispatched from a particular queue or durable subscription before the daemon will cease sending further messages. (Note: A MAR is an internal "Mark As Read" message).	100
maxDaemonConnectionRetries	number of retries attempting to connect to another daemon	10
maxInternalQueueSize	set the amount of VM memory allowed to be used by the daemon's message dispatch Queues (in bytes)	8388608
maxTopicDispatchQueues	maximum number of separate dispatch queues used for distributing publish/subscribe (Topic) messages	100
messageStoreType	type of message store (must always be set to SPIRITDB)	SPIRITDB
name	unique name for the message server	generated automatically
networkConnectionQueueFilters	filter used to restrict queue propagation across cluster boundaries	
networkConnectionTopicFilters	filter used to restrict topic propagation across cluster boundaries	
password	password for firewall proxy server	null
pingEnabled	use ping protocol to determine network outage	false
pingTimeout	time (ms) before next ping	10000
proxyHost	Hostname or IP address of firewall proxy server (only applicable if HTTP protocol is being used)	null
proxyPort	Port of firewall proxy server	0
serviceDiscoveryChannel	Multicast channel used for automatic discovery of daemons. Only used if the allowAutoDiscovery parameter is set	multicast:// 224.0.0.4:3495

Table 9 Properties Text File (Continued)

Property Name	Description	Default value
`spiritDbDispatchThreads`	Number of threads used to dispatch messages from the message store. There will be this number of threads dispatching from queues and the same number of threads dispatching from topics.	2
`storeName`	JNDI name (relative to parent context) under which the configuration of this object is stored. Not normally changed except through admin tool.	null
`timeSocketBlockedBeforeClosing`	max time in milliseconds that a client is allowed to be blocked receiving traffic before being closed	30000
`username`	User name for proxy	null

Note: *If you want to add comments to your properties configuration file, put a # character at the start of the line.*

When you start the daemon using the startserver command, specify the name of the properties file using the /p (windows) or -p (unix) argument. This is explained in more detail in **Starting a Daemon** on page 56.

3.3.18 Configuring a Daemon's Logging Properties

Each JMS Grid daemon creates log files that record the daemon's runtime activities.

The JMS Grid Admin Tool enables you to specify two log file properties:

- `maxLogFileSize`–The maximum size for each of that daemon's log files.
- `maxLogBackupIndex`–The maximum number of backup log files that will be created for a given daemon. A new backup log is created when the current log size grows greater than its `maxLogFileSize`. The maximum number of actual log files will be 1 more than this number (counting the 'live' log file). After `maxLogBackupIndex` files have been reached, the oldest log file is deleted.

Default values:

```
maxLogFileSize - 2000000 (2 MBytes)
maxLogBackupIndex -10
```

Prerequisites

- You should be familiar with **Editing a Daemon's Configuration** on page 61.
- You should be familiar with **Accessing a Daemon's Log File** on page 74.

To configure log properties

1 Follow the basic steps in **Editing a Daemon's Configuration** on page 61.

2 In the Message Daemon Properties dialog, bring the **General** tab to the top.

3 Fill in the `maxLogFileSize` and/or `maxLogBackupIndex` fields with the values you require.

Changing a Daemons working directory

You can change the daemon's working directory using the `/w` option (`-w` on Unix) in the `startserver` command when starting the daemon from the command line. See **Starting a Daemon** on page 56.

3.4 Networks of Clusters of Daemons

A robust, fault tolerant, production-ready JMS Grid Message Server will consist of many concurrently running daemons. These daemons may be running on many different machines. If there is a hardware or software failure for some of these daemons, then the Message Server will continue to function transparently.

Concurrently running daemons can be organized into clusters of daemons and networks of clusters of daemons.

3.4.1 Network and Cluster Concepts

What is a Network?

A network is a grouping of inter-connected clusters. It is the highest order of grouping for a JMS Grid Message Server. A network may be spread across multiple hardware platforms and Wide Area Networks.

What is a Cluster?

A cluster is a group of cluster daemons. Each daemon in a cluster automatically connects to every other daemon in that cluster. For this reason, clusters tend to span Local Area Networks rather than Wide Area Networks. Each daemon in a cluster will normally run on the same hardware platform.

What is a Cluster daemon?

A cluster daemon differs from a single daemon in that it is intended to run as part of a community of inter-connected, communicating processes. A ClusterID specifies the cluster to which the daemon belongs.

Fundamentally, single and cluster daemons are very similar. The only difference being that a cluster daemon has a ClusterID. A cluster daemon will share its messages with all other cluster daemons with the same ClusterID.

3.4.2 Creating a New Network

A network is a grouping of clusters. Before you can create clusters of daemons you must create a network into which you will add cluster(s).

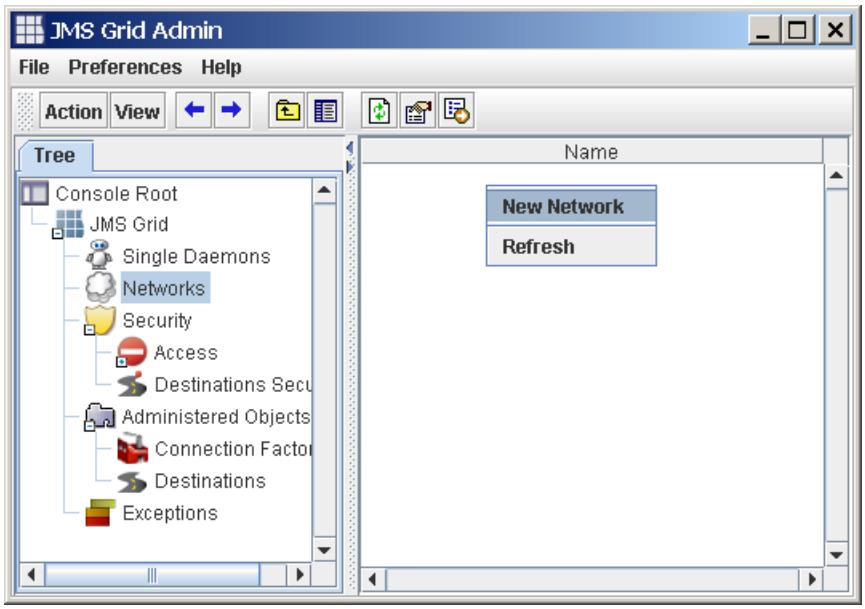
Prerequisites

- You should be familiar with **Navigating the Tree View** on page 49.
- You should be familiar with **Networks of Clusters of Daemons** on page 78.

To create a Network

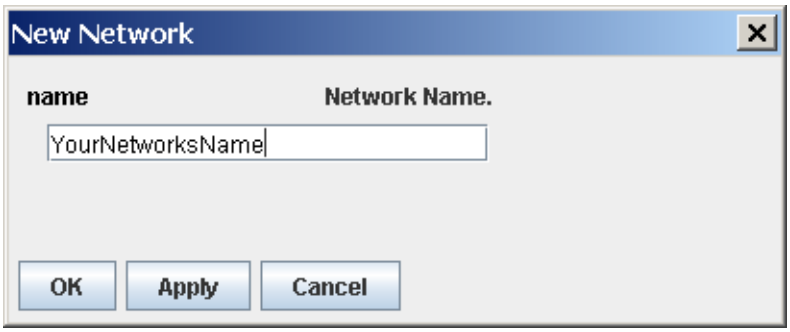
1. In the Tree View, navigate to `Console Root > JMS Grid > Networks`.
2. Bring up the Detail View's Panel menu by right mouse button clicking away from any existing Networks in the Detail View.
3. From the pop-up menu, select **New Network**.

Figure 39 Create a Network



4. This opens the New Network Dialog. Enter the name of the new network into the Name field.

Figure 40 Network Name



Note: *The name of a network must be different from the names of all the other networks stored in this directory.*

5 Click the **OK** button.

Deleting a Network

When a Network configuration is no longer needed, it can be removed from the Admin Tool configuration.

Prerequisite

- You should be able create a network. See **Networks of Clusters of Daemons** on page 78.

3.4.3 Deleting a Network

1 In the Tree View, navigate to: `Console Root > JMS Grid > Networks`

2 In the Detail View, select the Network you wish to delete.

3 Right mouse click with the mouse still over the Network and from the pop up Item menu that appears, select Delete.

Note: *If the Network contains any clusters then you will not be allowed to delete the Network. Clusters must be deleted before the Network can be deleted.*

4 Click **Yes** in the confirmation dialog that appears.

3.4.4 Creating a New Cluster

A cluster is a group of cluster daemons. Before you create any cluster daemons you must create a cluster that will group those daemons together. Before you create a cluster, you must create a network into which you will add your new cluster.

Prerequisite

- You should be familiar with **Navigating the Tree View** on page 49.
- You should be familiar with **Networks of Clusters of Daemons** on page 78.
- You should be familiar with **Creating a New Network** on page 78.

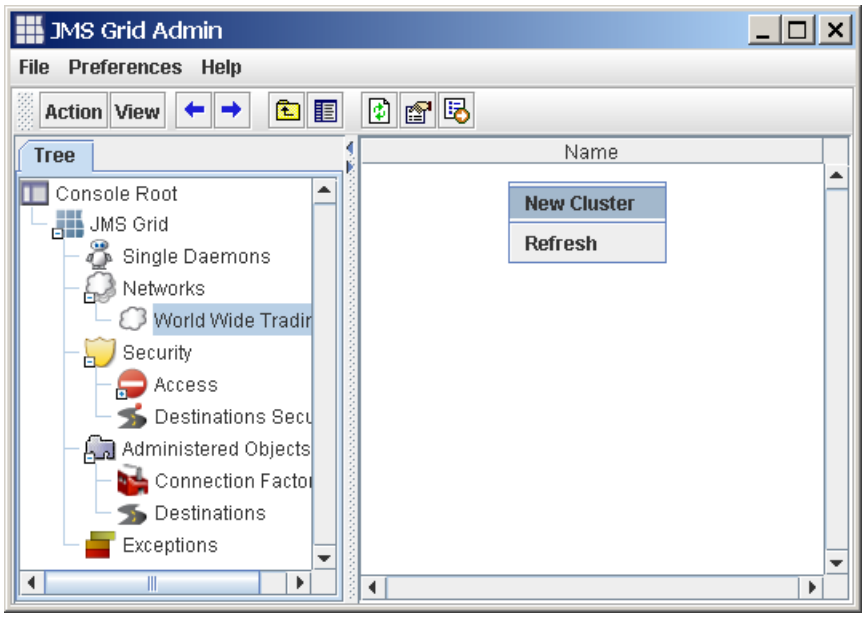
To create a new cluster

1 In the Tree View, navigate to the network node to which you want to add a cluster.

2 Bring up the context sensitive pop-up menu by either right mouse button clicking in any empty part of the Detail View or by right mouse button clicking on the Detail View's column heading.

3 From the pop-up menu, select New Cluster.

Figure 41 New Cluster



4 This opens the New Cluster Dialog. Enter the name of your new cluster into the Name field.

Figure 42 Cluster Name

Note: *The name of a cluster must be different from the names of all the other clusters within your configuration data store. You cannot use the same cluster name even if they are in different networks. If you try to duplicate a name, you will receive a warning when you apply the new cluster and will not be allowed to create that new cluster. You must only use alphanumeric characters for the cluster name. Do not use spaces.*

5 Click the **OK** button.

6 A **Warning** dialog appears asking 'Apply changes?' Click **OK**.

A new cluster node will now be displayed in the Detailed View.

3.4.5 Creating a Configuration for a Cluster Daemon

Cluster daemons are the type of daemons that are grouped together to form a community of inter-connected, communicating processes called a cluster. This section explains how to create a cluster daemon's configuration.

This does not actually create a running cluster daemon. It only creates a configuration that will be used when a cluster daemon is started. To find out how to start a cluster daemon see How To start a daemon.

Note: *Daemons in a cluster work closely together. As such, you should configure each daemon to be compatible with the other daemons in the cluster. The only property that must be consistent across each daemon in a given cluster is the* `MessageStoreType`*. It is recommended that cluster daemons that are in the same cluster be configured so that all properties are consistent, except those that must be unique.*

Prerequisite

- You should be familiar with **Creating a New Network** on page 78.

To create a new cluster

1. In the Tree View, navigate down to the cluster into which you want to add a new cluster daemon.
2. In the Detail View, right mouse button click away from any of the existing daemons in that cluster. If the Detail View is filled, right mouse button click the Detail View column heading. From the panel menu that is brought up, select **New Daemon**.
3. The **Message Daemon Properties** dialog will appear. This dialog looks almost identical to the dialog that is used to create a new single daemon. The only difference is that the ClusterID field is populated with the value of the cluster into which this new daemon is going.
4. Enter a unique name for the daemon into the name field. Only use alphanumeric characters for the daemon name. Do not use spaces!

Note: *The daemon name must be unique to the names of all other daemons your whole JMS Grid installation. You cannot give cluster daemons the same name as any single daemons or any other daemons in any other cluster that are stored in the same configuration data store. The system will automatically generate a default Network URL on which the daemon will listen. See How To set up a daemon network URL. You can use the default Network URL or create your own.*

5. Click the OK button.

When a new daemon has been created, it is displayed in a row in the table in the Detail View.

Cluster daemons and single daemons are fundamentally very similar. Many of the How Tos relating to cluster daemons are covered in the Managing Single Daemons section.

See also

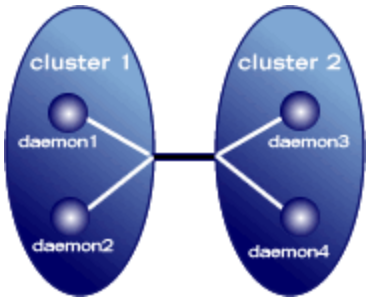
- **Specifying a Daemon Network URL** on page 63

3.4.6 Creating a Connection Between Clusters

JMS Grid enables clusters to be connected together so that messages can be routed between different clusters. In order to do this one or more daemons in one cluster must be connected to one or more daemons in another cluster.

Note: *A cluster must contain one or more daemons before it can be connected to another cluster.*

Figure 43 Daemon Clusters Communicating



When a connection starts and ends in more than one daemon, then the starting and ending daemons will be tried in turn until a successful pair is found to send the messages. For example, as shown above a connection has two daemons at either end of the connection. When attempting to send a message across this connection from Cluster 1 to Cluster 2, Cluster 1 might initially try to send a message from Daemon1 to Daemon3. If that failed, then might try sending from Daemon1 to Daemon4. If there was another failure, it may then switch to trying to send from Daemon2, and so on.

Note: *To create a more fail-safe connection between clusters, it is a good idea to create more than one connection between different daemons in each of the clusters. Then the connection is not reliant on single daemons in the clusters.* ***A cluster must contain one or more daemons before it can be connected to another cluster. Networks cannot be connected together. A Network is the highest logical grouping****.*

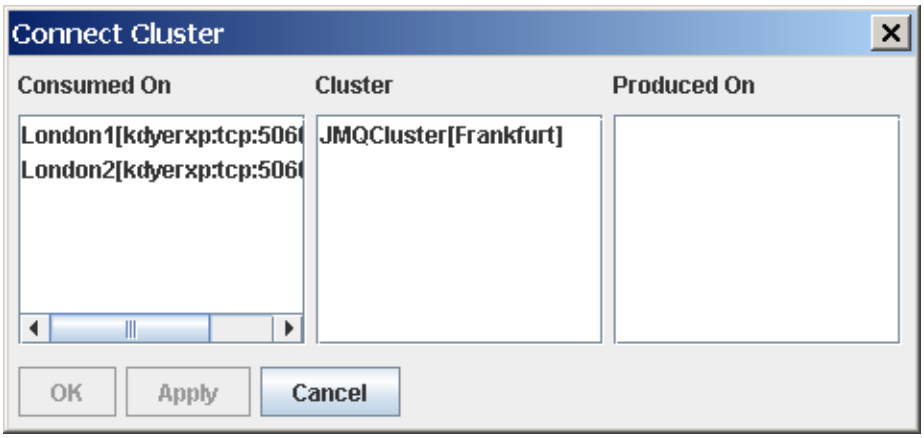
Prerequisite

- You should be familiar with **Creating a Configuration for a Cluster Daemon** on page 82.

To connect clusters

1. In the Tree View, navigate to the network that contains the cluster you want to connect.
2. In the Detail View, right mouse click over one of the clusters you want to connect. From the Item menu, select Connect. If that cluster contains any daemons the Connect Cluster dialog will be opened.

Figure 44 Connect Cluster



Note: *If that cluster does not contain any daemons a warning dialog will appear telling you that 'Cluster* `<Your Cluster>` *is not connectable, reason: No daemons in source cluster.'*

3 From the 'From Daemon' list in the Connect cluster dialog, select the daemon you want to connect to the daemon in the other cluster.

Note: *It does not matter which way round clusters are connected, i.e. cluster 1 daemon 1 → cluster 2 daemon 1 exhibits the same runtime behavior as cluster 2 daemon 1 → cluster 1 daemon 1*

4 From the 'To Cluster' list, select the cluster that you want to connect to.

5 From the 'To Daemon' list select the daemons in the 'To cluster' that you want as the other end of the connection.

6 If at least one item is selected from every visible pane, the OK button will enable. Click the OK button to confirm the connection.

Note: *If you are looking at the graphic view of the network, you will now see a line drawn between the two connected clusters. This line represents the connection.*

The Connect Cluster dialog can also be opened by:

1 Opening a Cluster's properties dialog and then clicking the Add button in the Network Connections section of that dialog.

2 From a network's graphic view, position the mouse cursor over one of the cluster ovals, right mouse click and select the Connect menu option from the pop-up.

3.4.7 Printing a Graphical View of a Network

This section explains how to obtain a printout of a graphical view of a network.

Prerequisites

- You should be familiar with **Toggling Between Detail and Graphical View** on page 52.
- You should be familiar with **Networks of Clusters of Daemons** on page 78.
- You should be familiar with **Deleting a Network** on page 80.
- You should be familiar with **Creating a New Cluster** on page 80.

To print a graphical view of a network

1. If you want to print off a graphical view of a network, in the Tree View, navigate to:
   `Console Root > JMS Grid > Networks <Network you want to print>`
2. Switch to the graphical view of that network or cluster. This is explained in How To switch between detail and graphical view.
3. Right mouse click with the mouse over the Network and from the pop up Item menu that appears, select **Print**.
4. The print dialog for your particular printer will appear. Fill this in and click the **Print** button.

3.4.8 Load Balancing Messages Across Cluster Daemons

This section explains how to configure a cluster daemon so that a client that connects to that cluster will use a certain type of load balancing to distribute messages across the daemons in that cluster.

Table 10 Load Balancing Options

Options	Description
`RandomLoading (Default)`	Do not redirect newly-connected clients to other daemons. The method for choosing which daemon a client connects to is left to the client.
`LeastUsedLoading`	Whenever a new client connects to this daemon, and another daemon exists in this cluster with fewer client connections, then the client connection will be relocated to that daemon. This connection strategy ensures that clients are evenly-loaded across the cluster.

Note: *This option is available for single daemons even though load balancing is only effective over a cluster of daemons. Setting this option on a single daemon will have no effect.*

Prerequisite

- You should be familiar with **Editing a Daemon's Configuration** on page 61.

To Load Balance Messages across cluster daemons

1. Follow the basic steps in **Editing a Daemon's Configuration** on page 61.

2 In the Message Daemon Properties dialog, bring the Clients tab to the top.

3 From the `connectionLoadBalancing` pull down at the top of the dialog box and select the types of load balancing you require.

3.4.9 Enabling Auto Discovery

Auto discovery enables clients to use multicast to automatically detect running daemons. The default load balancing mechanism is for the client to connect to the least loaded daemon in that cluster. If a load balancing mechanism is specified, then that mechanism will be used instead of 'least used'.

If no daemons are discovered (for example, if they are on a different network segment that the multicast cannot see), then the client's `messageChannels` property will be used instead.

The JMS Grid Connection Factory that is used by the messaging client must also be configured so that its autoDiscoveryAllowed flag is set, see **Editing Connection Factory Properties for a Normal JMS Grid Client Connection** on page 131.

Default value:

false

Prerequisites

- You hold be familiar with how to **Creating a Configuration for a Cluster Daemon** on page 82.

To enable or disable Auto Discovery

1 Follow the basic steps in **Editing a Daemon's Configuration** on page 61.

2 In the Message Daemon Properties dialog, bring the **Clients** tab to the top.

3 To enable auto discovery, ensure the AutoDiscoveryAllowed checkbox is ticked. To disable, ensure it's not ticked.

See also

- **Editing Connection Factory Properties for a Normal JMS Grid Client Connection** on page 131
- **Load Balancing Messages Across Cluster Daemons** on page 85

3.4.10 Specifying the Autodiscovery Multicast Channel

JMS Grid's auto discovery mechanism uses multicast to automatically detect running daemons. There is, however, a small chance that the default multicast channel used by JMS Grid clashes with the address used by another system running on the network. The JMS Grid Admin Tool allows you to overcome this by changing the multicast channel.

The default setting for the multicast channel is: `multicast://224.0.0.4:3495`

Prerequisites

- You should understand the principles of multicast.

- You should be familiar with **Enabling Auto Discovery** on page 86.

To use multicast to autodiscover running daemons

1. Follow the basic steps in **Editing a Daemon's Configuration** on page 61.
2. In the Message Daemon Properties dialog, bring the Inter-Daemon tab to the top.
3. Enter the new multicast message channel into the field labeled **serviceDiscoveryChannel.**

3.4.11 Viewing Network and Cluster Daemon Connections

One of the key features of JMS Grid is the ability to connect daemons together. Two types of inter-daemon connections exist in JMS Grid:

Cluster connections – Connections between daemons that belong to the same cluster. Within a cluster, every daemon is implicitly connected to every other daemon creating an interconnected lattice.

Network connections – Explicit connections from daemons in one cluster to daemons in another cluster.

A combination of these connections enables complex topologies of daemons to be deployed.

The JMS Grid Admin Tool provides a number of ways to enable you to view the daemons and their connections.

Prerequisite

You should be familiar with **Editing a Daemon's Configuration** on page 61.

To display connection between daemons

The JMS Grid Admin Tool provides a number of ways to let you view the topology of your networks of clusters of daemons.

From the Daemon Properties dialog

1. Follow the basic steps in **Editing a Daemon's Configuration** on page 61.
2. In the Message Daemon Properties dialog, bring the Inter-Daemon tab to the top.
3. Two lists are shown at the top of the dialog. The top list, Cluster Connections, gives a list of all the other daemons in the cluster to which this daemon is connected. The lower list, Network Connections, shows the connections between this daemon and daemons in other clusters.

Note: *For single daemons the Network Connections list and Cluster Connections list will always be empty.*

A Graphical View of a Network

The graphic view of a Network shows how clusters are connected together by displaying a line between two clusters that are connected – as shown below.

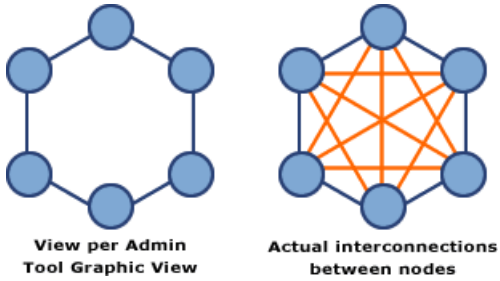
To see the graphic view of a Network, do the following:

Using the JMS Grid Admin Tool's Tree View, navigate down to the network you want to view. Then switch the view type. This is described in more detail in**Toggling Between Detail and Graphical View** on page 52.

A Graphical View of a Cluster

Using the JMS Grid Admin Tools Tree View, navigate down to the cluster you want to view. Then switch the view type. This is described in more detail in How to switch between detail and graphical view.

Figure 45 Graphical View of Cluster



Note: *The graphical view of a cluster does not show every connection between every daemon in that cluster. This is because the view would soon become too cluttered. Instead, only a single connection is shown between daemons. So if, in a graphic view, you can trace a route between two daemons, then in reality there will be a direct connection between those two daemons. The figure above shows the graphical view you will see of a six-daemon cluster and the actual connections that will exist between nodes.*

3.4.12 Configuring Daemon Reconnections

If a daemon fails to establish an initial connection to a peer or the connection terminates for any reason, the daemon will attempt to reconnect. In the case of network connections, the daemon will iterate through the list of URLs associated with the network connection until it is successful. After each unsuccessful attempt, the daemon will wait `daemonConnectionRetriesTimeout` milliseconds before attempting to reconnect to the next network URL in the sequence. This sequence will only be attempted `maxDaemonConnectionRetries` times.

Default values:

`maxDaemonConnectionRetries`: 2147483647 (equivalent to `Integer.MAX_VALUE`)

`daemonConnectionRetriesTimeout`: 5000 (milliseconds)

Prerequisite

- You should be familiar with **Editing a Daemon's Configuration** on page 61.

To configure daemon reconnections

1 Follow the basic steps in **Editing a Daemon's Configuration** on page 61.

2 In the Message Daemon Properties dialog, bring the Inter-Daemon tab to the top.

3 Into the field marked `maxDaemonConnectionRetries`, enter the number of times you want the daemon to attempt to reconnect after it has lost a connection to a remote daemon.

4 Into the field marked `daemonConnectionRetriesTimeout`, enter the time in milliseconds before a daemon will attempt to reconnect to another daemon in its cluster or network after a connection attempt has failed.

3.4.13 Configuring Message Filters on Inter-daemon Network Connections

JMS Grid uses dynamic subscription routing throughout clusters and networks of clusters. When a client subscribes to receive messages from either a Topic or a Queue, this subscription information is propagated to every daemon in the message server network. For large deployments, this can result in unnecessarily large amounts of routing information being shared.

It is possible to prevent subscription routing information being passed between clusters by using network connection filters. These filters only allow the forwarding of subscription routing information that match the filter.

The JMS Grid Admin Tool enables you to set up separate filters on either destination types (queues or topics).

If no filters are specified, then no filtering is applied, that is, messages are forwarded to all destinations. However, if a filter is created to a single destination of one type, then only messages to that single destination of that destination type are allowed to pass. Unless empty, a filter specifies the names of destinations that are allowed to pass, not the names of destinations that are blocked.

Prerequisite

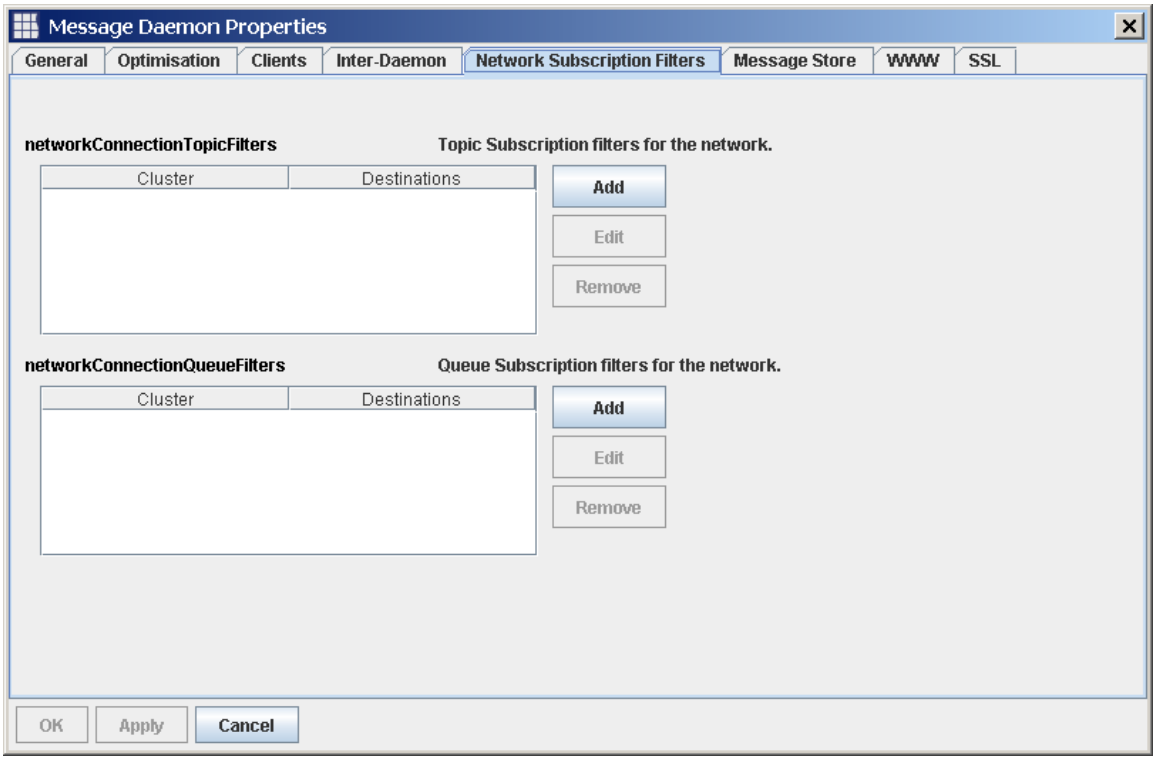
Editing a Daemon's Configuration on page 61.

To set message filters

1 Follow the basic steps in **Editing a Daemon's Configuration** on page 61.

2 In the Message Daemon Properties dialog, bring the Network Subscription Filter tab to the top – this dialog is shown below.

Figure 46 Message Daemon Properties



3 The dialog box shows two lists, one for filters on Topic destinations, the other for filters on queue destinations.

4 To select all the topics you want to allow messages to, click the Add button to the right of the Topic filter list. To select all the queues you want to allow messages to, click the Add button to the right of the Queue Filter list. In both cases, a Subscription Filter Properties dialog box is opened.

5 In the Subscription Filter Properties dialog, select the network within which you want to create a subscription filter.

6 In the destinations part of the dialog, enter the names of the destination you want to be passed across to other clusters on that network.

3.5 JMS Grid Security

JMS Grid's security system enables administrators to define who has the right to do what within the messaging system and to protect confidential information.

3.5.1 Introduction to Security

In JMS Grid, security is applied when:

- Messaging clients attempt to communicate with the message server.

- Users try to configure the message server.
- Users try to manage the runtime operation of the server.

Messaging Clients Attempt to Communicate with the Message Server

The JMS Grid Admin Tool provides administrators with fine-grained control over which JMS messaging clients are allowed to connect to the server. Once a client is connected, the administrator can then decide which destinations that client can send messages to and which destinations that client is allowed to read messages from. The administrator can also specify destinations to which encrypted messages are sent

Users try to Configure the Message Server

JMS Grid defines a special Administrator permission. Only Users with this permission are allowed to log into the JMS Grid Admin Tool and are able to configure the message server.

Users try to Manage the Runtime Operation of the Server

An administrator must supply a username and password before being allowed to manage the runtime operation of the message server. Runtime management of JMS Grid is described in the JMS Grid JMX Management Guide.

3.5.2 Security Concepts

JMS Grid contains an extensive security package which makes it possible to provide all of the common security functions. These include:

- **Authentication** the ability to establish an identity
- **Authorization** the ability to control access to resources
- **Encryption** the ability to protect sensitive information

Authentication

Authentication is the process of determining someone's, or something's, identity. The most common means of establishing an identity, and the method used in JMS Grid, is to associate a password with a username.

The JMS Grid Admin Tool is used to create Users and set their initial passwords. When security is enabled, JMS clients must authenticate themselves to the JMS Grid Message Server by supplying the correct username and password combination.

Authorization

Authorization is the process of determining whether a known identity has the privilege to perform a task or access a resource.

Authorization is performed in JMS Grid by defining Permissions, which can then be allocated to different Users. Collections of Permissions, known as Groups, can also be created. Groups make it easier to allocate a large collection of Permissions to a User.

Encryption

JMS messages can sometimes contain confidential information. JMS Grid provides the facility to define Secure Destinations. When Secure Destinations are created any messages sent to those Destinations are encrypted using a Destination specific encryption key.

3.5.3 What are Permissions?

Permissions are the fundamental way to provide authorization in the JMS Grid.

Each Permission has the following attributes:

Table 11 Permission Attributes

Attribute	Description
`Name/ Permission Name`	This is a unique name, chosen by the administrator, and is intended to be meaningful to human operators. When Permissions are allocated to Users and Groups, administrators use the Permission Name to identify specific Permissions.
`Resource Name/ Destination Name`	This is the queue or topic that the Permission is intended to grant access to. This can be a simple string, or a string terminated with an asterisk: '*'. For example, if you have a range of queues, all beginning with the word "trades", then you could define a Permission granting access to all of those queues with a single Resource name of trades*.
`Resource Type/ Destination Type`	This is the type of Resource being protected by a Permission. Available types are: Queue, Topic, Temporary Queue, Temporary Topic and Special Permissions.
`Access Mode`	Controls whether Users granted this permission will be given read, or read/write access to the Resource(s) specified.

It is important to realize that creating a Permission does not in itself grant any access privileges to any Users or Groups. Permissions are merely ways of defining access control privileges in advance of allocating those privileges to Users or Groups.

Table 12 Default Permissions

Name	Resource Name	Resource Type	Access Mode
`WriteAllQueues`	*	Queue	read/write
`WriteAllTopics`	*	Topic	read/write

Table 12 Default Permissions

Name	Resource Name	Resource Type	Access Mode
`WriteAllTmpQueues`	*	Temporary Queue	read/write
`WriteAllTmpTopics`	*	Temporary Queue	read/write
`Administrator`	n/a	Special	n/a
`AnonymousLogin`	n/a	Special	n/a
`ManagementAPIQueues`	`com.spirit.management.*`	Queue	read/write
`ManagementAPITopics`	`com.spirit.management.*`	Topic	read/write

The Management API Permissions are used by JMS Grid's JMX-based runtime Management system. This is described in **JMX Management** on page 221.

Special Permissions

In addition to the normal Permissions associated with granting access to JMS destinations, JMS Grid defines various Special Permissions. Special Permissions cannot be created, modified or removed by administrators. However you can choose whether or not to assign Special Permissions to Users.

Special Permissions are associated with privileges, which are recognized by the JMS Grid code. The Special Permissions are as follows:

Anonymous Login Permission

The Anonymous Login Permission is a Permission that can be granted to the Default User account. When this Permission has been granted any JMS Grid Message Server session that has not logged in, or has logged in under an unknown username, will inherit the privileges granted to the Default User account.

This Permission only has any meaning when it is added to the Default User account.

Administrators may add this Permission to other User accounts, but this will have no effect.

If the Default User account does not have this Permission, and security is enabled, then anonymous logins and logins from unknown Users will both be prevented.

The table below can summarize this:

Table 13 Anonymous Login Permission

Security State	Anonymous Login Permission	
	Granted	**Not Granted**
Security enabled	Anyone can log-on	unknown Users prevented from log-on

Table 13 Anonymous Login Permission (Continued)

Security State	Anonymous Login Permission	
Security disabled	Anyone can log-on	Anyone can log-on

Administrator Permission

This Permission can be added to any User or Group account. Users that inherit this Permission are allowed to login to the Admin Tool.

This Permission is granted to the Admin User account. You cannot remove this Permission from the Admin User account.

3.5.4 What are Groups?

Groups are collections of Permissions and other Groups. They are used to enable Administrators to define complex lists of Permissions that can then be easily added to Users or Groups. Groups help to simplify the administration of security and access items.

Table 14 Group Attributes

Attribute	Description
Name	The group's name should be meaningful to a human operator.
Permissions	These are the named Permissions that the Group possesses. A Group can have zero or more Permissions.
Groups	Groups can themselves be members of other Groups. When a group is a member of another group, it inherits all the permissions defined in its parent group.

Table 15 Default Groups

Name	Parent	Permissions
Super	none	WriteAllQueues WriteAllTopics WriteAllTmpQueues WriteAllTmpTopics
Default	Super	none
Management	none	ManagementAPIQueues ManagementAPITopics

Note: *The Groups Detail View only shows Permissions that are mapped directly to each Group. Permissions that are inherited from parent groups are not shown. To see all*

the Permissions belonging to a Group, highlight the Group, right-click, and select the properties menu.

3.5.5 What are Users?

Users are the security objects that ultimately determine whether all authentication and authorization is successful or not.

Each User has the following attributes.

Table 16 User Attributes

Attribute	Description
Username	This is the name by which the User is known.
Password	Clients logging into JMS Grid must provide the correct username and password combination in order to authenticate themselves to JMS Grid. This password does not get stored in the JMS Grid Admin Tool store, nor does it ever get passed from one machine to another. Instead, it is used to derive a public/private key pair. Only the public key gets stored. The Private key is derived every time from the password in order to sign a session specific token every time the User logs in.
Permissions	Permissions can be added directly to a User. This way, individual Users can be given customized access to particular queues and topics.
Groups	Users can have zero or more Groups. Users inherit all of the Permissions allocated to all of the Groups to which they belong. Since Groups also inherit all the Permissions from Groups to which they belong, it is therefore possible to create highly sophisticated combinations of Users and Permissions.

The pre-installed Users are shown in the table below.

Table 17 Pre-installed Users

Name	Group	Permissions	Account Enabled
admin	Super	Administrator	true
default	Default	AnonymousLogin	true
demo	Super	none	true

3.5.6 What are Secure Destinations?

A message from a client application is automatically encrypted if it is sent to a destination that is secure. A JMS destination is made secure by creating a Destination Security admin object with a name that matches the JMS destination.

A Secure Destination does not necessarily map to a physically existing Destination. Queues and Topics can be created at run time. JMS Grid therefore allows administrators to define the properties of Secure Destinations in advance of the creation of the corresponding queue or topic.

Each secure destination has the following attributes.

Table 18 Secure Destinations

Attribute	Description
destinationType	Whether the Destination is a Queue or a Topic.
destinationName	The JMS Name of the Destination to make secure.
EncryptedDestination	A Secure Destination can have its encryption temporarily switched off. This may be required for debugging purposes, for example.
EncryptionKey	A sample of the encryption key. This is intended to give some visual feedback when the "Generate" button is pressed in the Secure Destination Properties dialog box. The Generate button can be used to generate a new encryption key for the Secure Destination.

3.5.7 Typical Usage of Permissions, Groups and Users

The simplest way to use Groups is to create a “flat” set of Groups. This is done in the following stages.

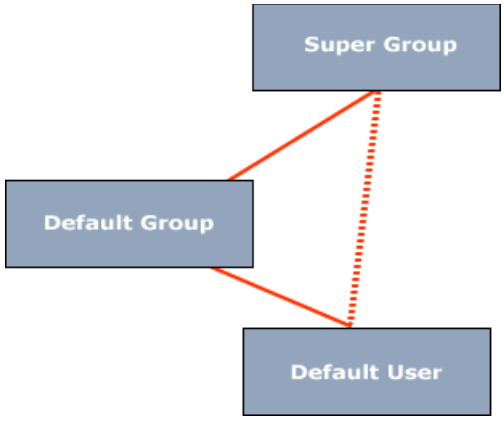
1. Create all of the desired Permissions.
2. Create a small set of Groups.
3. Add each Permission to one or more Groups. Each Group contains only Permissions.
4. Groups do not contain other Groups.
5. Allocate one and only one Group to each User.

For small, coarse-grain permission scenarios, a flat Group space is often sufficient and presents a simple, easily managed set-up.

However, in addition to being able to include Permissions in a Group, Groups can be made parents of other Groups. This allows Groups to be arranged into hierarchies, where each Group inherits Permissions from its parent Groups as well as from its directly allocated Permissions.

Note: *The JMS Grid Admin Tool does not check whether or not a User has inherited a Group multiple times from other groups. For example, in the default setup, the Default User is part of the Default Group. The Default User inherits the Super Group because the Default Group is a child of that Group. However the Admin Tool will allow you to directly associate the Super Group with the Default User, even though it already effectively has all of the privileges of that Group. This scenario is shown below.*

Figure 47 Groups



Note: *It is also possible to add the Super Group to the Default Group, creating a cycle in the above permissions setup. This could lead to runtime problems and should be avoided wherever possible.*

Users can also be associated with more than one Group. One possible way to use this feature is to create a Group that gives read or write permission to a wild-carded section of a single destination name. Individual Users can then be given access to one or more destination sections simply by giving them the appropriate Group membership.

For example If you had the following Destinations:

- myCompany.finance.public
- myCompany.finance.private
- myCompany.marketing.public
- myCompany.marketing.private

Create the following Groups which give access to the following destinations.

Table 19 Group Name and Destination

Group name	Permission	Destination
writeMyCompany	write	myCompany*
readMyCompany	read	myCompany*
writeFinance	write	myCompany.finance*

Table 19 Group Name and Destination (Continued)

Group name	Permission	Destination
readFinance	read	myCompany.finance*
writeMarketing	write	myCompany.marketing*
and so on…		

So, for a User with a high level of authority who needed to read and write to all company information, he or she would be added to the 'writeMyCompany' and 'readMyCompany' Groups. For some-one who needed to see all finance and marketing information, they would be added to 'readFinance' and 'readMarketing'.

3.5.8 What is the Default Security Configuration?

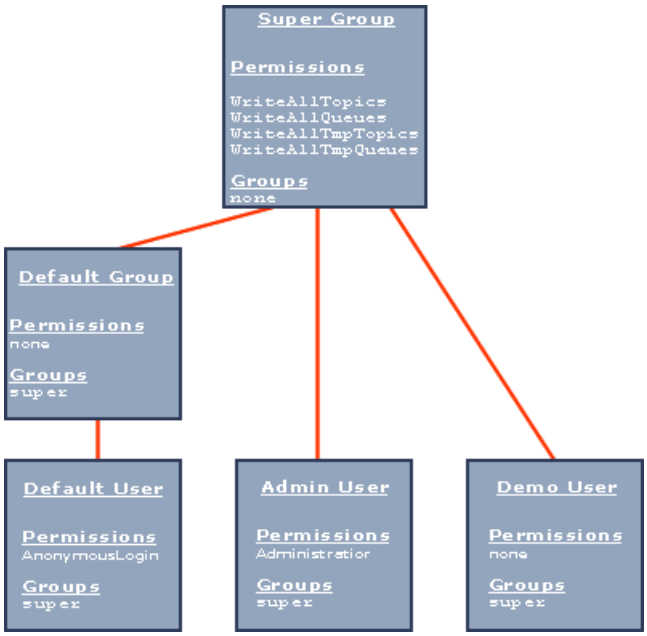
JMS Grid comes configured with a set of default Permissions, Groups and Users. The net effect of these default security objects is to create a security plan which permits all Users, even those logged in under an unknown name or logged in anonymously, to access all topics and queues.

The Permissions hierarchy is illustrated below. At the top is a Super Group that holds Permissions that collectively allow access to all queues and topics.

Two Users, admin (password admin) and demo (password demo), are members of the Super Group. They are therefore able to access all queues and topics. The admin User has the Administrator (special) Permission. The admin User is therefore permitted to login to the JMS Grid Admin Tool.

The Default Group is also a member of the Super Group. It has one member, the default User (password default). The default User inherits full queue and topic read /write access from the Super Group through the Default group. The default User is also assigned the Anonymous Login (special) Permission. This permits both anonymous logins and logins from unknown Users.

Figure 48 Super Group



Administrators can change the default security objects in any way that is required. For example, the Default Group can have its membership of the Super Group removed and replaced with separate, more restrictive, privileges.

3.5.9 Enabling JMS Grid Security

By default, security is disabled in JMS Grid. This means that there is no specific authentication and authorization of a client. All clients behave as if they are the default User – i.e. they are authenticated as being the default User, that they have the permissions of the default User and Secure Destination encryption is not available to them.

If security is enabled on the JMS Grid Message Server, it performs authentication, authorization of a client and allows encryption of Secure Destination messages.

Table 20 Summary of JMS Grid Security Behavior

Enabled	Disabled
Authentication Client logs in as a specific User. If no User is specified, client is logged on as the default User	Authentication No logging on for client
Authorization Client has permissions of the specific User they logged on as	Authorization Client has full permissions

Table 20 Summary of JMS Grid Security Behavior (Continued)

Enabled	Disabled
Encryption Message encryption to secure destinations is allowed	Encryption Message encryption to secure destinations is ***not*** allowed

Prerequisite

- You should be familiar with JMS Grid security.

To enable security on a message server

The basic steps of how to enable security on a message server and on individual clients is explained in the following sections:

- **Enabling JMS Grid Security** on page 99
- **Enabling JMS Grid Security (Command Line)** on page 100
- **Enabling JMS Grid Security (JMS Grid Admin Tool)** on page 101

Enabling JMS Grid Security (Command Line)

By default, no security checking is performed by JMS Grid. In order to enable encryption and to force authentication and authorization checks, a command line option to enable security can be set.

Note: *Security of the Message Server can also be enabled or disabled from within the JMS Grid Admin Tool. Any Message Server Security option that is specified on the command line will override the setting that is specified in the Admin Tool.*

Prerequisite

- You should be familiar with **Enabling JMS Grid Security** on page 99.

A JMS Grid Message server daemon is started up using the startserver script. The use of this script is also explained in **Starting a Daemon** on page 56. The command line syntax of this script varies depending operating system being used.

Windows

Use the startserver batch script using the following syntax:

```
startserver [/n serverName] [/w workingDirectory] [/s userName
password]
```

Unix

On Unix, the syntax for the startserver script is:

```
startserver [-n serverName] [-w workingDirectory] [-s userName
password]
```

Server security is enabled by including the –s or /s option on the command line. A username and password must also be provided when specifying the -s or /s option. See the note below.

To disable the security of the daemon you must do two things:

1. Omit the –s or /s option from the command line.
2. Ensure that security is disabled from within the JMS Grid Admin Tool, as described in How To enable or disable JMS Grid Message Server security from the JMS Grid Admin Tool.

Username and Password

The username and password that must be provided with the –s or /s option are currently only used by JMS Grid's Runtime Management System and are not used by your messaging clients applications.

Enabling JMS Grid Security (JMS Grid Admin Tool)

By default, no security checking is performed by the JMS Grid Message Server. In order to enable encryption and to force authentication and authorization checks, a flag can be set in the Admin Tool.

Note: *Any change in this property will take effect when the JMS Grid Message Server is restarted.*

Security of the Message Server can also be enabled or disabled from the command line when starting the Message Server.

Prerequisites

- You should be familiar with how to use the JMS Grid Admin Tool GUI.
- You should be able to enable JMS Grid security.

To enable or disable message server security

1. Open the Settings dialog. On the Admin Tool Menu bar, click **Preferences** followed by **Admin Settings**.

Note: *When you open the **Admin Settings** dialog, a warning may appear saying: "Admin and JMS Admin Object Store are selected to use the same plugin. Plugin settings will be shared!" Click the **OK** button.*

2. On the **Settings** dialog, choose the **Security Settings** tab.
3. To enable security, check the **EnableSecurity** check box at the top of the dialog box. To disable security, uncheck this checkbox.
4. Click the OK button.
5. A Warning dialog will appear asking if you want to apply the changes. Click the Yes button.

See also

- **Enabling JMS Grid Security (Command Line)** on page 100

Setting System-wide Security Parameters

Certain security settings can be applied on a system wide basis.

The Security Settings tab of the Admin Settings dialog displays a number general security settings. The Admin Settings dialog is opened by selecting `'Preferences > Admin Settings'` from the Admin Tool's pull down menu. The meaning of each of the dialog's settings is described below.

Note: *In the latest release of JMS Grid, only the 'enable Security' property is editable. Other properties may become editable in future releases.*

Table 21 Enable Security is Editable

Setting	Description
Enable Security	By default, no security checking is performed by JMS Grid. In order to enable encryption and to force authentication and authorization checks, this flag must be set.
Encryption Algorithm	This is the name of the encryption algorithm used both for secure destination encryption and for key distribution. It is currently fixed as Triple DES used in Cipher Block Chaining mode. Future versions of JMS Grid will allow a choice of encryption algorithms.
Encryption Type	This is currently fixed to allow only symmetric algorithms. Future version of JMS Grid will allow both symmetric and public key cipher algorithms. The choice will affect the list of available algorithms displayed in the Encryption Algorithm field.
Encryption Provider	This is the name of the Cryptographic Service Provider (CSP) providing cryptographic services on behalf of JMS Grid. This is currently fixed as Cryptix. Future versions of JMS Grid may allow a choice of several supported CSPs.
Encryption Key Size	The size of the encryption keys used. Some encryption algorithms allow variable size keys with larger keys providing stronger encryption. As the Encryption Algorithm is currently Triple DES, which has a fixed size key, it is currently not possible to change the key size of 192 bits. Future versions of JMS Grid may support encryption algorithms that permit variable size encryption keys.
Key Caching	JMS Grid can potentially support two key distribution mechanisms. In the first, the JMS Grid Message Server always holds encryption keys centrally. Encrypted messages are then transmitted and received using session keys. The second method encrypts the destination keys using the session key and caches them within the Client side of the JMS Grid Framework. This is considerably faster than the central key storage method. Currently, only this second method of key caching within the Client side is supported. Future versions of JMS Grid will permit disabling of key caching.

3.5.10 Creating a New Permission

Permissions are the fundamental way to provide authorization in JMS Grid. When a Permission is added to a User, then that User is granted the privileges that are defined in that Permission.

JMS Grid provides a number of default Permissions that are useful for broad-based privileges, such as, allowing a User to read and write to all Queue destinations.

However, your security requirements may need you to set up narrower privileges for Users, such as, only allow messages to be read from Topics whose name begins with 'trade'.

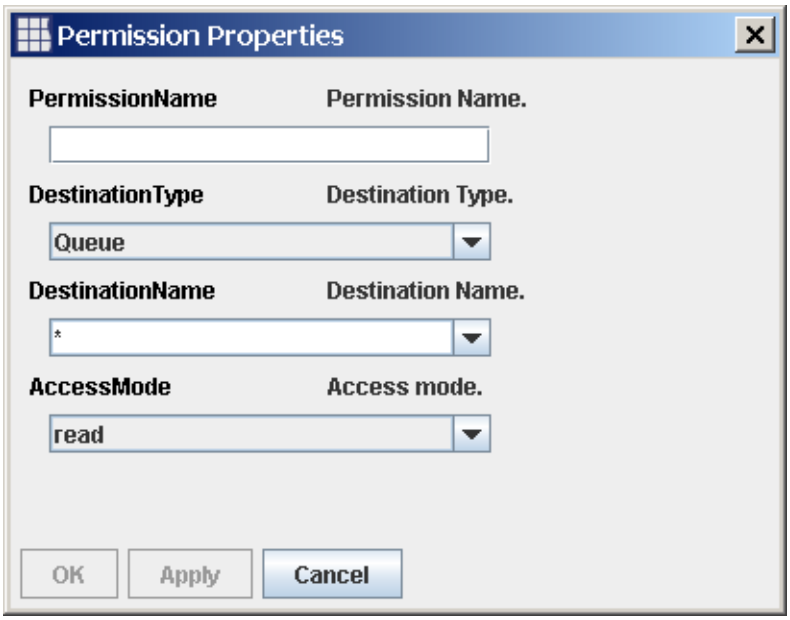
Prerequisites

- You should be familiar with **Navigating the Tree View** on page 49.
- You should understand **Security Concepts** on page 91.

To create a new permission

1. In the Tree View, navigate to the Permissions node at: `Console Root > JMS Grid > Security > Access > Permissions`
2. In the Detail View, position the cursor away from the existing Permissions. Click the right mouse to bring up the Panel Menu and select **New**. The **Permission Properties** dialog will open, see below. If the Detail View is full, you can bring up the Panel Menu by right mouse clicking on the Detail View's column headings.

Figure 49 Permission Properties



Note: *If the elements displayed in the Detail View fill the screen such that it is not possible to position the cursor away from them, then right click the table heading in order to bring up the Panel Menu.*

3 Enter the fields that define your new Permission. A full description of the meanings of the fields is given in What are Permissions?

Note: *By default the `destinationName` combo box has a * value - this means this permission will apply to all destinations of the selected connection type.*

The `destinationName` field is an editable Combo box that allows you to:

- Select an existing, individual Destination from the pull down menu.
- Type in the name of a wildcard Destination.
- Type in the name of a temporary Destination that does not currently exist, but which will exist at runtime.

Users may select different connection types from the `destinationType` combo box; this selection will affect the `destinationName` combo box. For example if a Queue item is selected as the destination type then `destinationName` will only contain the names of Queues.

4 Click the **OK** button then confirm the changes at the 'Apply Changes?' warning dialog.

5 The new Permission will appear in the table of permissions in the Detail View.

3.5.11 Editing a Permission

This section explains how to edit the properties of a Permission.

Prerequisite

- You should be familiar with **Creating a New Permission** on page 103.

To edit a permission

1 In the Tree View, navigate to the Permissions admin object at: `Console Root > JMS Grid > Security > Access > Permissions`

2 The Detail View will now show a table of all existing Permissions. Select the Permission to edit.

Note: *You are not allowed to edit the default Permissions. If you try, a warning dialog will open telling you than the item's properties cannot be altered.*

3 From the Item Menu that appears, select **Properties**. The **Permission** Properties dialog will appear.

4 Modify the required properties. The meanings of the property fields are explained in What are Permissions?

5 When the edits are complete, click the **OK** or **Apply** buttons. A prompt for saving the changes appears. Click the **OK** button to apply the changes to the Permission.

Another Way to open the Properties Dialog

You can open an item's properties dialog by selecting it in the Tree View and then click on the Properties button in the Toolbar.

3.5.12 Deleting a Permission

When messaging clients are no longer using a redundant permission, it is possible to remove it from the JMS Admin Object Store.

Prerequisite

- You should be familiar with **Creating a New Permission** on page 103.

To delete a permission

1. In the Tree View, navigate to: `Console Root > JMS Grid > Security > Access > Permissions`
2. In the Detail View, select the Permission instance to delete.

Note: *If you want to delete multiple Permissions at the same time, you can select multiple Permissions by holding down the Control key as you select each of the Permissions you want to delete.*

You are not allowed to delete the default permissions.

3. Right mouse button click, and from the pop up Item menu that appears, select **Delete**.
4. Click **Yes** in the confirmation dialog that appears. The Permission will now be deleted from the chosen Admin Store.

Note: *If a Group contains a Permission that you delete, then the Permission is not removed from that Group.*

3.5.13 Creating a Group

Groups are collections of Permissions that can also contain other Groups. They are used to make it possible for administrators to define complex lists of Permissions that can then be associated with Users or added to other Groups in a single administrative task.

Only experienced administrators should create Groups. Cyclic Permission inheritance may lock the entire application.

Prerequisite

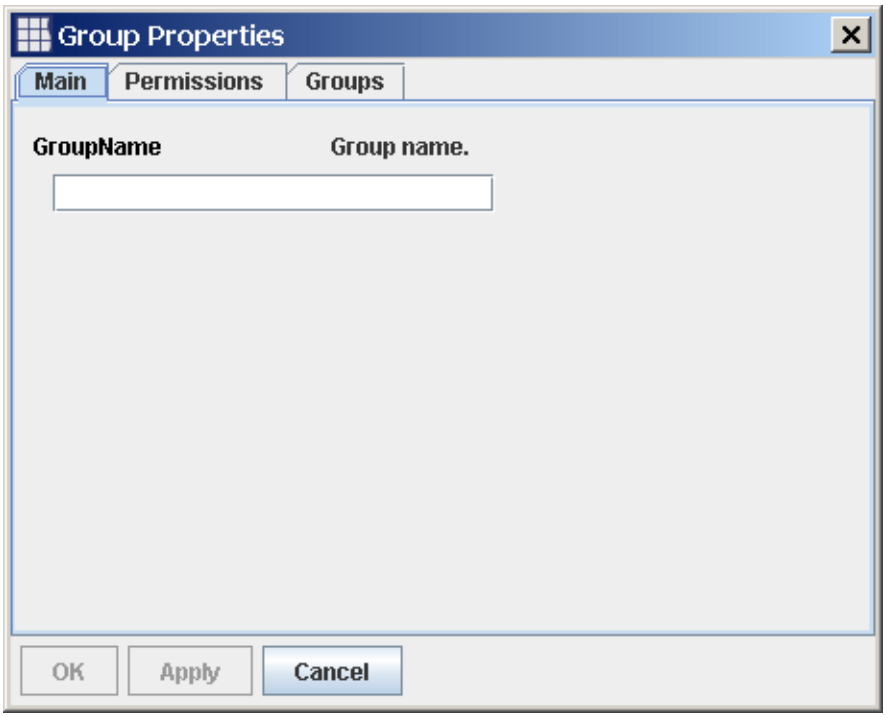
- You should be familiar with **Navigating the Tree View** on page 49.
- You should be familiar with **Groups** in **Security Concepts** on page 91.

Before creating a Group you will have created the Permissions that you want to add to that Group, see **Creating a New Permission** on page 103, and What are Groups in **Security Concepts** on page 91.

To create a group

1. In the Tree View, navigate to the Permissions node at: `Console Root > JMS Grid > Security > Access > Groups`
2. In the Detail View, position the cursor away from the existing Groups. Click the right mouse button to open the Panel Menu and select **New**. The **Group Properties** dialog opens, see below.

Figure 50 Group Properties



Note: *If the elements displayed in the Detail View fill the screen such that it is not possible to position the cursor away from them, then right click the table heading in order to bring up the Panel Menu.*

3. Fill in the fields to define your new Group. The Group dialog has been divided into three tabs. Into the Main tab, enter a unique name for your new group. On the Permissions tab, select all of the Permissions to group together from the Permissions pull down. From the Groups tab, select all the groups you want to be parents of your new group. Your new Group will inherit all the permissions of these parent groups. A full description of the meanings of the fields is given in What are Groups?
4. Click the **OK** button then confirm the changes at the 'Apply Changes?' warning dialog.

3.5.14 Editing a Group

This section explains how to edit the properties of a Group.

Prerequisite

- You should be familiar with **Creating a Group** on page 105.

To edit a group

1. In the Tree View, navigate to the Groups admin object at: `Console Root > JMS Grid > Security > Access > Groups`
2. The Detail View will now show a table of all existing Groups. Select the Group you wish to edit, then click the right mouse button.

Note: *You are not allowed to edit the pre-defined 'Super' Group. If you try a warning dialog will open telling you than that item's properties cannot be altered. You can, however, edit the other pre-defined groups.*

3. From the Item Menu that appears, select Properties. The Group Properties dialog will appear.

Note: *The Groups Properties dialog has an Inherited Permissions tab if the Group you are editing has a parent Group. The Inherited Permissions tab shows a read-only list of which Permissions your Group has inherited from its parent Group.*

4. Modify the required properties. The meanings of the property fields are explained in What are Groups?
5. When the edits are complete, click the **OK** or **Apply** buttons. A prompt for saving the changes appears. Click the **OK** button to apply the changes to the Group.

Another Way to open the Properties Dialog

You can open an item's properties dialog by selecting it in the Tree View and then clicking on the Properties button in the Toolbar.

3.5.15 Deleting a Group

When Users are no longer using a Group, it is possible to remove it from the JMS Admin Object Store.

Prerequisite

- You should be familiar with **Creating a Group** on page 105.

To delete a group

1. In the Tree View, navigate to: `Console Root > JMS Grid > Security > Access > Groups`
2. In the Detail View, select the Group instance you wish to delete

Note: *If you want to delete multiple Groups at the same time, you can select multiple Groups by holding down the Control key as you select each of the Groups you want to delete.You are not allowed to delete the pre-defined Group called 'Super'.*

3 With the mouse still over that Group, right mouse button click, and from the pop up Item Menu that appears, select Delete.

4 Click **Yes** in the confirmation dialog that appears. The Group will now be deleted from the chosen Admin Store.

Note: *If the Group that is deleted is still associated with a User, or is a member of another Group, then the User or owning Group will still refer to that deleted Group. If the Group that is deleted is a parent of child Group, then the child Group's parent becomes 'none'.*

3.5.16 Creating a User

Users are the security objects that ultimately determine whether all authentication and authorization is successful or not.

Note: *Only experienced administrators should create User accounts. Cyclic permission inheritance may lock the entire application.*

Prerequisites

- You should be familiar with **Navigating the Tree View** on page 49.
- You should be familiar with What are Users in **Security Concepts** on page 91.

Before creating a User you will have created the Permissions and Groups of Permissions that you will want to associate with that User, see How To create a new Permission, and How To create a Group.

To create a User

1 In the Tree View, navigate to the Permissions node at: `Console Root > JMS Grid > Security > Access > Users`

2 In the Detail View, position the cursor away from the existing Users. Click the right mouse button to open the Panel Menu and select New. The User Properties dialog opens, see below.

Figure 51 New User - Properties

Note: *If the elements displayed in the Detail View fill the screen such that it is not possible to position the cursor away from them, then right click the table heading in order to bring up the Panel Menu.*

3 Fill in the fields to define your new User.

Note: *The User Properties dialog has been divided into three tabs. In the Main tab, enter a unique name for your new User and a password for that User. Retype the same password in the VerifyPassword field,. If the VerifyPassword does not match the password, you will not be allowed to create that User. The password must be at least 6 characters long. In the Permissions tab, select all of the Permissions that User will inherit directly. From the Groups tab, select all the groups of permissions that User will further inherit. A full description of the meanings of the fields is given in What are Users?*

4 Click the **OK** button then confirm the changes at the 'Apply Changes' warning dialog.

3.5.17 Creating an Administrator

An Administrator is an especially powerful User that is allowed to log in to the JMS Grid Admin Tool and hence is able to edit the configuration of a JMS Grid Message Server Instance.

The JMS Gird default administrative User is called "admin." You should change this default password.

Prerequisites

- You should be familiar with **Creating a User** on page 108.
- You should be familiar with **Changing a User's Password** on page 111.

To create an Administrator

1. Follow the instructions in Create a User.
2. With the User Properties dialog box open select the Permissions tab. From the Permissions pull down menu, select the Administrator permission.
3. Complete the User Properties dialog as per Create a User.

3.5.18 Editing a User's Access Rights

This section explains how to edit the access rights of a User – that is, the destinations that the User is allowed to read messages from and send messages to.

Prerequisite

- You should be familiar with **Creating a User** on page 108.

To edit a User's rights

1. In the Tree View, navigate to the Users admin object at `Console Root > JMS Grid > Security > Access > Users`.
2. The Detail View will now show a table of all existing Users. Select the User to edit. Click the right mouse button to bring up the Item Menu.

Note: *You are not allowed to edit the access rights of the admin User. If you try, a warning dialog will open telling you that item's properties cannot be altered.*

3. From the Item Menu that appears, select **Properties**. A modified **User Properties** dialog will appear. The modified **User Properties** dialog displays the username, but will not let you change it. This modified dialog does not have the password fields. This is explained in How To change a User's password.
4. Modify the access rights by changing the available Permissions and Groups that User is associated using the lists on the **Permissions** and **Groups** tabs.
5. When the edits are complete, click the **OK** or **Apply** buttons. A prompt for saving the changes appears. Click the **OK** button to apply the changes to the Group.

Note: *That User must log out of the system and log back into it again before the new access rights will take effect.*

Another Way to open the Properties Dialog

You can open an item's properties dialog by selecting it in the Tree View and then clicking on the Properties button in the Toolbar.

3.5.19 Changing a User's Password

This section explains how to change a User's password.

Prerequisite

- You should be familiar with **Creating a User** on page 108.

To change a User's password

1. In the Tree View, navigate to the Users admin object at: `Console Root > JMS Grid > Security > Access > Users.`
2. The Detail View will now show a table of all existing Users. Select the User whose password you want to edit. Click the right mouse button to bring up the Item Menu
3. From the Item Menu that appears, select Change Password. The User Password dialog will open.
4. Into the Password field, enter the new password. Retype this password into the VerifyPassword field so that it is exactly the same as per the password field.

Note: *The capitalization of passwords is important. Passwords must be at least 6 characters long.*

5. When the edits are complete, click the **OK** or **Apply** buttons. A prompt for saving the changes appears. Click the **OK** button to apply the changes to the Group.

Note: *If there are any differences between the password and its verification, then you will see a warning dialog telling you that the verification field did not match the password.*

See also

- **Changing a Password Without being an Administrator** on page 117.

3.5.20 Re-enabling a User's Account

JMS Grid enforces a "three strikes and out" policy. If a messaging client makes more than three attempts to log into a User account using the wrong password, then that User account is disabled. This is intended to deter automated dictionary attacks.

Note: *The three strikes rule does not apply to administrator logins to the JMS Grid Admin Tool. These are performed manually. As such, they are less susceptible to dictionary attacks and are therefore exempted.*

Prerequisite

- You should be familiar with **Creating a User** on page 108.

To enable a User's account

1. In the Tree View, navigate to the Users node. In the Detail View you will see that for the disabled account, the entry in the Account Enabled column is false.
2. In the Detail View, select the row of the disabled User.

3 Right mouse click this row. In the Item menu, the Enable Account option will be active. Select this option.

4 In the Detail View, the User's account enabled status will revert to true.

3.5.21 Deleting a User

When a User is no longer allowed access to your system, you will want to clean that User admin object out of the admin object store.

Prerequisite

- You should be familiar with **Creating a User** on page 108.

To delete a User

1 In the Tree View, navigate to: `Console Root > JMS Grid > Security > Access > Users.`

2 In the Detail View, select the User you wish to delete.

Note: *If you want to delete multiple Users at the same time, you can select multiple Users by holding down the Control key as you select each of the Users you want to delete. You are not allowed to delete the admin User nor the default User.*

3 With the mouse still over that User, right mouse click, and from the pop up Item Menu that appears, select **Delete**.

4 Click **Yes** in the confirmation dialog that appears. The User will now be deleted from your admin store.

Note: *If a User is deleted while he or she is still logged into the system, then they will be able to continue working as normal. However, once they have logged out, they will be unable to log in again.*

3.5.22 Sending Encrypted Messages

A message from a client application is automatically encrypted if it is sent to a destination that is *secure*. A destination is made secure by creating a Destination Security admin object that has a name matching the name of a JMS destination.

Prerequisite

- You should be familiar with **What are Secure Destinations?** on page 96.

To send encrypted messages a Secure Destination must be created that has a name which matches the Destination Name of a JMS destination (or temporary destination) to which the encrypted messages are to be sent. See Create a Secure Destination object.

3.5.23 Creating a Secure Destination Object

A Secure Destination object is used to tell JMS Grid that messages sent to a JMS destination that matches this name may be encrypted.

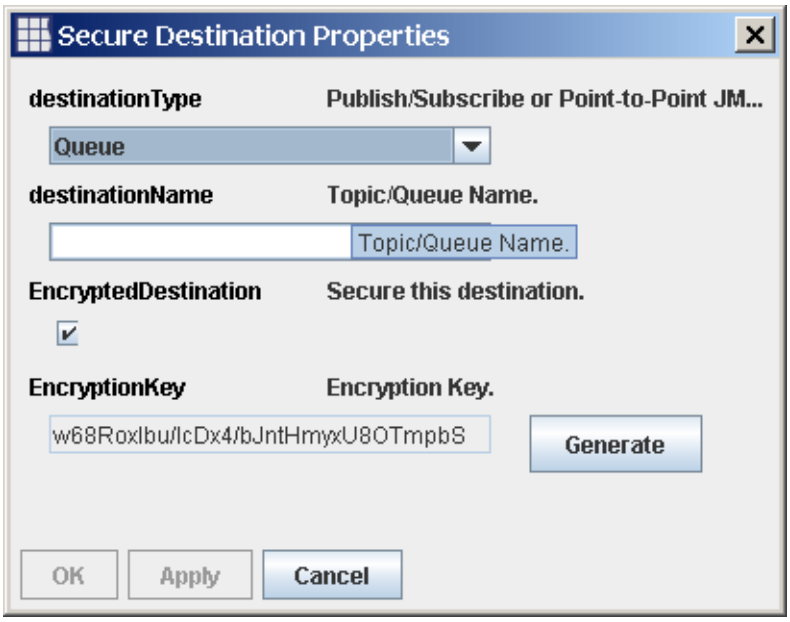
Prerequisites

- You should be familiar with **Navigating the Tree View** on page 49.
- You should be familiar with **What are Secure Destinations?** on page 96.

To create a secure destination

1. In the Tree View, navigate to: `Console Root > JMS Grid > Security > Destinations Security.`
2. In the Detail View, position the cursor away from any existing Destination Security objects. Click the right mouse and select New from the Panel Menu that appears. The Secure Destinations Properties dialog will open, see below.

Figure 52 Secure Destination Properties



Note: *If the elements displayed in the Detail View fill the screen such that it is not possible to position the cursor away from them, then right click the table heading in order to bring up the Panel Menu.*

3. Complete the fields to define your new Secure Destination object. A full description of the meanings of the fields is given in What are Secure Destinations?

Note: *Users may select different types of destination from the* `DestinationType` *combo box; this selection will affect the behavior of the* `DestinationName` *combo box. For example, if a Queue item is selected as a connection type then* `DestinationName` *combo box will contain the names of Queues. The selection content of* `DestinationName` *may be empty; this either means that there are no Destinations stored in JNDI for the connection type selected, or all of the Destinations for the selected type already have security profiles.*

`DestinationName` is an editable combo box. This allows Users to create security profiles for Destinations that may be created later.

The Encryption key is generated automatically using global *Security Settings*. This key can be regenerated using **Generate Key** button.

4 Click the **OK** button then confirm the changes at the 'Apply Changes' warning dialog.

3.5.24 Making all Existing Destinations Secure

A system may contain many Destinations. To make each Destination secure using the procedure outlined in How To create a Secure Destination object, could be very laborious. This How To describes a short cut that, in one step, creates a secure destination for all existing destinations. This procedure will not automatically make temporary Destinations secure since these are not known until runtime.

Prerequisite

- You should be familiar with **Creating a JMS Destination** on page 133.

To secure all destinations

1 In the Tree View, navigate to: `Console Root > JMS Grid > Security > Destinations Security`.

2 In the Detail View, position the cursor away from any existing Destination Security objects. Click the right mouse button and select **Secure All Destinations**.

Note: *If the elements displayed in the Detail View fill the screen such that it is not possible to position the cursor away from them, then right click the table heading in order to bring up the Panel Menu.*

3 In the Detail View a Destination Security object will be created for each pre-defined JMS Destination administered object.

3.5.25 Editing a Secure Destination Object

This section explains how to edit the properties of a Secure Destination object.

Once a Secure Destination object has been mapped to an existing JMS destination, it is not possible re-allocate it to another JMS destination. However, encryption to this destination can be turned on or off, and a new encryption key can be generated, if required.

Important: *With the latest version of JMS Grid it is recommended that Secure Destination objects are not edited. This is because key versioning is not currently provided. If a key changes then some clients will be unable to read messages.*

Prerequisite

- You should be familiar with **Creating a Secure Destination Object** on page 112.

To edit a destination object

1. In the Tree View, navigate to the Destinations Secure admin object at: `Console Root > JMS Grid > Security > Destinations Security.`
2. The Detail View will now display a table of all existing Secure Destinations. Select the Secure Destination you want to edit. Click the right mouse button to bring up the Item Menu.
3. From the Item Menu that appears, select **Properties**. The Secure Destination Properties dialog will appear.
4. Modify the required properties. The meanings of the property fields are explained in What are Secure Destinations?
5. When the edits are complete, click the **OK** or **Apply** buttons. A prompt for saving the changes appears. Click the **OK** button to apply the changes to the Group.

Another Way to open the Properties Dialog

You can open an item's properties dialog by selecting it in the Tree View and then clicking on the Properties button in the Toolbar.

3.5.26 Deleting a Secure Destination Object

When a User is no longer required or allowed access to the system, it is possible to remove that User admin object from the admin object store.

Note: *Subject to caching on the daemon or client, a client will stop encrypting messages destination immediately after that destination's Secure Destination object is deleted in the Admin Tool.*

Prerequisite

- You should be familiar with **Creating a Secure Destination Object** on page 112.

To delete a secure destination object

1. In the Tree View, navigate to: `Console Root > JMS Grid > Security > Destinations Security.`
2. In the Detail View, select the Secure Destination instance to delete.

Note: *If you want to delete multiple Secure Destinations at the same time, you can select multiple Secure Destinations by holding down the Control key as you select each of the Secure Destinations you want to delete.*

3. With the mouse still over that Secure Destination, right mouse click, and from the pop up Item Menu that appears, select **Delete**.
4. Click **Yes** in the confirmation dialog that appears. The Secure Destination will now be deleted from the chosen admin store.

3.5.27 Tightening JMS Grid's Security

JMS Grid's security package gives administrators fine-grained control over who is allowed to do what within a JMS Grid system. However, out of the box, JMS Grid is a very open system. This enables new Users to quickly get started with it. However, for a secure, commercial and sensitive system there are a number of steps it is important to take so that your system is not left vulnerable to attack.

Prerequisites

- You should be familiar with **JMS Grid Security** on page 90 and **Security Concepts** on page 91.

3.5.28 Changing the Admin User Password

Anyone in your organization who has access to this guide will easily be able to find the initial password to the all-powerful admin User. See How To change a User's password.

Important: *It is **vitally** important that you don't lose this password. All other passwords in the system can be changed if the original is forgotten. If the admin User's password is forgotten – and you have not created any other Administrators – then your position is irretrievable. You will have to re-create the admin store for your JMS Grid system and start all over again!*

3.5.29 Changing the Permissions of the Default User

When a messaging client's security is disabled, or if a secure client attempts to log on using a non-existent User, JMS Grid reverts to the default User. The default User has been given a generous set of Permissions that allows anonymous log-in and the ability to read and write messages from all permanent and temporary Destinations. So in its out-of-the-box state, JMS Grid can give more access to non-existent Users and unsecured clients than it gives to known, secure clients. See **Editing a User's Access Rights** on page 110.

Alternatively, the default Group can be edited to remove Super as its parent Group – see **Editing a Group** on page 106.

3.5.30 File Security on the Administration Object Store

Although only JMS Grid Users with *Administrator* Permission can log in to the JMS Grid Admin Tool, unauthorized Users may be able to gain access to your configuration data using other software. Depending on which storage mechanism is being used to store configuration data, the unauthorized User could use a different JNDI browser, LDAP browser or even just a simple file browser to access and alter your precious configuration. If using a JNDI or LDAP admin store, refer to your implementation's documentation for ways to prevent this sensitive information being compromised.

3.5.31 Changing a Password Without being an Administrator

If you have administrator security privileges, you can open the JMS Grid Admin Tool and change any User's password

However, if you don't have administrator privileges you cannot open the JMS Grid Admin Tool. How can you change your own password?

You can edit your own password using a command line utility program that is run separately from the JMS Grid Admin Tool.

Prerequisite

- You should be familiar with **Creating a User** on page 108.

To change a password

1. Ensure that JMS Grid security has already been enabled on a server – see **Enabling JMS Grid Security** on page 99.
2. Ensure that one or more JMS Grid Message Daemon's are running somewhere on your system. You will need the URL of one of these daemons to feed into the password utility. (The URL is also know as the Resource Location, Bind Address or Message Channel).
3. Open a command shell (or Windows command prompt) and navigate to the root of your JMS Grid installation.
4. At the command line type:

On Windows

```
cpass daemonURL
```

On Unix

```
cpass daemonURL
```

where `daemonURL` is the URL of one of the running daemons in your JMS Grid system.

Example

```
C:\JavaCAPS51\JMS_Grid>cpass tcp://localhost:50607
```

5. Follow the command line prompts to change your password. In the example below, the sample User input is shown in bold type.

Example

```
Please enter User Name: myUserName
Please enter Old Password: myOldPassword
Please enter New Password: myNewPassword
2002-09-03 15:29:02,433 INFO   - Connected to JMS Grid Message Daemon
(testDaemon@ tcp://mikebpc:50607) version 5.2
The password was successfully changed...
```

See also

- **Changing a User's Password** on page 111.

3.6 SSL Configuration

Data sent over a network is vulnerable to interception and corruption. If the data is important and includes sensitive information you might want to take steps to keep it private and maintain its integrity in transit.

3.6.1 What is SSL?

The Secure Sockets Layer (SSL) and the Transport Layer Security (TLS) protocols are designed for this purpose. JMS Grid allows you to send messages over an SSL connection so that you can make your JMS communications more secure.

Some Concepts

The SSL protocol consists of several elements:

- Encryption of communications for security
- Optional authentication to verify the identity of the other party
- Check sum facilities for data integrity

When an SSL connection is made, first of all client and server go through a process of handshaking. If this is completed successfully then encrypted communications commence. Very simply, the handshake consists of negotiating a cipher suite, establishment of identity and agreeing encryption methods. The cipher suite is negotiated transparently between client and server, where the server chooses to use the best mutually acceptable suite. At the authentication phase both parties can optionally authenticate themselves by presenting a digital certificate. It is common for the server to do this, but much less so for the server to demand this of the client. Authentication means that the party receiving the certificate checks that it corresponds to whom it thinks the server is and is verified by a certificate authority it trusts. Finally, if authentication is successful then an encryption key is established so secure communication can occur.

Any SSL configuration you do mainly affects what happens during the handshaking phase and concerns authentication. In JMS Grid the daemon must authenticate itself by default, but client authentication is optional. In order to authenticate itself a party to an SSL connection needs access to a certificate and it also needs a public/private key pair for the early part of the handshake. These are to be found in a *key store*. The party in receipt of the certificate also needs information about who to trust, which it will find in a special type of key store called a trust store. Configuring SSL for JMS Grid means ensuring the daemon and clients know where all these items are to be found, and setting the client authentication policy.

For more information about SSL and TLS see:

http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

Key and Trust Stores for JMS Grid

JMS Grid uses the following settings for key manager algorithm, trust manager algorithm and key store type:

Key Manager algorithm: default, as defined by the `ssl.KeyManagerFactory.algorithm` property in `<java-home>/lib/security/java.security`

Trust Manager algorithm: default, as defined by the `ssl.TrustManagerFactory.algorithm` property in

`<java-home>/lib/security/java.security`

KeyStore type: default, as defined by the `keystore.type` property in

`<java-home>/lib/security/java.security`

We do not describe here how to set up key and trust stores or obtain digital certificates. See the system documentation with your jre to explain how to do this. For Sun Microsystem jre's the keytool utility should be used. It is documented at `http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html` for Solaris and `http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html` for Windows. A useful example is given at `http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html#CreateKeystore`

Special Note about Sample Key and Certificates

We supply a sample key store for the daemon but this has only a test key entry, mainly for use with the examples. This allows the daemon to authenticate itself to any client which has the corresponding certificate in its trust store. There is also a sample certificate for use by the daemon when it is acting as a client. The situations where this applies are when SSL is used to make cluster and network connections. Additionally, due to the placement of this certificate in the embedded jre's trust store it allows SSL to be used for the JMX commands.

These keys and certificates are for testing only and should never be used in a deployed system. Please consult system documentation, as mentioned above, and certification authorities for assistance to do this.

Special Note about SSL Provider Pluggability

JMS Grid is supplied with an embedded jre, which is in most cases one from Sun Microsystems, or otherwise from IBM. Early versions of the Sun Microsystems jre, i.e. before 1.4, allowed different SSL implementations to be plugged in, at least in the jre licensed for use in the United States. When the Java Secure Server Extension became a standard part of the jre it was the non-pluggable version that was bundled with it. This remained true until jre 5.0 update 6 was released in December 2005. It is therefore likely that the jre provided with JMS Grid does not allow other SSL providers to be plugged in. At the present time this would be an unsupported JMS Grid SSL configuration.

For further explanation of the pluggability restrictions of SSL see:

http://blogs.sun.com/roller/page/andreas/20051202

3.6.2 Configuring the Daemon's use of SSL

There are two ways in which you can configure the daemon's use of SSL. Which one you use depends on your personal preferences. These methods are:

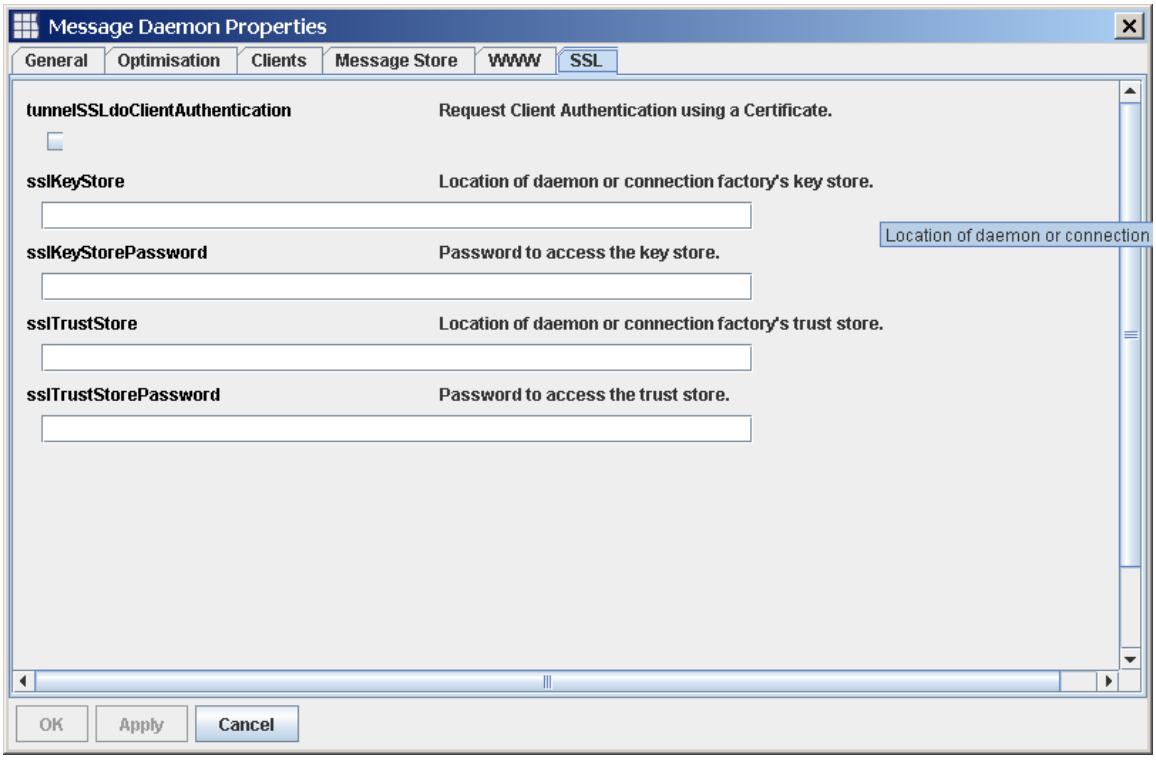
- Via the administration tool
- Via a property file

Configure Daemon SSL using the Administration Tool

The first thing you need to do is create a daemon which will listen for SSL connections. This is explained in: **Configuring a Single Daemon** on page 54 and **Specifying a Daemon Network URL** on page 63.

You should have the Message Daemon Properties dialog open during the creation of a daemon. Click on the **SSL** tab and the dialog will look like this:

Figure 53 Message Daemon Properties



There are potentially five things you need to fill in here:

- `DoClientAuthentication` check this box if you want clients to be required to authenticate themselves. The default value is false.
- `SSLKeyStore` and `SSLKeyStorePassword`. This is the file which contains the daemon's own certificate with which it will authenticate itself and its key pair. You

need to specify its location and password, the location should be available on the daemon's classpath. The easiest way to do this is to put the key store in the properties directory. You must provide values here or the daemon will fail to bind to the SSL port.

- `SSLTrustStore` and `SSLTrustStorePassword`. This is the file which contains the certificates of trusted parties. It is only relevant when using client authentication. You need to specify its location and password, where the location is available on the daemon's classpath. The easiest way to do that is to put it in the *properties* directory. If you do not specify a value then the file `<java-home>/lib/security/jssecacerts` or `<java-home>/lib/security/cacerts` will be used with the default password, "changeit". If the first file does not exist then the second will be used, if that does not exist then an empty trust store will be created. Don't forget that the daemon has its own jre, in the jre directory. Also, don't forget when you are doing this about the need for daemons to authenticate themselves to each other in cluster and network connections: whatever file is used needs to have the daemon's trusted certificate in it.

Configuring Daemon SSL using a Property File

Prerequisite

- You should be familiar with **Configuring a Daemon from a Properties Text File** on page 75.

Make sure you have added an SSL location to the list of `bindAddresses`. Then there are five properties you may need to give values for to configure SSL use:

sslKeyStore and sslKeyStorePassword

The key store is the file which contains the daemon's own certificate with which it will authenticate itself and its key pair. You need to specify its location and password, the location should be available on the daemon's classpath. The easiest way to do this is to put the key store in the *properties* directory. You must provide a value here or the daemon will fail to bind to the SSL port.

sslTrustStore and sslTrustStorePassword

The trust store is the file which contains the certificates of trusted parties. It will only be relevant when using client authentication. You need to specify its location and password, where the location is available on the daemon's classpath. The easiest way to do that is to put it in the *properties* directory. If you do not specify a value then the file `<java-home>/lib/security/jssecacerts` or `<java-home>/lib/security/cacerts` will be used with the default password, "changeit." If the first file does not exist then the second will be used, if that does not exist then an empty trust store will be created. Don't forget that the daemon has its own jre in the jre directory.

Also, don’t forget when you are doing this about the need for daemons to authenticate themselves to each other in cluster and network connections. Whatever file is used needs to have the daemon’s trusted certificate in it.

tunnelSSLdoClientAuthentication

Set to true if you want clients to be required to authenticate themselves. Defaults to false.

3.6.3 Configuring the Client's use of SSL

There are three ways in which you can configure the daemon's use of SSL. Which one you use depends on your personal preferences and to some extent how the daemon will use SSL. These methods are:

- Via the administration tool
- Via a property file
- Via system properties

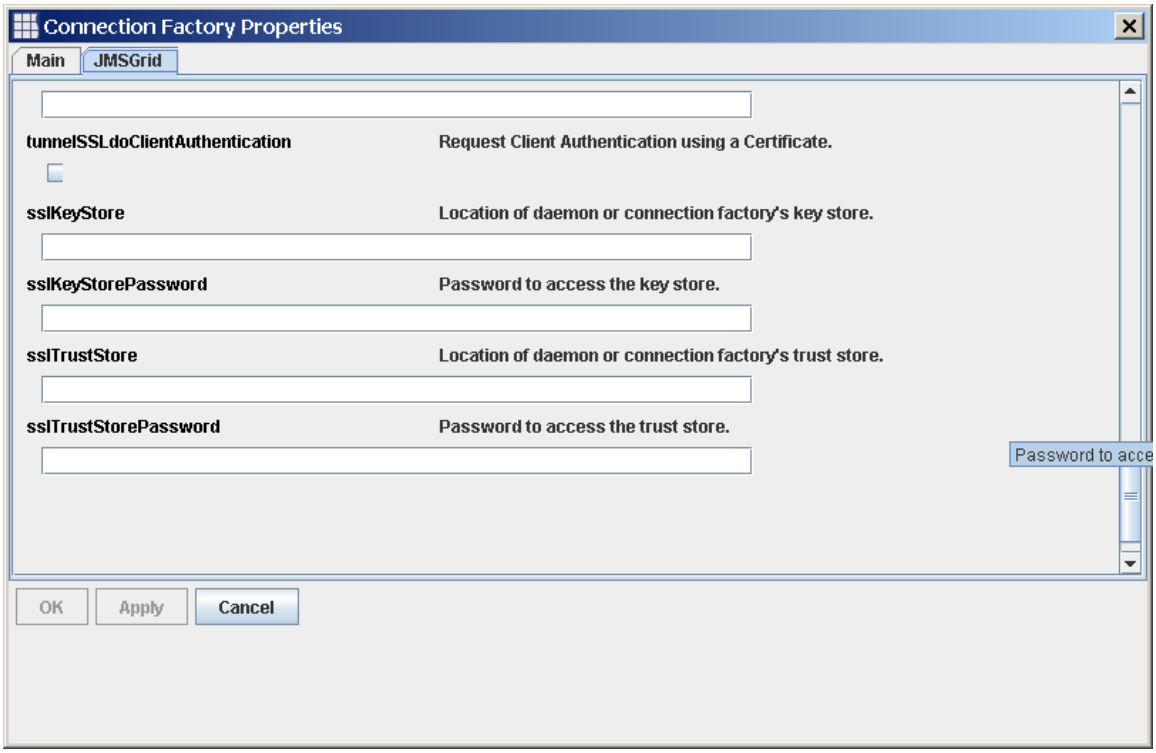
System properties cannot be used where the daemon is demanding client authentication.

Configuring the Client's SSL using the Administration Tool

The first thing you need to do is create a connection factory which will use SSL to connection to the daemon. This is explained in **Creating a Connection Factory** on page 125 and **Editing Connection Factory Properties for a Normal JMS Grid Client Connection** on page 131.

You should have the Connection Factory Properties dialog open during the creation of a connection factory. Click on the JMS Grid tab and scroll down to the bottom and the dialog will look like this:

Figure 54 Connection Factory Properties



There are potentially five things you need to fill in here:

- `DoClientAuthentication` tick this box if you want clients to be required to authenticate themselves. Defaults to false.
- `SSLKeyStore and SSLKeyStorePassword`. This is the file which contains the daemon's own certificate with which it will authenticate itself and its key pair. You only need to specify these values if the daemon is using client authentication. You need to specify its location and password. The location must be somewhere on the client's classpath. You must provide a value here or the client will fail to bind to the SSL port.
- `SSLTrustStore and SSLTrustStorePassword`. This is the file which contains the certificates of trusted parties. You need to specify its location and password, where the location is on the client's classpath. If you do not specify a value then the file `<java-home>/lib/security/jssecacerts` or `<java-home>/lib/security/cacerts` will be used with the default password, "changeit." If the first file does not exist then the second will be used, if that does not exist then an empty trust store will be created. Also, don't forget when you are doing this about the need for daemons to authenticate themselves to each other in cluster and network connections: whatever file is used needs to have the daemon's trusted certificate in it.

Configuring Client SSL using a Property File

Prerequisite

You should be familiar with how to**Configuring a Daemon from a Properties Text File** on page 75.

You should have have added an SSL location to the list of `bindAddresses`. Then there are five properties you may need to give values for to configure SSL use:

privateKeyFilename and privateKeyPassword

The first property is the file which contains the client's own certificate, with which it will authenticate itself, and its key pair and the second is its password. You only need to specify these values if the daemon is using client authentication. You need to specify its location and password. The location must be somewhere on the client's classpath. You must provide a value here or the client will fail to bind to the SSL port

tunnelSSLTrustStore and tunnelSSLTrustStorePassword

The trust store is the file which contains the certificates of trusted parties. You need to specify its location and password, where the location is somewhere on the client's classpath. If you do not specify a value then the file `<java-home>/lib/security/jssecacerts` or `<java-home>/lib/security/cacerts` will be used with the default password, "changeit". If the first file does not exist then the second will be used, if that does not exist then an empty trust store will be created.

tunnelSSLdoClientAuthentication

Set to true if you want clients to be required to authenticate themselves. Defaults to false.

Configuring Client SSL use with System Properties

The JSSE specification defines four system properties which can be used to define the locations and passwords for the key and trust store. These properties are:

```
javax.net.ssl.keyStore
javax.net.ssl.keyStorePassword
javax.net.ssl.trustStore
javax.net.ssl.trustStorePassword
```

You can set the values of these system properties for your client programme and they will work in the same way as setting them via a property file or using the administration tool.

Using SSL Clients from SpiritWave with JMS Grid

If you are using client programs built with earlier versions of JMS Grid, which was then known as SpiritWave, and you want to use them with a JMS Grid daemon then you need to take some extra steps. As you will probably be aware, SpiritWave did not use key and trust stores but relied on keys and certificates being available each in their own individual file. These are the steps that you need to take:

- Export the daemon's certificate from its trust store. SpiritWave expects keys and certificates to be present in individual files. You can export the daemon's certificate by using the keytool utility. It is documented at **http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/keytool.html** for Solaris and **http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html** for Windows. Here is an example of how to do the export:

```
keytool -export
        -alias <daemon's certificate alias>
        -file <filename>.der
        -keystore <key store file name>
```

You need to know the daemon's certificate alias, which will be the one which was used when the certificate was created or imported. You can list all the aliases of entries in a key store using the `-list` option of keytool. Note that certificates are exported in binary DER format by keytool, so use this as the file's suffix. The daemon's certificate will be found in its key store, as defined by the `sslKeyStore` property.

- Make the exported certificate available to the client. In SpiritWave the client property `clientRootCACertFilename` should be set to the name of the der file you created in the first step. You can do this either by using a property file or by setting the property on a connection factory created using the Administration Tool. Client programmes expect to find this file somewhere on the class path.
- Check the SSL URL. When connecting from SpiritWave to a JMS Grid daemon you can no longer use 'localhost' as the host name when using SSL. You must use the hostname of the machine. So, for example, connecting to the URL `ssl://localhost:444`, which works with SpiritWave client and daemons, you will need to change this to `ssl://<machine-name>:444` when connecting a SpiritWave client to a JMS Grid daemon. If you forget to do this you will get a "Connection refused" error.

3.7 JMS Administration

3.7.1 Introduction

This section explains how to use the JMS Grid Administration Tool to set up JMS Administered Objects. There are two kinds of JMS Administered Objects:

1. Connection Factories
2. Destinations

The JMS Grid Administration Tool can be used to set up JMS Connection Factories regardless of the underlying message infrastructure that is being used. It is only used to create Destinations when the JMS Grid server is being used for the underlying messaging infrastructure.

3.7.2 Creating a Connection Factory

As the name suggests, a Connection Factory is an object that is used to manufacture new connections.

Each of these connections is used to provide a communications link between a JMS messaging client and a JMS message server.

JMS Grid allows a client to either:

1. Retrieve a previously created Connection Factory from the Admin Tool JMS Admin Object Store.

   or

2. Programmatically create and configure a Connection Factory

Programmatic creation of connection factories is not part of the standard JMS specification and is explained in detail in the JMS Grid Programmer's Guide.

This section describes how an administrator can create and configure a Connection Factory that will be subsequently looked up by a messaging client. This is the standard mechanism specified by the JMS specification for a client to obtain a Connection Factory.

A JMS messaging client uses JNDI to look up a Connection Factory that a JMS Grid administrator has created, configured and stored. The Connection Factory will be stored in the Admin Tool JMS Admin Object Store. See **Specifying how Configuration Data is Stored** on page 139.

One of the key features of JMS Grid is the ability to allow developers to choose a different vendor's underlying messaging infrastructure. It is in the Connection Factory that the administrator specifies which messaging provider is to be used. When the provider has been chosen, the Admin Tool enables the administrator to set the properties of the vendor specific Connection Factory.

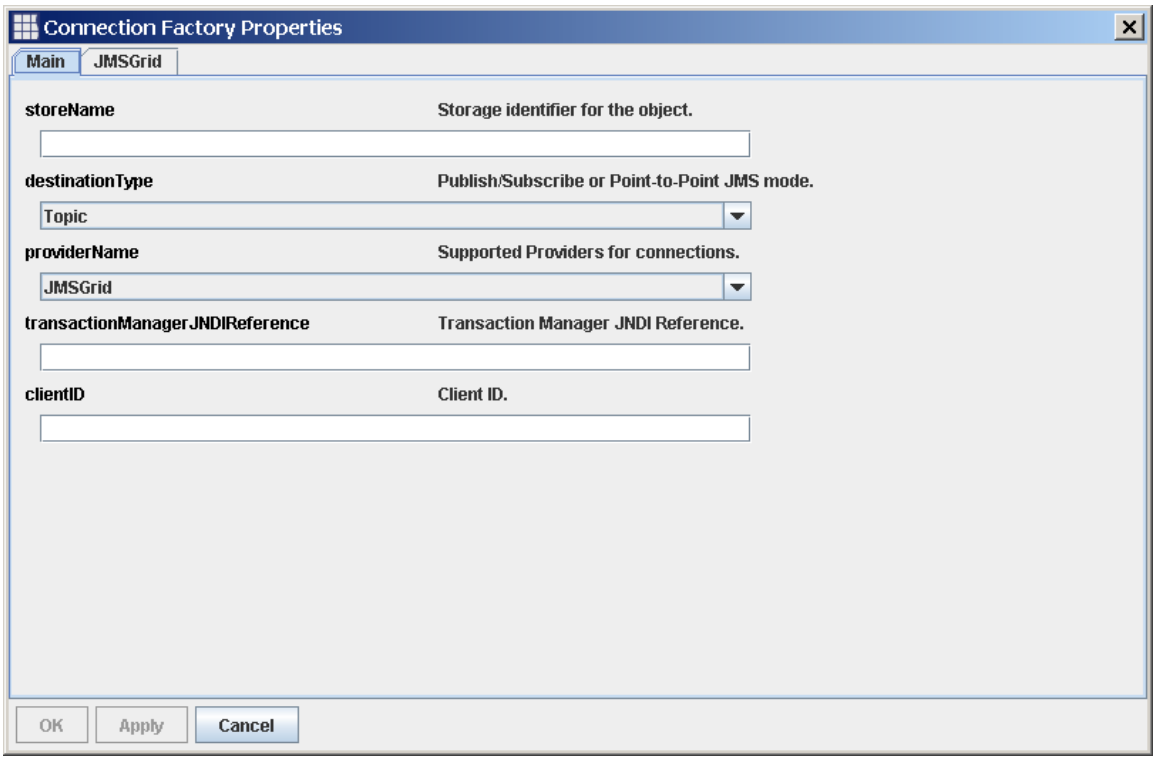
Prerequisite

- You should be familiar with **Navigating the Tree View** on page 49.

To create a connection factory

1. In the Tree View navigate to: `Console Root > JMS Grid > Administered Objects > Connection Factories.`
2. In the Detail View, position the cursor so that it is not over any existing Connection Factories. Right mouse click to display the Panel Menu. From the Panel Menu, select **New**, to open the **Connection Factory Properties** dialog box, see below. If the Detail View is full, you can right mouse click over the Detail View's Column headings to open the Panel Menu.

Figure 55 Connection Factory Properties



3. Fill in the following fields in the Connection Factory Properties dialog box - view table below.

Table 22 Connection Factory Properties

Name	Description	Default Value	Required
`storeName`	A unique JNDI store name for a Connection Factory. JMS messaging clients will use this JNDI name to lookup an instance of a Connection Factory.		Y

Table 22 Connection Factory Properties (Continued)

Name	Description	Default Value	Required
`destinationType`	Destination type - Queue or Topic. If message sending is to be involved in a distributed transaction then either XAQueue or XATopic must be selected.	Topic	Y
`driverName`	Set to "JMSGrid" (default) if the connection should connect to a remote daemon. Set to "JMSGridEmbedded" if the connection should use an daemon embedded within the same JVM.	JMSGrid	N
`clientID`	The clientID that a connection will be assigned when created by this factory.		N

Note: *The Apply and OK buttons will be greyed out until all the required fields have been specified. The fields available will depend on the value of driverName that is being used.*

Setting a Connection Factory's clientID

Note: *The Connection Factory clientID is a problematical part of the JMS specification. Before setting a clientID for a Connection Factory, it is important to consider the pros and cons of setting a clientID in the Connection Factory itself with your development team. The client ID is only used in conjunction with clients that use JMS Durable Subscriptions. Each durable subscriber client requires a client ID so that if the client crashes, the message server can identify a new client as the replacement (the replacement will have the same client ID as the original client). Thus, undelivered messages can be sent to this replacement. The JMS spec states that the preferred way to assign a client identifier is for it to be configured in a client specific Connection Factory. This means you will need to set up an individual Connection Factory for every durable messaging client. This is acceptable if only a few, well-known durable clients are required, but in a large, flexible system with thousands of durable clients, thousands of connection factories would potentially need to be created - one for each client - each one with a different clientID. If the clientID is left unset, on the other hand, individual messaging clients can programmatically set their own clientID using the Connection.setClientID(string) method call. Thus, all clients can potentially share the same Connection Factory, reducing the administrative overhead and making for a more flexible and scalable system.*

4 You must now fill in the required driver specific properties of the Connection Factory. These properties are entered on the other tab that is behind the Main tab on the Connection Factory Properties dialog. The name of this tab, and the properties available, will vary depending on which value is selected for the driverName property.

5 Once all the required fields have been filled in click the **OK** button. A prompt to save the changes appears. Click the **OK** button to apply the changes. A new Connection Factory will appear in the Detailed View.

See also

- **Editing Connection Factory Properties for a Normal JMS Grid Client Connection** on page 131

3.7.3 Editing a Connection Factory's Properties

This section explains how to edit the properties of a Connection Factory.

Prerequisite

You should be familiar with **Creating a Connection Factory** on page 125.

To edit a connection Factory's properties

1 In the Tree View, navigate to the Connection Factories admin object at: `Console Root > JMS Grid > Administered Objects > Connection Factories.`

2 The Detail View will now show a table of all existing Connection Factories. Right mouse-click over the table row that contains the Connection Factory you wish to edit.

3 From the Item Menu that appears, select Properties. The Connection Factories dialog will be displayed.

4 Modify the required properties. The meanings of the property fields on the Main tab are explained in **Creating a Connection Factory** on page 125. The properties on the driver specific tabs are explained further in the topics listed below.

5 When the edits are complete, click the **OK** or **Apply** buttons. A prompt for saving the changes appears. Click the OK button to apply the changes to the Connection Factory.

Another Way to open the Properties Dialog

You can open an item's properties dialog by selecting it in the Tree View and then clicking on the Properties button in the Toolbar.

See also

Editing Connection Factory Properties for a Normal JMS Grid Client Connection on page 131

3.7.4 Exporting a Connection Factory's Properties

This action displays a File Dialog, which prompts for a new properties file into which the Connection Factory's properties are exported.

It is possible to export Connection Factory settings into a Java properties file for debugging purposes.

Prerequisite

You should be familiar with **Creating a Connection Factory** on page 125.

To export a Connection Factory's properties

1. In the Tree View, navigate to the Connection Factories admin object at: `Console Root > JMS Grid > Administered Objects > Connection Factories.`
2. The Detail View will now show a table of all existing Connection Factories. Right mouse-click over the table row that contains the Connection Factory you wish to export.
3. From the Item Menu that appears, select **Export properties…** The **File** dialog will appear.
4. Select a suitable location and file name and click **OK**. Selecting an existing file name will overwrite the contents of that property file.

3.7.5 Creating Multiple Copies of a Connection Factory

If you want to create many similar Connection Factories you could repeat the steps explained in How To create a Connection Factory, many times over. However this would be a laborious process.

The JMS Grid Admin Tool provides you with a much quicker way to do this – it allows you to make multiple copies of an existing Connection Factory.

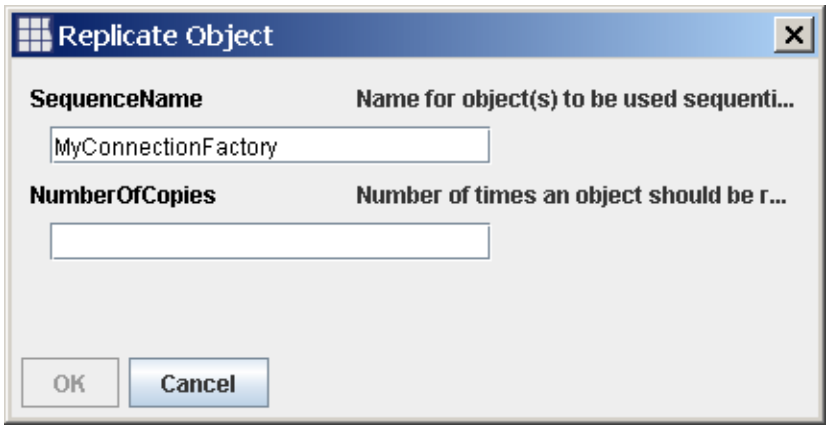
Prerequisite

You should be familiar with**Creating a Connection Factory** on page 125.

To create multiple copies of a Connection Factory

1. Create a single configuration for a Connection Factory as explained in Creating Connection Factory. Ensure that the configuration parameters of this configuration closely match the configurations of your copies before deciding to replicate it.
2. In the Detail View, select the Connection Factory to replicate. Click the right mouse button to bring up the **Item Menu**. From the Item Menu select **Replicate**. This will open the **Replicate Object** dialog, as shown below.

Figure 56 Replicate Object



3 In the **SequenceName** field, enter the name to be used for the replicated objects. The Admin Tool applies a simple numbering rule to the names of each of the replicas. A sequential number, starting from 0, is appended to the end of the SequenceName you give to create the replica's name. For example, if you entered MyConnectionFactory as the sequence name, and asked for 3 copies to be made, then the replicated daemons' store names would be *MyConnectionFactory0*, *MyConnectionFactory1* and *MyConnectionfactory2*.

4 In the **NumberOfCopies** field enter the number of copies to make.

5 Click the **OK** button.

6 In the Detail View list the sequence-named replicas will appear.

7 If a client ID had been set on your original Connection Factory, you will need to edit the client ID's for each of your replica Connection Factories. More information on Connection Factory client ID's is given in **Creating a Connection Factory** on page 125.

3.7.6 Deleting a Connection Factory

When messaging clients no longer use a redundant Connection Factory, it is possible to remove it from the JMS Grid Admin Tool (JMS Admin Object Store).

Prerequisite

You should be familiar with**Creating a Connection Factory** on page 125.

To delete a Connection Factory

1 In the Tree View, navigate to: `Console Root > JMS Grid > Administered Objects > Connection Factories.`

2 In the Detail View, select the Connection Factory instance to delete.

Note: *If you want to delete multiple Connection Factories at the same time, you can select multiple Connection Factories by holding down the Control key as you select each of the Connection Factories you want to delete.*

3 With the mouse still over that Connection Factory, right mouse button click, and from the pop up Item Menu that appears, select **Delete**.

4 Click **Yes** in the confirmation dialog that appears.

Note: *Once a Connection Factory has been deleted, new messaging clients that attempt to look it up will no longer be able to able to find it in the JMS Admin Object Store.*

3.7.7 Editing Connection Factory Properties for a Normal JMS Grid Client Connection

This section explains how to set up the Connection Factory properties that are specific for a normal JMS Grid client connection.

If you are using a JMS Grid embedded connection then different properties will be required.

The `driverName` property defines which type of connection is required.

Prerequisite

You should be familiar with**Creating a Connection Factory** on page 125.

To edit s server specific Connection Factory properties

1 Make sure the Connection Factory Properties dialog is open.

How to open this is explained in **Editing a Connection Factory's Properties** on page 128.

2 Ensure the `driverName` is set to "JMSGrid" on the dialog's main tab. This is the default.

3 The JMS Grid tab shows a number of fields - click the table icon below. Only the **messageChannels** field must be filled in before the Connection Factory can be created. All other fields are optional.

Table 23 Message Server Tab

Name	Description	Default Value	Required
`Clusters`	Clusters available for connections		N
`messageChannels`	A list of URLs used to connect to a daemon		Y
`localClientAddress`	Force a client to open a socket on a particular network interface with a specific port. Required for firewalls. Should be supplied in the form networkAddress:portNumber		N

Table 23 Message Server Tab (Continued)

Name	Description	Default Value	Required
randomConnection	If multiple connection URLs are specified in the messageChannels property then if randomConnection is set (which is the default) then the client will randomly choose a daemon to connect to. Otherwise it will choose the first daemon in the list.	true	N
defaultConnectionRetries	The number of times a client may retry connection		N
defaultConnectionRetries Timeout	Interval between connection retries in seconds		N
pingEnabled	Enable pinging between a client and a demon		N
pingTimeout	Interval between pings in milliseconds		N
proxyHost	Hostname or IP address of firewall proxy server		N
proxyPort	Port of firewall proxy server		N
userName	User name		N
password	User password		N
waveCloseConnectionOnSlowConsumer	Close connection on slow consumer	true	N
waveCloseSessionOnSlowConsumer	Close session on slow consumer	true	N
waveConsumerMessageQueueMaxSize	Maximum number of messages stored by a session		N
autoDiscoveryAllowed	Determines whether the daemon will register as a service for clients to automatically discover, and whether the daemon itself will use multicast discovery to locate other daemons.	false	N
clientSideTransactions	Enable transactions to be controlled on the client - as opposed to the server	true	N
consumeDispatchThread	Use a separate thread to dispatch messages from the connection to individual sessions	false	N
produceDispatchThread	Enable a thread on the Connection to manage dispatching to the daemon	false	N

The JMS Grid tab will only show the messageChannels (a.k.a. Network URLs) for the daemons that have configurations. To view these available messageChannels, a cluster must be selected from the Clusters pull down. When a cluster is selected, the messageChannels pull down will then show all the Network URLs of all the daemons in that cluster. In order to see available Network URLs of the single daemons, select 'None' as the Clusters option in the Clusters pull down.

3.7.8 Creating a JMS Destination

A JMS destination is a named storage area within the message server for messages that are in transit. A JMS messaging client sends messages to a particular named destination in the message server. Other JMS clients receive messages up from a particular named destination in the message server.

The administrator creates configurations for destinations that are stored as JNDI entries in the JMS Admin Object Store. JMS compliant messaging client applications can then lookup a destination's configuration. When the client sends a message to a destination with that configuration, the message server will create a physical destination.

Note: *The administrator is not creating the physical destination that stores messages in transit. The message server itself creates the physical destinations when clients first send messages to them.*

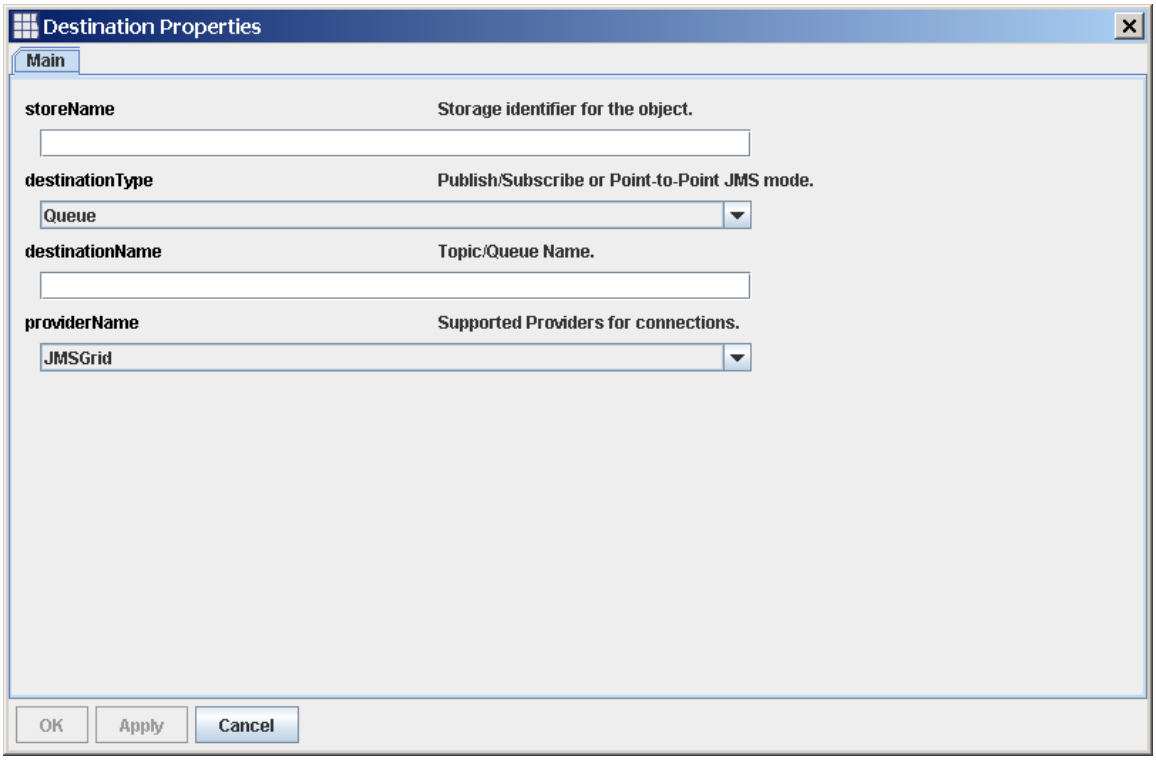
Prerequisite

You should be familiar with **Navigating the Tree View** on page 49.

To create a JMS destination

1. In the Tree View navigate to: `Console Root > JMS Grid > Administered Objects > Destinations.`
2. In the Detail View, position the cursor so that it is not over any existing destinations. Right mouse button click to display the Panel Menu. From the Panel Menu, select New, to open the Destination Properties dialog box, see below. If the Detail View is filled with existing destinations, right mouse button click when the cursor is over the Detail View column headings.

Figure 57 Destination Properties



3 Fill in the fields in the form to match the destination to create. See the table below for explanations about each of the fields.

Table 24 Creating a New Destination

Name	Description	Default Value	Required
`storeName`	A unique JNDI name for a Destination. The client's application then uses this name to lookup the destination in the JNDI tree.		Y
`destinationType`	The type of destination - a Queue or a Topic.	Queue	Y
`destinationName`	The JMS name of the destination.		Y
`providerName`	This property is no longer used and should be set to JMSGrid (default)	JMSGrid	Y

Store Name and Destination Name

A destination appears to have two names – a store name and a destination name. These names have different purposes.

The store name

The store name is simply the JNDI location of the destination. This is the place where messaging clients will lookup that destination. If someone changes the place

in the JNDI hierarchy where a destination is stored, then its store name will change to reflect the new location.

The destination name

The destination name is the identifier for that destination. This is the name that other objects use to refer to a given destination. For example, a destination is made secure by creating a separate Secure Destination object. The Secure Destination object is told which destination it is securing using its destination name. Another use of the destination name is when security Permissions are created – a Permission defines read/write access permissions to particular destinations. The destination name is used to specify the destination.

4 When the fields have been populated, click the OK button. A prompt for saving the changes appears. Click the OK button to apply the changes. A new destination will be displayed in the Detail View.

3.7.9 Editing a Destination's Properties

This section explains how to edit the properties of a destination.

Prerequisite

You should be familiar with **Creating a JMS Destination** on page 133.

To edit a destination's properties

1 In the Tree View, navigate to the Destinations admin object at: `Console Root > JMS Grid > Administered Objects > Destinations.`

2 The Detail View will now show a table of all existing Destinations. Select the Destination you wish to edit, then click the right mouse button.

3 From the Item Menu that appears, select Properties. The Destination Properties dialog will appear.

4 Modify the required properties.

5 When the edits are complete, click the **OK** or **Apply** buttons. A prompt for saving the changes appears. Click the OK button to apply the changes to the Destination.

Another Way to open the Properties Dialog

You can open an item's properties dialog by selecting it in the Tree View and then clicking on the Properties button in the Toolbar.

3.7.10 Creating Multiple Copies of a Destination

If you want to create many similar Destinations you could repeat the steps explained in How To create a destination, many times over. However this would be a laborious process.

The JMS Grid Admin Tool provides you with a much quicker way to do this – it allows you to make multiple copies of an existing Destination.

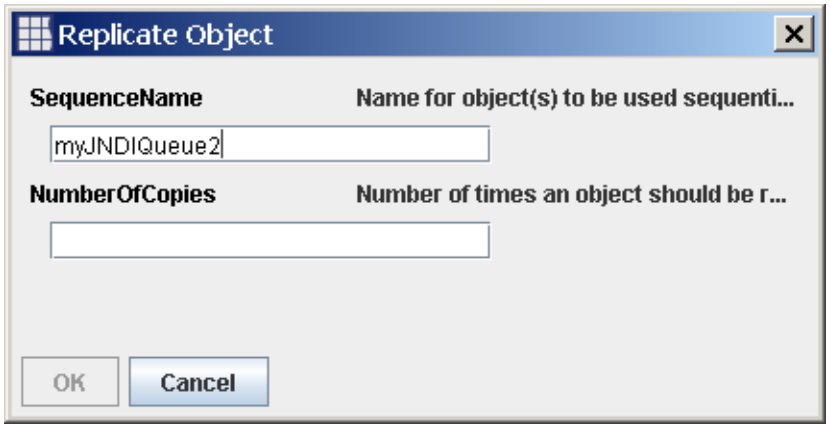
Prerequisite

You should be familiar with **Creating a JMS Destination** on page 133.

To create multiple copies of a destination

1. Create a single configuration for a Destination as explained in How To create a destination. You should ensure that the configuration parameters of this configuration closely match the configurations of your copies before deciding to replicate it.
2. In the Detail View, select the Destination you want to replicate. Click the right mouse button to bring up the **Item Menu**. From the **Item Menu** select **Replicate**. This will open the **Replicate Object** dialog, as shown below.

Figure 58 Replicate Object



3. Into the **SequenceName** field, enter the JNDI store name to be used for the replicated objects. The Admin Tool applies a simple numbering rule to the names of each of the replicas. A sequential number, starting from 0, is appended to the end of the SequenceName you give to create the replica's name. For example, if you entered myJNDIQueue as the sequence name, and asked for 3 copies to be made, then the replicated daemons would be called `myJNDIQueue0, myJNDIQueue1` and `myJNDIQueue2.`
4. Into the **NumberOfCopies** field enter the number of copies to make.
5. Click the **OK** button.
6. In the Detail View list the sequence-named replicas will appear.

Note: *If you now look at the properties of the replicated Destinations, you will see that both the JNDI Store name and the Destination Names have been modified as a sequence number has been appended to their original names. Thus, you do not need to modify the destination name of your replica destinations, so long as you are happy with the automatically generated destination names.*

3.7.11 Deleting a Destination

When messaging clients no longer use a redundant destination, it is possible to remove it from the JMS Admin Object Store

Prerequisite

You should be familiar with **Creating a JMS Destination** on page 133.

To delete a destination

1. In the Tree View, navigate to: `Console Root > JMS Grid > Administered Objects > Destination.`
2. In the Detail View, select the Destination instance to delete.

Note: *If you want to delete multiple Destinations at the same time, you can select multiple Destinations by holding down the Control key as you select each of the Destinations you want to delete.*

3. With the mouse still over that Destination, right button mouse click, and from the pop up Item Menu that appears, select **Delete**.
4. Click **Yes** in the confirmation dialog that appears.

Note: *Once a Destination has been deleted, new messaging clients that attempt to look it up will no longer be able to able to find it in the JMS Admin Object Store.*

3.8 Managing Client Applications

3.8.1 Running a Simple Client Application with JMS Grid

This section explains the simplest way to run a client application with the JMS Grid Message Server. Additional client configuration is covered in How To enable or disable JMS Grid client security.

In this section your messaging client is either:

1. Looking up a fully configured Connection Factory from the JMS Admin Object Store. The client will then use the properties specified in the Connection Factory to connect up to the JMS Grid Message Server.

   or

2. Programmatically defining its own Connection Factory

Note: *In previous releases of JMS Grid, a number of configuration parameters were passed to the message client application from the command line. Most of these are now set up as properties of a Connection Factory using the JMS Grid Admin Tool. One exception is the client's security provider which must still be specified from the client's command line, see How To enable or disable JMS Grid client security.*

Prerequisite

You should be familiar with **Starting a Daemon** on page 56.

To run a client application

1. Open the Windows Command Prompt from which you will execute the client application.
2. You must now set up the command shell's environment so that the client application can find the JMS Grid java code libraries that it requires. To do this, run the `setenv` script which can be found in the root of your JMS Grid installation. This sets the environment variable `JMSGRID_CPATH` to contain all the classpath entries needed to run a JMS Grid client.
3. To run your client application you must set the `-CLASSPATH` command line option to the %JMSGRID_CPATH% environment variable

```
java -classpath .;%JMSGRID_CPATH% myClient
```

This client application doesn't specify any client security options. This is explained in How To enable or disable JMS Grid client security.

Another Way to Specify CLASSPATH

Rather than specify the CLASSPATH on the java command line, you can set the `%CLASSPATH%` environment variable to `'.;%JMSGRID_CPATH%'`

See also

Creating a Connection Factory on page 125

3.8.2 Enabling a Client to Connect to a Daemon through a Firewall

If a client does not specify the port number to which it wants its socket to connect, then the Operating System will allocate it to a random port number.

One way that a firewall can be opened up to allow access through it is to specify a port number through which network traffic can travel.

JMS clients connect to the JMS Grid Message Daemon through Connections that are generated by Connection Factories. To enable a client to connect to a JMS Grid Message Daemon through a firewall, you must configure the client's Connection Factory to manufacture Connections that will create sockets on a specified port.

Prerequisite

You should be familiar with **Editing Connection Factory Properties for a Normal JMS Grid Client Connection** on page 131.

To enable clients to connect to Daemons through firewalls

1. Follow steps 1 and 2 of **"Editing Connection Factory Properties for a Normal JMS Grid Client Connection"** before performing the instructions below.
2. In the JMS Grid tab of the Connection Factory Properties dialog, specify the `localClientAddress` to be the network address and port number to which the client should attach its socket.

The `localClientAddress` should be in the following format:

```
networkAddress:portNumber
```

Example

```
myComputer: 54321
```

3.9 Advanced Administration

3.9.1 Specifying how Configuration Data is Stored

The JMS Grid Admin Tool is used for creating and manipulating configuration information that defines the behavior of a running JMS Grid Message Server installation.

The configuration data is stored in the form of the administrative objects that you navigate around in the Admin Tool Tree View. These administrative objects must be stored somewhere so that the run-time server can read the configuration values as shown in How the Admin Tool Works. This section explains how to specify where you want to store your installation's administrative objects.

The JMS Grid Admin Tool lets you store configuration data in two different stores: Admin Store and JMS Admin Object Store:

JMS Admin Object Store - Stores the JMS Administered Objects, namely Connection Factories and Destinations (Queues and Topics).

Admin Store - Stores all the other JMS Grid specific administered objects, such as configuration data about daemons, networks, clusters and security.

The JMS Grid Admin Tool leaves the storage options as open as possible. Available options are JNDI, LDAP, local or remote XML or basic file storage.

Note: *Be careful about switching from one Admin Store to another. The core security configurations, such as Users and Permission objects, will be copied from the old Admin Store and recreated in the new Admin Store. However, other admin objects such as daemon configurations and connection factories are not copied across. If your old Admin Store is persistent, then these configurations will still be stored in that old Admin Store and you can switch back to them. However, watch out if the old Admin Store isn't persistent – for example, WebLogic's JNDI, as this configuration data will be lost.*

Prerequisite

You should be familiar with **Using the JMS Grid Admin Tool** on page 46.

You should be familiar with **Deciding which Type of Configuration Data Store to use** on page 144.

To specify how configuration data is stored

1 Open the Settings dialog.

2 On the Menu Bar, click **Preferences** followed by **Admin Settings**.

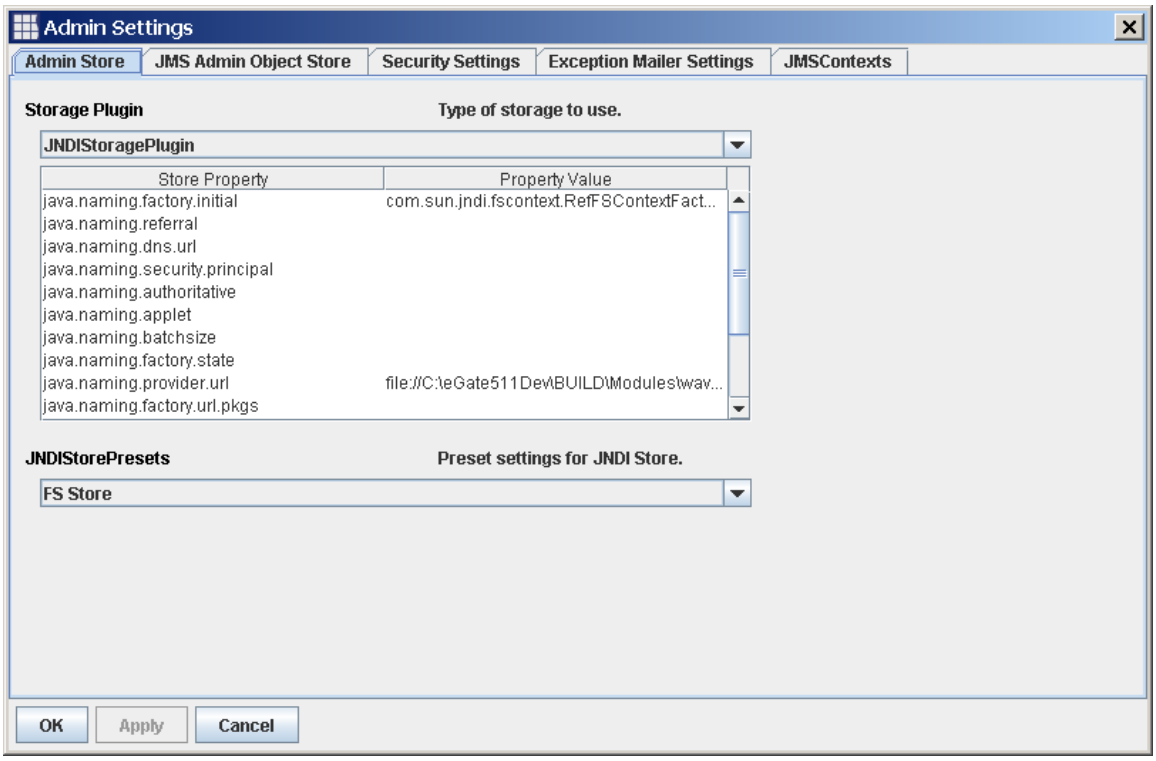
When you open the Admin Settings dialog, a warning will appear:

Figure 59 Warning Message

Click the **OK** button.

3 The Settings dialog will appear, as shown below.

Figure 60 Admin Settings



4 Select the store you want to set up – either the Admin Store or the JMS Admin Object Store. You select the store by clicking on the relevant tab at the top of the setting dialog.

5 Select which storage mechanism you want to use from the Storage Plug-in pull down. You are provided with the following options:

FileStoragePlugin - A file based mechanism. This is only useful for getting started.

JNDIStoragePlugin - An implementation of the Java Naming and Directory Interface.

XMLStoragePlugin - Administrative objects are stored on your local computer as XML

XMLRemoteStoragePlugin - Administrative objects are stored on a remote computer as XML.

LDAPStoragePlugin - An implementation of the Lightweight Directory Access Protocol.

6 Depending on which storage mechanism you chose, you will now have to set one or more properties. The most important property to set is the one that specifies the location of the store. Click the table icon below to see the mandatory properties that must be set for each storage type.

The table below shows the mandatory properties that must be set for each storage type.

Table 25 Storage Type Properties

Storage Type	Property	Description	Default value
File	`initialContext`	Folder into which the root node of the admin object hierarchy is stored.	<JMS Grid Installation folder>/file
JNDI	`java.naming.provider.url`	URL location of the root node of the admin object hierarchy.	file://localhost/<JMS Grid Installation folder>/jndi/wave
	`java.naming.factory.initial`	The class name of the initial context factory for the JNDI provider.	com.sun.jndi.fscontext.RefFSContextFactory
XML	`initialContext`	Folder into which the root node of the admin object hierarchy is stored.	<JMS Grid Installation folder>/xml
XML Remote	`inputURL`	URL location of the remote XML store.	file://<JMS Grid Installation folder>/xmlremote
	`outputPath`	The network location to where data is written.	<JMS Grid Installation folder>/xmlremote/
	`contextFile`	The file that represents the entire xml store. A single XML file contains all the subcontexts and objects.	Context.xml

Table 25 Storage Type Properties (Continued)

Storage Type	Property	Description	Default value
LDAP	`java.naming.provider.url`	URL location of the root node of the admin object hierarchy.	ldap://localhost:389
	`java.naming.factory.initial`	The class name of the initial context factory for the LDAP provider.	com.sun.jndi.ldap.LdapCtxFactory

Note: *Programmers who are writing JMS messaging clients will need to know the configuration of your JMS Admin Object Store so that they can direct their client code can look up Connection Factories and Destinations.*

7 To apply the alterations to the settings, click the **Apply** button. To apply the alterations and dismiss the Settings dialog, click the **OK** button. To discard any changes that have been entered, click the **Cancel** button.

3.9.2 Exporting Configuration Data to a File

There are times when you will want to export the configuration data that has been entered. The JMS Grid Admin Tool enables you to export configuration data to a Comma Separate Value file.

A Comma Separate Value (CSV) file is useful for storing table-like data structures. In a CSV file, each table row of the original table is output as a line of the file and, as its name suggests, a comma separates each value.

Which configuration data is exported depends on which item is currently selected in the Tree View. The data that is exported is always the table data that is shown in the Detail View when a given item is selected in the Tree View. For example, when the Single Daemons node is selected, a CSV file is created that contains a line for each of the single daemons that exist. For each single daemon, details are exported of its Name, Host, Protocol and Port, as seen in the Detail View when the Single Daemons node is selected.

Note: *The export function does not export all the configuration data for the admin objects, it only exports what is shown in the Detail View.*

The format of the CSV is as follows:

The first line contains the column header names. For example, when the Single Daemons node is selected in the Tree View, the first row will contain the text…

```
Name,Host,Protocol,Port
```

Each subsequent line contains the data for the corresponding row in the Detail View. The lines are ordered in the same order as the rows shown in the Detail View. For example, when the Single Daemons node is selected in the Tree View, the second line of

the CSV file will have the Detail View data for the single daemon that's displayed in the first row of the Detail View table – for example.

```
MyDaemonName,MyHost,tcp,2361
```

Details of exactly what will be exported for each selection in the Tree View is explained in the tables in **Reference** on page 146.

Note: *When exporting security admin objects such as Permissions, Users, Groups and Secure Destinations, then the data is not exported in the input format expected by the BatchUpdate utility.*

Prerequisite

The admin objects that you wish to export must have been created.

See also

Using the JMS Grid Admin Tool on page 46

Navigating the Tree View on page 49

Configuring a Single Daemon on page 54

Network and Cluster Concepts on page 78

Creating a New Cluster on page 80

Creating a Configuration for a Cluster Daemon on page 82

Creating a New Permission on page 103

Creating a Group on page 105

Creating a User on page 108

Creating a Secure Destination Object on page 112

Creating a Connection Factory on page 125

Creating a JMS Destination on page 133

To export configuration data to a file

1. In the tree view, navigate to the node whose subnode's data you want to export.
2. From the Admin Tool's Toolbar, click the Export icon.
3. In the Export View dialog that is opened, enter the name of the file you want the Detail View data to be exported to.
4. Click the **Save** button.
5. The Detail View data will be saved as a CSV format file.

See also

Reference on page 146.s

3.9.3 Deciding which Type of Configuration Data Store to use

The JMS Grid admin tool leaves the storage options for configuration data as open as possible. Available options are:

- File storage
- JNDI using FSContext
- JNDI using some other JNDI provider
- JNDI using JMS Grid JNDI provider
- LDAP
- local XML storage
- remote XML storage

You should choose the correct storage mechanism before you start creating your configuration data. If you later discover that you have selected a storage type that doesn't meet your requirements and that you will have to switch to another, then all your configuration data will have to be re-created using the new storage type.

This section discusses the various storage options in turn.

File storage

This is a simple file-based mechanism.

Advantages:

- Included with JMS Grid
- Simple to use and understand.

Disadvantages:

- Because it is file-based, where daemons will run on remote computers, those computers need file access to see the configuration. This may have security implications.
- There is a single point of failure if the hardware storing the configuration data fails.

3.9.4 JNDI Storage using FSContext

JNDI is the Java Naming and Directory Interface, the standard Java API for accessing configuration data. By default JMS Grid uses the FSContext JNDI provider from Sun Microsystems. This stores data as directories and files in the file system.

Advantages:

- Included with JMS Grid
- Simple to use and understand.

Disadvantages:

- Because it is file-based, where daemons will run on remote computers, those computers need file access to see the configuration. This may have security implications.
- There is a single point of failure if the hardware storing the configuration data fails.

3.9.5 JNDI Storage using some other JNDI Provider

You can use any other JNDI provider.

Advantages

- Wide choice of JNDI providers
- Can use a JNDI provider that is already in use within your organization
- Can use the JNDI provider that is provided with your application server
- Can use a JNDI provider that is fault-tolerant

Disadvantages

- Not included with JMS Grid

3.9.6 JNDI Storage using JMS Grid JNDI Provider

JMS Grid contains its own JNDI provider in which a single JMS Grid daemon works as the JNDI provider server.

Advantages:

- Included with JMS Grid
- URL-based; no need for remote file access

Disadvantages

- There is a single point of failure if the single JMS Grid daemon fails

3.9.7 LDAP Storage

LDAP is the Lightweight Directory Access Protocol, a simplified version of the X500 directory access protocol. LDAP is a broader standard than JNDI as it is not limited to Java clients. Many organizations use a corporate LDAP provider as a corporate standard. See **LDAP Provider Support** on page 23.

Advantages

- Ability to conform to corporate standards for configuration data storage
- URL-based; no need for remote file access
- Can use a LDAP provider that is fault-tolerant

Disadvantages

- Not included with JMS Grid

- Performance can sometimes be an issue
- Complex to configure and administer

3.9.8 XML Storage

JMS Grid allows administrative objects to be stored on the local computer as an XML document.

Advantages

- Included with JMS Grid
- The XML document is formatted in a way that is easy for people to read, which can be convenient for debugging

Disadvantages:

- Because it is file-based, where daemons will run on remote computers, those computers need file access to see the configuration. This may have security implications.
- There is a single point of failure if the hardware storing the configuration data fails.

3.9.9 Remote XML Storage

JMS Grid allows access to a previously-created XML configuration document via a URL.

Advantages

- Included with JMS Grid
- The XML document is formatted in a way that is easy for people to read, which can be convenient for debugging
- Allows a remote XML document to be accessed via a protocol such as HTTP instead of via the file system

Disadvantages:

- Access to the XML document is read-only

3.10 Reference

3.10.1 Detail View Tables

This section describes each column of each of the Detail View Tables.

```
Console Root > JMS Grid > Single Daemons
```

Table 26 Single Daemons

Name	Description
Name	Shows the name of the daemon.
Host	The host on which the daemon resides.
Protocol	The primary protocol supported. For a list of the protocols and ports supported view the general properties of the daemon.
Port	The port to which clients and other daemon use for communicating with the daemon.

```
Console Root > JMS Grid > Networks > <aNetwork> > <aCluster>
```

Table 27 Networks <aNetwork> <aCluster>

Name	Description
Name	Shows the name of the daemon.
Host	The host on which the daemon resides.
Protocol	The primary protocol supported. For a list of the protocols and ports supported view the general properties of the daemon.
Port	The port to which clients and other daemon use for communicating with the daemon.

```
Console Root > JMS Grid > Security > Access > Permissions
```

Table 28 Permissions

Name	Description
Name	A unique Permission name.
Resource Name	A name of a resource associated with this permission.
Resource Type	Type of destination, for example, a Queue.
Access Mode	Access mode granted by this permission, for example, read.

```
Console Root > JMS Grid > Security > Access > Users
```

Table 29 Users

Name	Description
Name	A unique User name.

Table 29 Users (Continued)

Name	Description
Group	A name of the Groups that this User is associated with.
Permission	The permission rights assigned to this User. This does not show the inherited permission rights.
Account Enabled	Whether or not the User account is enabled.

`Console Root > JMS Grid > Security > Access > Groups`

Table 30 Groups

Groups	Description
Name	
Name	A unique Group name.
Parent	The name of any Groups that this group is a member of. The Group inherits all Permissions of its parent Groups.
Permissions	Permissions assigned to this group.

`Console Root > JMS Grid > Security > Destinations Security`

Table 31 Destinations Security

Name	Description
Name	JMS Destination name to which this Destination Security object is associated.
Encrypted	Whether or not the JMS Destination is encrypted i.e. secure.

`Console Root > JMS Grid > Administered Objects > Connection Factory`

Table 32 Connection Factory

Name	Description
Destination Type	The type of Destinations the Connection Factory will manufacture e.g. XAQueue
Store Name	A unique JNDI name by which the factory can be looked up in a JNDI store.
Provider	The provider (or driver name) used to create Destinations, e.g. JMS Grid.
Status	Indicates whether the Connection Factory has installed successfully. If not, the entire row will be rendered red and the reason for the failure will be displayed in this field.

`Console Root > JMS Grid > Administered Objects > Destinations`

Table 33 Destination

Name	Description
Destination Type	The type of Destination the Connection Factory will provide e.g. Queue
Store Name	A unique JNDI name by which the destination can be looked up in the JNDI store.
Destination Name	The JMS name for the destination.

Chapter 4

JMS Programming

This chapter is intended for Java programmers building a JMS client application that uses JMS Grid. It describes the Java code needed to connect to JMS Grid and send and receive messages.

Because JMS Grid is essentially an implementation of the JMS standard, much of the code you need to write is entirely standard; therefore, you should supplement the information contained in this chapter with one of the various JMS client programming textbooks that are available. A list of suggested books and online resources is given on page 105.

It also gives some information on how to administer JMS Grid using Java code, although if you use the JMS Grid administration tool Java programming is not generally required. See the JMS Grid Administration chapter for more information.

Sections Contained in this Chapter

- **Overview of JMS** – an introduction to the basic concepts of the Java Message Service API
- **Building a JMS Application** – describes the basics of JMS and how to write a JMS-compliant client application
- **Additional Programming Features** – describes a number of additional features of JMS Grid which go beyond those defined in the JMS standard
- **Programming Examples** – describes the example client programs provided with JMS Grid. You should run these examples and examine the code used to produce them. You can also use these examples as the basis for your own client programs, especially when learning to use JMS Grid.

4.1 Overview of JMS

The Java Message Service (JMS) API is an API for accessing enterprise messaging systems from Java programs. It is defined in a specification from Sun Microsystems (see References on page 105) and has been implemented by a number of independent software vendors. It has quickly become the industry standard API for enterprise messaging.

This section introduces the basic concepts on which the JMS API is based. This is only a very brief summary of JMS. For full details the specification itself should be consulted. The following concepts are covered:

- Message Types
- Messaging Models
- Synchronous and Asynchronous Consumers
- Persistent Messages
- Message Acknowledgement & Redelivery
- Message Expiry

4.1.1 Message Types

JMS defines five different message types, depending on the payload carried:

- Object - for transporting any serializable Java object.
- Bytes - for transporting a stream of uninterpreted bytes.
- Stream - for transporting a self-defining stream of Java primitives.
- Text - for transporting String objects, including XML documents.
- Map - for transporting a self-defining set of name-value pairs where names are Strings and values are Java primitive types.

4.1.2 Messaging Models

The basic purpose of JMS is to allow a client to create a message and dispatch it to the JMS provider, which will then deliver it to one or more clients. JMS defines two different ways in which this may be done. These are:

- The point-to-point model
- The publish-and-subscribe model

These two models (the term "paradigm" is sometimes used) correspond to the two leading messaging models provided by existing messaging products.

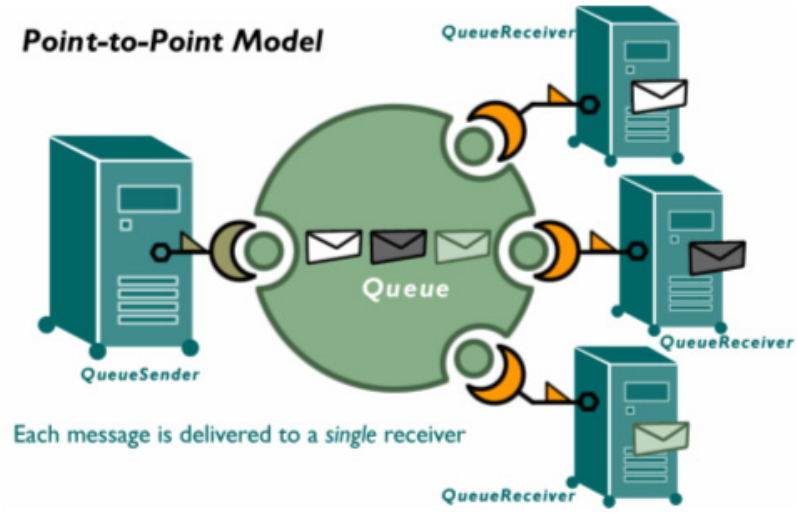
Point To Point Messaging

With point-to-point (P2P) messaging, JMS clients send messages to and receive messages from message queues.

Multiple clients may send messages to a particular queue, and multiple clients may monitor a particular queue for messages. However each message will be only be delivered to a single client. This is similar to a letter sent to a particular address; there might be multiple potential recipients at the given address, but the letter will be opened by only one of them. Point-to-point messaging is ultimately a one-to-one process.

When a message is sent to a queue, if there are no receivers waiting for messages from a queue, the JMS provider will retain the messages until a receiver appears for it to send the messages to.

Figure 61 Point to Point Model



Messages are by default persistent. When a persistent message is sent to a queue, and there are no receivers waiting for messages from a queue, the JMS provider will retain the messages until a receiver appears for it to send the messages to. Messages can also be defined as being non-persistent. When a non-persistent message is sent to a queue, and there are no receivers waiting for messages from a queue, the message may be discarded.

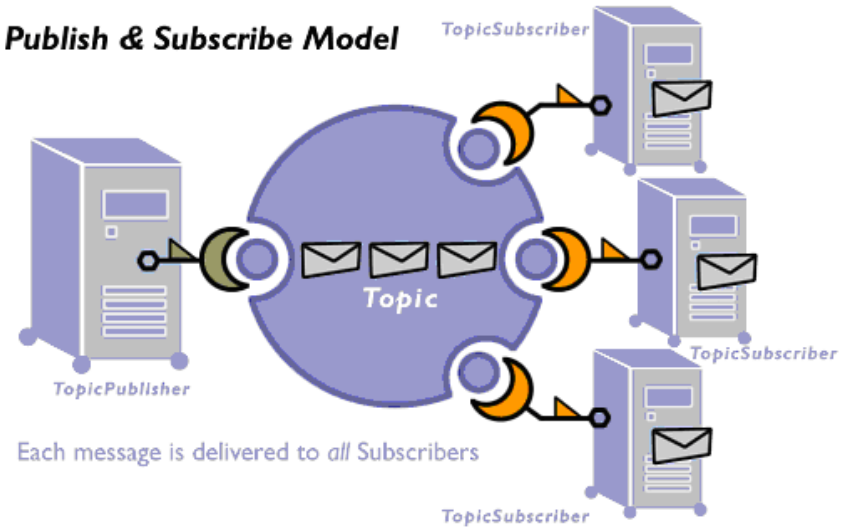
Publish and Subscribe

With publish-and-subscribe (Pub-Sub) messaging, JMS clients publish messages to a topic, and other JMS clients subscribe to messages from that topic.

Multiple clients may publish messages to a particular topic, and multiple clients may subscribe to a particular topic for messages. When a message is published to a topic, the JMS provider will deliver it to all the clients who are subscribing to that topic. Publishing a message to a topic is therefore a bit like 'broadcasting' it to all its subscribers.

Normally a subscriber only receives those messages that are published while it is active, irrespective of whether those messages are persistent or non-persistent. However a client may create a durable subscription to a topic and then terminate. The JMS provider will then store all persistent messages to the topic. When the client restarts and re-establishes the durable subscription, the stored messages will be delivered to the client.

Figure 62 Publish and Subscribe Model



Generic Terms

As may be apparent from the previous two sections, the JMS specification defines distinct terminology for the two messaging models. P2P is all about sending messages to, and receiving messages from, queues. Pub-Sub is all about publishing messages to, and subscribing to messages from, topics.

The term destination is used as a generic term for a queue or topic.

The term message producer is used as a generic term for a queue sender or topic publisher.

The term message consumer is used as a generic term for a queue receiver or a topic subscriber.

4.1.3 Synchronous and Asynchronous Consumers

A JMS client may consume messages either synchronously or asynchronously.

A client receives a message synchronously by calling one of the receive() methods. This will block until a message is received or until a timeout expires.

A client may receive messages asynchronously by registering an object that implements the JMS MessageListener interface. When a message arrives the listener's onMessage() method is called.

4.1.4 Persistent Messages

Messages sent to a queue or topic may be either persistent or non-persistent. Messages are persistent by default. Configuring a message to be persistent has three consequences:

"With point-to point messaging, whether the message is persistent or not affects the behavior of the JMS provider when there are no receivers on a queue. When a persistent message is sent to a queue, and there are no receivers waiting for messages from a queue, the JMS provider will retain the messages until a receiver appears for it to send the messages to. If the message is not persistent, however, the message may be discarded.

With publish-and-subscribe messaging, whether the message is persistent or not affects how the JMS provider handles durable subscriptions which have no active subscriber. If a client has created a durable subscription to a topic and then terminated, the JMS provider will then store all persistent messages published to that topic. However any non-persistent messages will not be stored. When the client restarts and re-establishes the durable subscription, the stored messages will be delivered to the client.

"Making a message persistent also offers protection in case the JMS provider crashes. Once a persistent message has been sent, the JMS provider will undertake not to lose that message, even if the provider fails and has to be restarted. If a message is non-persistent, and the JMS provider fails, then the message may be lost.

4.1.5 Message Acknowledgement and Redelivery

When a JMS provider delivers a message, it retains the message until the consumer has acknowledged it. This means that if the client fails before it has acknowledged the message then the provider can deliver it a second time.

In order to benefit from this, a message consumer normally does not acknowledge a message until it has finished processing it.

There are four ways in which a consumer may acknowledge receipt of a message.

1. Messages may be received in a transaction. If the transaction is committed all the messages consumed during that transaction are acknowledged. If the transaction is rolled back then the messages are not acknowledged.
2. Messages may be explicitly acknowledged by the consumer. Acknowledging a message automatically acknowledges all previous messages.
3. Messages may be automatically acknowledged. With this option, the client automatically acknowledges receipt of a message when it has either successfully returned from a call to receive() or the MessageListener it has called to process the message successfully returns.
4. Messages may be automatically acknowledged, but lazily. The client automatically acknowledges receipt of a message but the acknowledgement may not be sent until a later time. This means that if the client fails after it has processed a message but before the acknowledgement has been sent then the provider may deliver it a second time even though it has already been processed.

4.1.6 Message Expiry

When a message is sent to a queue or published to a topic, the client can specify a 'time to live' value.

The JMS provider will not deliver a message whose time to live has expired. Messages stored in a queue or a durable subscription will be deleted when their time to live expires.

4.2 Building A JMS Application

This section describes the basics of JMS and how to write a JMS compliant client application. It will cover the following:

- The Basic Structure of a JMS application
- Obtaining a JMS Connection
- Obtaining a JMS Session
- Obtaining a JMS Destination
- User Security page
- JMS Messages page
- Publish & Subscribe Messaging Using Topics
- Point to Point Messaging Using Queues
- Local Transactions
- Global Transactions
- Message Selectors
- Closing Down

4.2.1 The Basic Structure Of A JMS Application

All JMS applications begin with the same basic steps:

1. Create a connection to the JMS provider and obtain a Connection object.

   This establishes a communications channel to the JMS provider and authenticates the client.

2. Obtain a Session object for this connection.

   A session is a factory for creating message producer and consumer objects that will all operate within the same thread. This means that if a client desires to have one thread producing messages while others consume them, the client should use a separate Session for its producing thread

3. Obtain a Queue or Topic object corresponding to the particular message destination you wish to use.

   Point-to-point messaging uses queues, whereas publish-and-subscribe messaging uses topics.

   After this, a message producing application will then:

4 Create a message producer object corresponding to the session and the particular queue or topic to which it wishes to produce messages.

To send messages to a queue a QueueSender object is needed. See Creating a QueueSender.

To publish messages to a topic a TopicPublisher object is needed. See Creating a TopicPublisher.

- Create a Message object with the required payload.
- Use the message producer to send or publish the message.

A message consuming application will:

- Create a message consumer object corresponding to the session and the particular queue or topic from which it wishes to consumer messages:

To receive messages from a queue a QueueReceiver is needed. See Creating a QueueReceiver.

To subscribe to messages on a topic a TopicSubscriber is needed. See Creating a TopicSubscriber.

- Call the start() method on the Connection object to start delivery of incoming messages.
- Wait for messages to arrive, either synchronously by calling the receive() method on the message consumer or asynchronously by defining a MessageListener object whose onMessage() method is invoked when a message is received.

When an application is finished, it should:

1 Close any message producer or message consumer objects

2 Close the session

3 Close the connection

4.2.2 Obtaining A JMS Connection

JMS Grid offers two alternative approaches to obtaining a JMS connection:

1 Create a JMS connection factory and use it to create the JMS connection

2 Use JNDI to obtain the JMS connection factory and then use it to obtain the JMS connection. The connection factory must have been previously created and stored in JNDI, probably by an administrator.

The JMS specification recommends that the second approach (JNDI) be used, because it allows the configuration of the connection factories - which is vendor-specific and not covered by JMS - to be kept separate from the client code.

Client programs may, however, create the connection factories themselves and use them directly. This avoids the need to perform a JNDI lookup. However it means that configuration information needs to be hard-wired into the client code, making it more complex to administer and less portable between JMS providers.

Creating a Connection Factory and using it to Create a Connection

The simplest way to obtain a JMS connection is to create a connection factory and use it to create the connection. JMS Grid provides four connection factory classes, one for each type of connection defined in the JMS standard.

Table 34 Connection Factory Classes

Connection Factory Classes	
Connection Factory Class	Used To Create Instances Of
com.spirit.wave.jms.WaveQueueConnectionFac tory	javax.jms.QueueConnection
com.spirit.wave.jms.WaveTopicConnectionFac tory	javax.jms.TopicConnection
com.spirit.wave.jms.WaveXAQueueConnectionFactory	javax.jms.XAQueueConnection
com.spirit.wave.jms.WaveXATopicConnectionFactory	javax.jms.XATopicConnection

The type of connection factory you need depends on the type of JMS destination (queue or topic) you will be using and whether you will be using a global transaction.

This example shows how to obtain a JMS queue connection by first creating a QueueConnectionFactory, and then using it to create a QueueConnection:

```
// assumes you have imported java.util.*, javax.jms.*
// and com.spirit.wave.jms.WaveQueueConnectionFactory;
Properties factProps = new Properties();
factProps.setProperty("driverName","JMSGrid");
factProps.setProperty("clientID","abc123")
WaveQueueConnectionFactory qcf =
    new WaveQueueConnectionFactory(factProps);
QueueConnection queueConnection =
qcf.createQueueConnection();
```

The property driverName specifies the JMS Grid driver to be used. In this example, this property is set to "JMSGrid", which is the default and which specifies that an ordinary JMS Grid connection should be created which connects to a remote daemon. The only other valid value is "JMSGridEmbedded", which specifies that the connection should use an embedded daemon that is running within the same JVM.

You may also set the property clientID. This allows you to specify the JMS client identifier that will be used with all connections created from this connection factory.

Other driver-specific properties may need to be specified.

If the application used Topics instead of Queues it would need to create a TopicConnectionFactory and then use that to create a TopicConnection:

```
// assumes you have imported java.util.*, javax.jms.* and
// com.spirit.wave.WaveTopicConnectionFactory;
TopicConnection topicConnection =
qcf.topicQueueConnection();
Properties factProps = new Properties();
factProps.setProperty("driverName","JMSGrid");
factProps.setProperty("clientID","abc123")
WaveTopicConnectionFactory tcf =
    new WaveTopicConnectionFactory(factProps);
```

```
TopicConnection topicConnection =
tcf.createTopicConnection();
```

It is also possible to set properties on an existing ConnectionFactory as shown below.

```
WaveQueueConnectionFactory qcf =
    new WaveQueueConnectionFactory(factProps);
Properties factProps = new Properties();
factProps.setProperty("driverName","JMSGrid");
qcf.setProperties(factProps);
QueueConnection queueConnection =
qcf.createQueueConnection();
```

The new set of properties will replace any existing properties defined on the ConnectionFactory.

Using JNDI for Obtaining a Connection Factory

If a connection factory has already been created and stored in a JNDI namespace, the client program can simply lookup the connection factory by name and then use it to create the JMS connection.

In the following example, a JNDI lookup is performed to obtain a QueueConnectionFactory, which is then used to create a QueueConnection:

```
// assumes you have imported java.util.*, javax.naming.* and
javax.jms.*;
Properties factProps = new Properties();
factProps.put(
Context.INITIAL_CONTEXT_FACTORY,
    "com.spirit.directory.SpiritDirectoryContextfactory");
Context ctx = new InitialContext(factProps);
QueueConnectionFactory qcf;
Qcf = (QueueConnectionFactory)ctx.lookup("QueueConnectionFactory");
QueueConnection queueConnection = qcf.createQueueConnection();
```

If the application used Topics instead of Queues it would need to perform a JNDI lookup to obtain a TopicConnectionFactory and then use it to create a TopicConnection:

```
// assumes you have imported java.util.*, javax.naming.* and
javax.jms.*;
Properties jndiProps = new Properties();
jndiProps.put(
Context.INITIAL_CONTEXT_FACTORY,
    "com.spirit.directory.SpiritDirectoryContextfactory");
Context ctx = new InitialContext(jndiProps);
TopicConnectionFactory tcf;
Tcf = (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
TopicConnection topicConnection = tcf.createTopicConnection();
```

This is covered in more detail in Obtaining an Initial JNDI Context on page 22.

These examples use the JMS Grid Directory Service, which requires a JMS Grid Message Server daemon to be running. You can, in fact, use any JNDI provider. For details on how to specify another JNDI provider.

See Other JNDI providers.

Binding a Connection Factory to the JNDI Namespace

Before you can use JNDI to obtain a connection factory you will normally need to create a suitable connection factory instance and bind it to the JNDI namespace using a name of your choice.

This is normally performed using the JMS Grid administration tool. For details, see the JMS Grid Administration section.

Alternatively you can configure the connection factories using Java code. The remainder of this section describes how to do this. There are three steps:

1. Obtain an initial JNDI context. An example of how to do this was given in Obtaining an Initial JNDI Context on page 22. More details are given later in this chapter.
2. Create a connection factory. This was described in Creating a Connection Factory and using it to create a Connection on page 18.
3. Use the bind() method to bind the connection factory to the JNDI context using whatever name you wish to use.

The following example shows how you might create a queue connection factory and bind it to the JNDI namespace.

```
// assumes you have imported java.util.*, javax.naming.* and
javax.jms.*;

// obtain the initial JNDI context
Properties jndiProps = new Properties();
jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
              "com.spirit.directory.SpiritDirectoryContextFactory");
Context ctx = new InitialContext(jndiProps);

// create the connection factory
Properties cfProps = new Properties();
cfProps.setProperty("driverNames","JMSGrid");
WaveQueueConnectionFactory factory = new
                                WaveQueueConnectionFactory(cfProps);

// bind the connection factory to the JNDI context
String connectionFactoryName = "MyQueueConnectionFactory";
ctx.bind(connectionFactoryName, factory);
```

If you are using either the JMS Grid Directory Service or the JMS Grid VM Directory Service a number of connection factories are automatically configured. See Predefined Connection Factories.

Obtaining an Initial JNDI Context

The examples given above show how you can use a JNDI InitialContext to bind a connection factory to a JNDI namespace or to lookup an existing connection factory. Now let us look in more detail at how the initial JNDI context was obtained in those examples.

The first step was to create a new Properties object:

```
Properties jndiProps = new Properties();
```

You then set the property Context.INITIAL_CONTEXT_FACTORY (which evaluates to the string java.naming.factory.initial) to be the name of the factory class that will create the InitialContext for us:

```
jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,"com.spirit.directory.SpiritDirectoryContextFactory");
```

You then created the InitialContext, supplying the properties object as an argument:

```
Context ctx = new InitialContext(jndiProps);
```

JMS Grid Directory Service

When creating an `InitialContext`, by specifying the property `Context.INITIAL_CONTEXT_FACTORY` (which evaluates to the string `java.naming.factory.initial`) you are specifying the JNDI provider you wish to use.

In the above examples this property is set to `com.spirit.directory.SpiritDirectoryContextFactory`. This specifies that the JMS Grid Directory Service should be used. This is a distributed JNDI provider that stores JNDI bindings in whatever persistent store JMS Grid Message Server is configured to use. If you use this provider the JMS Grid Message Server daemon must be running. A number of other properties may be supplied as well. These are listed in the table below:

Table 35 JMS Grid Directory Service - Property Settings

Property Name	Property Value
`Context.INITIAL_CONTEXT_FACTORY (java.naming.factory.initial)`	com.spirit.directory.SpiritDirectoryContextFactory Must be supplied
`Context.PROVIDER_URL (java.naming.provider.url)`	A string of the form protocol://hostname:port specifying how to connect to the JMS Grid Message Server daemon Optional: the default is stream://localhost:50607
`Context.SECURITY_PRINCIPAL (java.naming.security.principal)`	A string specifying the identity of the principal (i.e. the username). This is used to authenticate this client with the JMS Grid Message Server daemon. Optional: the default is anonymous
`Context.SECURITY_CREDENTIALS (java.naming.security.credentials)`	A string specifying the credentials of the principal (i.e. the password). This is used to authenticate this client with the JMS Grid Message Server daemon. Optional: the default is anonymous.

JMS Grid VM Directory Service

The JMS Grid VM Directory Service is an in-memory JNDI provider. To use it, use the following property setting:

Table 36 JMS Grid VM Directory Service

Property Name	Property Value
`Context.INITIAL_CONTEXT_FACTORY (java.naming.factory.initial)`	com.spirit.directory.Spirit VMDirectoryContextFactory Must be supplied

The JMS Grid VM Directory Service is a non-distributed, non-persistent provider that simply stores object bindings in memory. It does not require a JMS Grid Message Server daemon to be running. Because the bindings are not persistent, the JMS Grid VM Directory Service is only suitable for applications that do not require central configuration.

Other JNDI Providers

You may, if you wish, use any third-party JNDI provider. To do so, you need to set the properties `Context.INITIAL_CONTEXT_FACTORY` and `Context.PROVIDER_URL` to appropriate values. You may also need to supply additional properties. For full details, consult the documentation for the JNDI provider you wish to use.

Here is an example of a client that uses the File System JNDI provider from Sun Microsystems, which is redistributed with JMS Grid in the file `fscontext.jar`:

```
// assumes you have imported java.util.*, javax.naming.* and
javax.jms.*;

Properties jndiProps = new Properties();
jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.fscontext
.RefFSContextFactory");
jndiProps.put(Context.PROVIDER_URL,"file://%JMS Grid%");
Context ctx = new InitialContext(jndiProps);
QueueConnectionFactory qcf;
qcf = (QueueConnectionFactory)ctx.lookup("QueueConnectionFactory");
QueueConnection queueConnection = qcf.createQueueConnection();
```

Some JNDI providers require a JNDI server daemon to be running. Any such daemon must already be running before you can use it to create an initial context.

Specifying the JNDI Provider Using the File Jndi.properties

In each example above, the JNDI provider was chosen by creating a Properties object, setting some properties, and supplying this object as an argument to the constructor for `InitialContext`.

Alternatively, the no-argument constructor may be used. That is, the `InitialContext` is created without specifying a properties object:

```
Context ctx = new InitialContext();
```

In this case, the constructor will look for a file jndi.properties in the classpath and obtain the properties java.naming.factory.initial and java.naming.provider.url from this. It is recommended that you store this file in the properties directory of your JMS Grid installation.

Here is an example of a jndi.properties file that specifies the use of the File System JNDI provider:

```
java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContext
Factory
java.naming.provider.url=file://%JMS Grid%
```

Specifying the JNDI provider in the file jndi.properties rather than in the client code means you can change to a different JNDI provider without having to change your client code. It is therefore highly recommended.

Predefined Connection Factories

The JMS Grid Directory Service and the JMS Grid VM Directory Service have the following connection factories automatically bound.

Table 37 Predefined Connection Factories

JNDI Name	Type of Connection Factory
QueueConnectionFactory	WaveQueueConnectionFactory
TopicConnectionFactory	WaveTopicConnectionFactory
XAQueueConnectionFactory	XAWaveQueueConnectionFactory
XATopicConnectionFactory	XAWaveTopicConnectionFactory

For each of these connection factories, the property driverName is set to JMSGrid. No other properties are set.

This means that if you use one of these providers you can very quickly create client applications that use JMS Grid Message Server without the need to initialize the JNDI namespace to contain any connection factories.

The JMS Grid Directory Service and the JMS Grid VM Directory Service will also generate Queues and Topics automatically. For details, See Automatic Queue and Topic Generation.

4.2.3 Obtaining A JMS Session

Once a JMS connection has been obtained by the application, the next step is to create a JMS session.

If you wish to use a point-to-point queue you need to create a QueueSession. If you wish to use a publish-and-subscribe topic you need to create a TopicSession.

When creating the session, the application needs to specify whether the session should be transacted and what type of message acknowledgement should be used.

The following code sample shows how to create a QueueSession:

```
boolean isTransacted = false;
int acknowledgeMode = Session.AUTO_ACKNOWLEDGE;
QueueSession queueSession
=queueConnection.createQueueSession(isTransacted,acknowledgeMode);
```

The following code sample shows how to create a `TopicSession`:

```
boolean isTransacted = false;
int acknowledgeMode = Session.AUTO_ACKNOWLEDGE;
TopicSession topicSession
=topicConnection.createTopicSession(isTransacted,acknowledgeMode);
The argument isTransacted specifies whether the session should be
transacted or not.
```

Non-transacted Sessions

If `isTransacted` is set to false then the session is non-transacted and the argument `acknowledgeMode` is used to specify how messages will be acknowledged. The possible values are listed in the following table.

Table 38 Acknowledgment Modes

Acknowledgement mode	Meaning
`Session.AUTO_ACKNOWLEDGE`	The session will automatically acknowledge a client's receipt of a message.
`Session.CLIENT_ACKNOWLDEGE`	The client will acknowledge a message by calling its acknowledge() method. Acknowledging a message acknowledges all messages that have so far been consumed in this session, up to and including the message being acknowledged.
`Session.DUPS_OK_ACKNOWLEDGE`	The session will lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails. This mode should therefore only be used by consumers that are tolerant of duplicate messages.

A session will retain a message until it has been acknowledged. If the acknowledgement mode is `Session.AUTO_ACKNOWLEDGE` then the session will retain the message until all consumers have received it.

If the acknowledgement mode is `Session.CLIENT_ACKNOWLDEGE` then a message must be explicitly acknowledged using its `acknowledge()` method.

A third acknowledgement mode is `Session.DUPS_OK_ACKNOWLEDGE`. This mode is intended to simplify the work performed by the session in filtering out duplicates, and should only be used by a client that can tolerate duplicate messages.

Transacted Sessions (Local Transactions)

If isTransacted is set to true then the session is transacted and the argument acknowledgeMode is ignored.

When a transacted session is used to produce messages, the messages will only be really sent or published when the session is committed using the method `session.commit()`. If the session is rolled back instead using the method

`session.rollback()` then all messages sent or published within the transaction will not be delivered and will instead be discarded.

When a transacted session is used to consume messages, then the client will only acknowledge that it has consumed any messages when the session is committed. If the session is rolled back instead then the messages will not be acknowledged.

A transacted session only involves messages produced and consumed within the session and nothing else. It doesn't involve messages produced or consumed in other JMS sessions, not does it involve other resources such as databases. In the terminology of transactions, it uses a local transaction. For more information See Local Transactions.

There is another type of transaction, called a global transaction. This is described in Global Transactions.

4.2.4 Obtaining a JMS Destination

A JMS destination is the generic term for both point-to-point queues and publish-and-subscribe topics. `Destination` is the common supertype of Queue and Topic.

JMS Grid offers three alternative ways of obtaining a JMS destination:

1. Obtaining a destination from the session using the `createQueue` and `createTopic` methods.
2. Creating a Queue or Topic object explicitly and using it directly.
3. Using JNDI to obtain the Queue or Topic object and then using it in your application. The Queue or Topic object must have been previously created and stored in JNDI, probably by an administrator.

The `createQueue` and `createTopic` methods provide the simplest way of obtaining a Queue or Topic object. However, although this is consistent with the JMS standard, such a simple API does not allow provider-specific parameters to be specified, and so you will get the default values of any such parameters.

The JMS specification recommends that the third approach (JNDI) be used, because it allows the configuration of the queues and topics - which is vendor-specific and not covered by JMS - to be kept separate from the client code.

Client programs may, however, create the queue and topic objects themselves and use them directly. This avoids the need to perform a JNDI lookup. However it may mean that configuration information needs to be hard - wired into the client code, making it more complex to administer and less portable between JMS providers.

Obtaining a Destination from the Session

The simplest way to obtain a JMS destination is to retrieve it from the Session object. To obtain a Queue, use the method `createQueue()` of the QueueSession:

Queue queue = `queueSession.createQueue` (destName);

To obtain a Topic, use the method `createTopic()` of the TopicSession:

Topic topic = `topicSession.createTopic(queueName);`

In both cases, `destName` is the name of the queue or topic that you wish to use.

Creating the Destination Explicitly

A Queue object may also be obtained by creating an instance of the class DefaultQueue. This is a JMS Grid class that implements the Queue interface:

```
// assumes you have imported com.spirit.wave.message.DefaultQueue;

DefaultQueue queue = new DefaultQueue(destName);
```

Similarly, a Topic object may be obtained by creating an instance of the class DefaultTopic. This is a JMS Grid class that implements the Topic interface:

```
// assumes you have imported com.spirit.wave.message.DefaultTopic;

DefaultTopic topic = new DefaultTopic(destName);
```

In both cases, destName is the name of the queue or topic that you wish to use.

Obtaining a Destination Using JNDI

If a Queue or Topic has already been created and stored in a JNDI namespace, the client program can simply lookup the Queue or Topic by its JNDI name.

```
Queue queue = (Queue)ctx.lookup(destJNDIName);
```

or

```
Topic topic = (Topic)ctx.lookup(destJNDIName);
```

In this example, ctx is a JNDI Context; See Obtaining an Initial JNDI Context for a discussion of how to obtain this.

`destJNDIName` is the JNDI name under which the Queue or Topic is bound to the JNDI namespace. This need not be the same as the name of the queue or topic. You can think of the JNDI name as a 'logical' name of the queue or topic. The 'actual' name of the queue or topic can be configured separately by a central administrator and need not be known by the client application.

Configuring a Destination

The JMS Grid classes DefaultTopic and DefaultQueue are configurable objects. It is possible to get and set properties on them:

```
void DefaultTopic.setProperties(Properties props);
Properties DefaultTopic.getProperties();

void DefaultQueue.setProperties(Properties props);
Properties DefaultQueue.getProperties();
```

This allows destination-specific properties to be set. To set a property on a Queue or Topic, it is necessary to first cast it to a `DefaultQueue` or a `DefaultTopic` as appropriate.

Note: *The setting properties on a Queue or Topic will replace any existing properties defined for that destination.*

BInding a Destination to the JNDI Namespace

Before you can use JNDI to obtain a queue or topic you will normally need to create a suitable instance of Queue or Topic and bind it to the JNDI namespace using a suitable JNDI name.

This is normally performed using the JMS Grid administration tool. See the JMS Grid Administration chapter for details.

Alternatively you can configure these objects using Java code. The remainder of this chapter describes how to do this. There are three steps:

1. Create an instance of DefaultQueue or DefaultTopic. This is described in Creating the Destination Explicitly.
2. Obtain an initial JNDI context. This is described in Obtaining an Initial JNDI Context.
3. Use the `bind()` method to bind the Queue or Topic to the JNDI context using whatever JNDI name you wish to use.

The following example shows how you might use Java code to create a Queue and bind it to the JNDI namespace:

```
DefaultQueue queue = new DefaultQueue(destName);
ctx.bind(destJNDIName,queue);
```

The following example shows how you might use Java code to create a Topic and bind it to the JNDI namespace:

```
DefaultTopic topic = new DefaultTopic(destName);
ctx.bind(destJNDIName,topic);
```

Note the difference between destName and destJNDIName in the above examples:

- `destName` is the name of the queue or topic.
- `destJNDIName` is the JNDI name under which the Queue or Topic is bound to the JNDI namespace

These two names need not be the same, and indeed it is generally considered desirable for them to be different, as this allows the destination name to be changed without the need to modify the name used in the client code. You can think of the JNDI name as a 'logical' name of the queue or topic. The 'actual' name of the queue or topic can be configured separately by a central administrator and need not be known by the client application.

Automatic Queue and Topic Generation

The JMS Grid Directory Service and the JMS Grid VM Directory Service will generate queue and topics automatically, as follows:

- If a lookup is performed using any name containing the case-insensitive string "queue", and that name does not already exist, then a default instance of DefaultQueue is returned.
- If a lookup is performed using a name containing the case-insensitive string "topic", and that name does not already exist, then a default instance of DefaultTopic is returned.

4.2.5 User Security

If you wish to make use of JMS Grid's security facilities, you should specify a User name and password when you create the connection:

QueueConnection queueConnection =

```
qcf.createQueueConnection(username,password)
```

or

```
TopicConnection topicConnection =
tcf.createTopicConnection(username,password)
```

Full details of JMS Grid's security features are described in **JMS Grid Security** on page 90.

4.2.6 JMS Messages

Message Types

JMS defines five different types of message, depending on the payload carried:

1. ObjectMessage - for transporting any serializable Java object.
2. BytesMessage - for transporting a stream of uninterpreted bytes.
3. StreamMessage - for transporting a self-defining stream of Java primitives.

   A StreamMessage differs from a BytesMessage in that it 'knows' the type of the primitives stored in it, and will throw an exception of an attempt is made to read bytes and convert them to the wrong primitive.
4. TextMessage - for transporting String objects, including XML documents.
5. MapMessage - for transporting a self-defining set of name-value pairs where names are Strings and values are Java primitive types.

Message Headers

All JMS Messages contain a fixed set of header fields. These contain values used by both clients and providers to identify and route messages. Some header fields are automatically set by the JMS provider, while others may be set by the JMS client.

Message Properties

All JMS Messages also contain a set of property values. These contain additional information about the message. Most properties are defined by the client application, and are set by the client application before the message is delivered. The JMS standard defines a number of properties that JMS providers may, or may not, set. JMS providers may also define properties of their own.

A message consumer can define a MessageSelector that is used to filter incoming messages on the basis of property values. For details see Message Selectors.

Creating a Message

Message objects are created using a variety of methods on the Session object. For the full list of methods, see the Javadocs for Session and the five different message types.

```
// Create an Object message in one step
ObjectMessage om = session.createObjectMessage(anObject);

// Create an Object message in two steps
ObjectMessage om = session.createObjectMessage();
om.setObject(anObject);

// Create a Bytes message
BytesMessage bm = session.createBytesMessage();
// then use various methods on BytesMessage to write bytes to
// the message

// Create a Text message in one step
TextMessage = session.createTextMessage(aString);

// Create a Text message in two steps
TextMessage tm = session.createTextMessage();
tm.setText(aString);
// Create a Stream message
StreamMessage sm = session.createStreamMessage();
// then use various methods on StreamMessage to write bytes to
// the message

// Create a Map message
MapMessage mm = session.createMapMessage
// then use various methods on MapMessage to write name-value
// pairs to the message
```

4.2.7 Publish and Subscribe Messaging Using Topics

Creating a TopicPublisher

A JMS client publishes messages to a topic using a TopicPublisher, which is created from the TopicSession object using the method createPublisher(). This method has one argument which is normally a Topic object but which may be null.

// topicSession is an instance of TopicSession

// topic is an instance of Topic

```
TopicPublisher publisher = topicSession.createPublisher(topic);
```

Having created a `TopicPublisher` you can then set default values for the `deliveryMode`, priority and `timeToLive` parameters for messages published using this `TopicPublisher`.

```
void setDeliveryMode(int deliveryMode)
```

deliveryMode defines whether the message is persistent or non-persistent. It may have the values DeliveryMode.PERSISTENT or DeliveryMode.NON_PERSISTENT. See Error! Reference source not found. on page Error! Bookmark not defined. for more information.

```
void setPriority(int defaultPriority)
```

priority is the message priority. JMS defines a 10 level priority value with 0 as the lowest and 9 as the highest. Clients should consider 0-4 as gradients of normal priority and 5-9 as gradients of expedited priority. Priority is set to 4, by default. Messages with higher priority may be delivered before messages with lower priority, though this depends on the JMS provider.

```
setTimeToLive(long timeToLive)
```

timeToLive is the length of time in milliseconds from its dispatch time that a produced message should be retained by the message system.

Publishing Messages

Having obtained a TopicPublisher, you can then publish messages using one of the publish() methods. The methods normally used are:

```
void publish(Message message)

void publish(Message message,int deliveryMode, int priority, long
timeToLive)
   // publisher is an instance of TopicPublisher
   // message is an instance of Message (or one of its subtypes)
   publisher.publish(message);
```

If the TopicPublisher was created using a null topic it is known as an 'unidentified producer' and the topic must be supplied on every message publish using one of the following methods:

```
void publish(Topic topic, Message message)

void publish(Topic topic, Message message,int deliveryMode, int
priority, long timeToLive)
```

If you use the methods with the deliveryMode, priority and `timeToLive` parameters then these override any defaults set on the `TopicPublisher`

Creating a TopicSubscriber

A JMS client subscribes to a topic using a `TopicSubscriber`, which is created from the `TopicSession` object using one of a variety of methods. The choice of method depends on whether the subscription is non-durable or durable and whether you wish to specify a message selector.

The most common method for creating a non-durable subscription is:

```
TopicSubscriber createSubscriber(Topic topic)
   // topicSession is an instance of TopicSession
   // topic is an instance of Topic

   TopicSubscriber subscriber = topicSession.createSubscriber(topic);
```

You can also specify a message selector. This is a text string that defines criteria used to filter incoming messages.

```
TopicSubscriber createSubscriber(Topic topic, String messageSelector,
Boolean nolocal)
```

The use of JMS message selectors is described in Message Selectors.

If you wish to create a durable subscription the JMS specification defines two methods for you to use:

```
TopicSubscriber createDurableSubscriber(Topic topic, String
subscriptionName)

TopicSubscriber createDurableSubscriber(Topic topic, String
subscriptionName,String messageSelector, Boolean nolocal)
```

Durable subscriptions are described in the section Durable Subscriptions.

JMS Grid also provides a number of additional methods to support JMS Grid's extensions to the JMS message selector mechanism, for both non-durable and durable subscriptions. These are described in *Error! Reference source not found. on page Error! Bookmark not defined*.

Having created a TopicPublisher you can then receive messages either synchronously or asynchronously.

Receiving Messages Synchronously

Messages may be received synchronously by calling one of the receive() methods on the TopicSubscriber. These will block until a message is received or until a timeout expires:

```
Message receive() throws JMSException
```

Receive the next message. This call blocks indefinitely until a message is produced.

```
// subscriber is an instance of TopicSubscriber
Message message = subscriber.receive();
```

```
Message receive(long timeout) throws JMSException
```

Receive the next message that arrives within the specified timeout interval (given in milliseconds). This call blocks until a message arrives or the timeout expires. A timeout of zero never expires and the call blocks indefinitely.

```
Message receiveNoWait() throws JMSException
```

Receive the next message if one is immediately available. If a message is not immediately available this call returns null.

Receiving Messages Asynchronously

Messages may be received asynchronously by creating an object that implements the MessageListener interface and registering it with the TopicSubscriber using the method `setMessageListener()`:

```
void setMessageListener(MessageListener listener)
```

The MessageListener interface defines one method, `onMessage()`, which is called whenever a message arrives.

```
// subscriber is an instance of TopicSubscriber
subscriber.setMessageListener(new MessageListener() {
    public void onMessage(javax.jms.Message message) {
        // process the message
    }
});
```

Durable Subscriptions

If a client needs to receive all the messages published on a topic, including the ones published while the subscriber is inactive, it should use a durable TopicSubscriber. The JMS provider will retain a record of this durable subscription and will ensure that all messages from the topic's publishers are retained until they are either acknowledged by this durable subscriber or they have expired.

The idea is that while the subscriber is not running, the messages that it would otherwise have received are stored by the JMS provider. When the subscriber is restarted, the stored messages are delivered.

A durable subscription is defined by a subscription name and a client ID. When the subscriber is restarted, the same subscription name and client ID must be used as when the subscription was first created.

The subscription name is a string specified by the client. The client ID is an attribute of the JMS connection, and can be specified using the setClientID() method of the JMS Connection.

Only one session at a time can have a TopicSubscriber for a particular durable subscription.

Note: *If you wish to create durable subscriptions you may first need to configure the underlying messaging product to use a durable message store. For more information, see the JMS Grid driver manual for the messaging product (either JMS Grid Message Server or a third-party product) you are using.*

To create a durable subscription, use one of the createDurableSubscription() methods on the session:

```
TopicSubscriber createDurableSubscriber(Topic topic, String
subscriptionName)

TopicSubscriber createDurableSubscriber(Topic topic, String
subscriptionName,String messageSelector, Boolean nolocal)
   // topicSession is an instance of TopicSession
   // Create a durable topic subscriber
   String subscriptionName;TopicSubscriber subscriber =
   session.createDurableSubscriber(topic,subscriptionName);
```

The TopicSubscriber can then be used to receive messages just like any other TopicSubscriber.

When the client wishes to terminate it can close down the subscriber, session and connection in the normal way. All messages subsequently published to the topic will be stored by the JMS provider.

When the client restarts it should create a connection using the same clientID as before, and create a durable TopicSubscriber using the same subscription name as before. The JMS provider will then deliver all the messages that were published to the topic while the client was inactive.

When the client no longer requires the durable subscription is should end it using the `unsubscribe()` method:

```
// We've finished with the durable subscription for the last time
session.unsubscribe(subscriptionName);
```

This method uses the supplied subscriptionName and the clientID of the connection to identify the subscription to be cancelled.

4.2.8 Point-to-Point Messaging using Queues

Creating a QueueSender

A JMS client sends messages to queue using a QueueSender, which is created from the QueueSession object using the method createSender(). This method has one argument which is normally a Queue object but which may be null.

```
// queueSession is an instance of QueueSession
// queue is an instance of Queue
QueueSender sender = queueSession.createSender(queue);
```

Having created a QueueSender you can then set default values for the deliveryMode, priority and timeToLive parameters for messages published using this QueueSender.

Sending Messages

Having obtained a QueueSender, you can then send messages using one of the send() methods. The methods normally used are:

```
void send(Message message)

void send(Message message,int deliveryMode, int priority, long
timeToLive)
   // sender is an instance of QueueSender
   // message is an instance of Message (or one of its subtypes)
   sender.send(message);
```

If the QueueSender was created using a null topic it is known as an 'unidentified producer' and the queue must be supplied on every message publish using one of the following methods:

```
void publish(Queue queue, Message message)

void publish(Queue queue, Message message,int deliveryMode, int
priority, long timeToLive)
```

If you use the methods with the deliveryMode, priority and timeToLive parameters then these override any defaults set on the QueueSender.

Creating a QueueReceiver

A JMS client receives messages from a queue using a QueueReceiver, which is created from the QueueSession object.

The most common method for creating a QueueReceiver is

```
QueueReceiver createReceiver(Queue queue)
   // queueSession is an instance of QueueSession
   // queue is an instance of Queue
   QueueReceiver receiver = queueSession.createReceiver(queue);
```

You can also specify a message selector:

```
QueueReceiver createReceiver(Queue queue, String messageSelector)
```

The use of JMS message selectors is described in Message Selectors.

Having created a QueueReceiver you can then receive messages either synchronously or asynchronously.

Receiving Messages Synchronously

Messages may be received synchronously by calling one of the `receive()` methods on the QueueReceiver. These will block until a message is received or until a timeout expires:

```
Message receive() throws JMSException
```

Receive the next message. This call blocks indefinitely until a message is produced.

```
// receiver is an instance of QueueReceiver
Message message = receiver.receive();
```

```
Message receive(long timeout) throws JMSException
```

Receive the next message that arrives within the specified timeout interval (given in milliseconds). This call blocks until a message arrives or the timeout expires. A timeout of zero never expires and the call blocks indefinitely.

```
Message receiveNoWait() throws JMSException
```

Receive the next message if one is immediately available. If a message is not immediately available this call returns null.

Receiving Messages Asynchronously

Messages may be received asynchronously by creating an object that implements the `MessageListener` interface and registering it with the `QueueReceiver` using the method `setMessageListener()`:

```
void setMessageListener(MessageListener listener)
```

The MessageListener interface defines one method, `onMessage()`, which is called whenever a message arrives.

```
// receiver is an instance of QueueReceiver
receiver.setMessageListener(new MessageListener() {
    public void onMessage(javax.jms.Message message) {
        // process the message
    }
});
```

Browsing Messages on a Queue

A client application may create a QueueBrowser to look at the messages on a queue without actually removing them from it. The QueueBrowser has a method `getEnumeration()` which returns an Enumeration of the queue's messages:

```
QueueBrowser browser=session.createBrowser(queue);
Enumeration enum=browser.getEnumeration();
while (enum.hasMoreElements()) {
```

```
System.out.println("Message on queue is: "+iter.nextElement());
}
```

It is also possible to specify a message selector. In this case the Enumeration will only contain messages that satisfy some condition defined in the message selector:

```
String selector="price BETWEEN 10 and 100";
QueueBrowser browser=session.createBrowser(queue,selector);
```

The JMS specification does not define whether the QueueBrowser should represent a snapshot of the queue or whether it is dynamically updated. However with JMS Grid a snapshot is taken when the call is made to `getEnumeration()`.

4.2.9 Local Transactions

A transaction allows you to send or consume a set of messages as a single operation, so that either all the messages are processed or none of them are.

When messages are produced in a transaction, the messages will only be really sent or published when the transaction is committed. If the transaction is rolled back then all messages sent or published within the transaction will not be delivered and will instead be discarded.

When messages are consumed in a transaction, then the client will only acknowledge that it has consumed any messages when the transaction is committed. If the transaction is rolled back instead then the messages will not be acknowledged.

This section describes local transactions. A local transaction is the simpler of the two types of transaction supported by JMS Grid. A local transaction only involves messages produced and consumed within the session and nothing else. A global transaction, on the other hand, might involve multiple JMS sessions together with multiple databases.

Support for local transactions in JMS Grid is implemented using the native facilities of the underlying messaging product or, for those products that do not support transactions, in the JMS Grid framework itself.

Starting a Local Transaction

If you wish to send or consume messages within a local transaction you need to create a transacted session. This is created in exactly the same way as a non-transacted session, using `QueueConnection.createQueueSession()` or `TopicConnection.createTopicSession()`, but with the `isTransacted` argument of these methods set to true (See Transacted Sessions – Local Transactions.

The following code sample shows how to create a transacted QueueSession:

```
boolean isTransacted = true;
int acknowledgeMode = 0;
QueueSession queueSession =
queueConnection.createQueueSession(isTransacted,acknowledgeMode);
```

The following code sample shows how to create a transacted TopicSession:

```
boolean isTransacted = true;
int acknowledgeMode = 0;
TopicSession TopicSession =
topicConnection.createTopicSession(isTransacted,acknowledgeMode);
```

Note: *When the isTransacted argument is set to true the acknowledgeMode argument is ignored.*

Creating a transacted session automatically starts a local transaction. There is no separate method to start the transaction.

Committing a Local Transaction

A local transaction is committed by calling the `commit()` method on the Session object:

```
void commit() throws JMSException
```

If the session is not transacted then a `javax.jms.IllegalStateException` will be thrown. Committing a local transaction automatically starts a new local transaction.

Rolling Back a Local Transaction

A local transaction is rolled back by calling the `rollback()` method on the Session object:

```
void rollback() throws JMSException
```

If the session is not transacted then a `javax.jms.IllegalStateException` will be thrown.

Rolling back a local transaction automatically starts a new local transaction.

Closing a transacted session (using `Session.close()`) automatically rolls back the transaction.

Sending Messages in a Local Transaction

When you have started a local transaction by creating a transacted session you can create a `QueueSender` or `TopicPublisher` and use it to `send()` or `publish()` messages, just as you would with a non-transacted session.

However because the session is in a transaction, the call to `QueueSender.send()` or `TopicPublisher.publish()` will not actually cause the message to be delivered. Only when you commit the transaction by calling `Session.commit()` method will the messages actually be sent to the topic or queue.

The commit is atomic. This means that if the commit fails for any reason then none of the messages will be sent. You will never get some of the messages being sent and some not being sent.

The following code sample shows two text messages being sent to a queue within a single local transaction (an example using topics would be very similar). Only when the `commit()` is performed are the two text messages actually sent.

```
// queueConnection is an instance of QueueConnection

// create a transacted QueueSession
boolean isTransacted = true;
int acknowledgeMode = 0;
QueueSession queueSession = queueConnection.createQueueSession
(isTransacted,acknowledgeMode);
```

```
// obtain a queue
// ctx is a javax.naming.Context
Queue queue = (Queue)ctx.lookup(queueName);

// create a QueueSender on the queue
QueueSender queueSender = queueSession.createSender(queue);

// start the JMS connection
queueConnection.start();

// create a TextMessage
TextMessage tm1 = queueSession.createTextMessage
    ("This is message 1");

// Send the message
queueSender.send(tm1);

// create a second TextMessage
TextMessage tm2 = queueSession.createTextMessage
    ("This is message 2");

// Send the message
queueSender.send(tm2);

// commit the transaction - this actually sends the messages
queueSession.commit()
```

If for some reason you don't want to commit the transaction (perhaps because an error occurred) you can instead call `rollback()` to abandon all message sends that were performed during the transaction. If your client crashes without committing then the session will automatically be rolled back.

```
// roll back the transaction
queueSession.rollback()
```

Consuming Messages in a Local Transaction

You can also use a transacted session to consume messages within a local transaction. Create a QueueReceiver or TopicSubscriber and then call either `setMessageListener()` to register a message listener or `receive()` to receive an individual message, just as you would with a non-transacted session.

Because the session is in a transaction, the client will only acknowledge that it has consumed any messages when the session is committed.

The following code sample shows two text messages being received synchronously (i.e. using `receive()`) from a queue within a single local transaction. Only when the `commit()` is performed are the two text messages actually acknowledged.

```
// queueConnection is an instance of QueueConnection

// create a transacted QueueSession
boolean isTransacted = true;
int acknowledgeMode = 0;
QueueSession queueSession =
queueConnection.createQueueSession(isTransacted,acknowledgeMode);

// obtain a queue
// ctx is a javax.naming.Context
Queue queue = (Queue)ctx.lookup(queueName);
```

```
// create a QueueReceiver on the queue
QueueReceiver queueReceiver =
    queueSession.createReceiver(queue);

// start the JMS connection
queueConnection.start();

// receive a message (and block until it arrives)
Message m1 = queueReceiver.receive()

// receive a second message (and block until it arrives)
Message m2 = queueReceiver.receive()

// commit the transaction - this acknowledges the receipt of
// the messages
queueSession.commit()
```

If for some reason you don't want to commit the transaction (perhaps because an error occurred) you can instead call `rollback()`. This means that none of the messages received during the transaction will be acknowledged. The JMS provider will consider them to have not been sent and will subsequently attempt to deliver them again. If your client crashes without committing then the local transaction will automatically be rolled back.

```
// roll back the transaction
queueSession.rollback()
```

You can receive messages asynchronously (i.e. using an event listener) within a local transaction in just the same way. The following code sample shows two text messages being received asynchronously from a queue within a single transaction. Only when `onMessage()` is called a second time is `commit()` called. This acknowledges receipt of this and the previous message.

```
// queueConnection is an instance of QueueConnection
// create a transacted QueueSession
boolean isTransacted = true;
int acknowledgeMode = 0;
queueSession QueueSession = queueConnection.createQueueSession
isTransacted,acknowledgeMode);
// obtain a queue
// ctx is a javax.naming.Context
Queue queue = (Queue)ctx.lookup(queueName);
// create a QueueReceiver on the queue
QueueReceiver queueReceiver =
    queueSession.createReceiver(queue);
// start the JMS connection
queueConnection.start();
// initialize the count of messages received
int messagesReceived=0;
// register a message listener to asynchronously receive messages
queueReceiver.setMessageListener(new MessageListener() {
    public void onMessage(javax.jms.Message message) {
        // increment the count of messages received
        messagesReceived++;
        // process the message
        . . .
        if (messagesReceived=2) {
            // acknowledge this and the previous message
            queueSession.commit()
            // re-initialize the count of messages received
            messagesReceived=0;
        }
    }
```

```
});
```

Any attempt to explicitly acknowledge a message by calling `Message.acknowledge()` within a local transaction will be ignored, even if the acknowledgeMode parameter was set to `Session.CLIENT_ACKNOWLEDGE`.

Avoiding Redelivery Loops

If a number of messages are consumed within a local transaction and the transaction is subsequently rolled back, the JMS provider will re deliver them immediately. If the rollback occurred because the message was "bad" in some way then this is likely to cause the message to be rolled back a second time. This can lead to an endless loop where the same message is repeatedly rolled back and redelivered. Your application should therefore avoid rolling back the transaction when an error occurs. It should handle the error in some other way and commit the transaction as normal.

4.2.10 Global Transactions

A global transaction is one that involves multiple transactional resources participating within the same transaction. When the global transaction is committed or rolled back, all the transactional resources are committed or rolled back in a single atomic operation. A global transaction typically involves a number of JMS sessions and a number of databases.

Global transactions must be coordinated by a transaction manager, and are started, committed or rolled back by the manager. They therefore tend to be used only in conjunction with an application server that includes a transaction manager.

4.2.11 Message Selectors

Although some message consuming applications are interested in processing all the messages they receive on a particular topic or queue, many applications need to be more selective.

The simplest approach is for the message consuming application to receive all messages and simply discard the ones it is not interested in.

An alternative approach is to use Message Selectors. This allows the selection of messages to be performed by the JMS provider rather than by each client. The use of a Message Selector reduces the complexity of the client and may significantly decrease the amount of network traffic.

A client that uses Message Selectors defines a conditional expression based on message headers and properties.

The following example shows a client subscribing to messages on a topic, which wishes to filter incoming messages on the basis of an application-defined message property called `price`:

```
Topic topic = session.createTopic("bank.alltrades");
String selector = "price BETWEEN 10 and 100";
boolean noLocal=false;
TopicSubscriber subscriber=session.createSubscriber(topic, selector,
noLocal);
```

The argument `noLocal` denotes whether messages produced by its own connection should be delivered.

Conditional expressions can refer to any application-defined, JMS-defined or JMS Grid specific property. They can also refer to any JMS header (with the exception of `JMSDestination` and `JMSReplyTo`). The above example assumes that messages have a user-defined property price.

The syntax that can be used is defined in the JMS specification. It is based on the WHERE syntax of the SQL92 standard. A detailed definition of the syntax, with examples, can be found in the Javadocs for the `javax.jms.Message` interface.

Message selectors may also be defined that filter certain types of message on the basis of their payload, thereby avoiding the need to use user-defined message properties. This is an additional feature of JMS Grid that goes beyond the JMS standard. For full details *See* **Content Based Message Selectors** on page 181.

Support for JMS message selectors in JMS Grid is implemented using the native facilities of the underlying messaging product or, for those products that do not support JMS message selectors, in the JMS Grid framework itself.

4.2.12 Closing Down

When an application is finished, it should:

- Close any message producer or message consumer objects

```
queueReceiver.close();
queueSender.close();
topicSubscriber.close();
topicPublisher.close();
```

- Close any open sessions. This will also close any message producer or consumer objects that are still open.

```
session.close();
```

- Close the JMS connection. This will also close any open sessions.

```
connection.close();
```

4.3 Additional Programming Features

This section describes a number of additional features of JMS Grid that go beyond those defined in the JMS standard. The features covered are as follows:

- Wildcard Destinations
- Content Based Message Selectors
- Subscription Listening
- The Session Inbox
- Detecting Slow Consumers

4.3.1 Wildcard Destinations

JMS Grid allows you to create Queue and Topic objects whose names contain wildcard characters. This allows a single Queue or Topic object to be used to send messages to, or consume messages from all the destinations that match the wildcard.

Wildcard Syntax

The wildcard syntax assumes that a queue or topic name is made up of a series of elements separated by dots. An asterisk (*) matches an entire element. A greater-than symbol (>) matches any number of elements up to the end of the name.

Table 39 Wildcard Syntax

Wildcard Character	Matches
*	an entire element
>	any number of trailing elements

For example

```
Trade.* matches Trade.settled and Trade.not_settled but not
Trade.fixed.apples

Trade.*.apples matches Trade.fixed.apples but not Trade.apples,
Trade.settled or Trade.not_settled

Trade.> matches Trade.settled, Trade.not_settled, Trade.apples and
Trade.variable.apples.
```

Note that you can't perform matches on part of an element. So

```
Tra> matches Tra> but nothing else

Tra* matches Tra* but nothing else
```

This means that the only wildcard destinations that will match a single-element destination name (i.e. one that does not contain dots) are > and *

```
> matches all destinations
* matches all single-element destinations
```

Creating a Wildcard Destination

Wildcard destinations are created in exactly the same way as ordinary destinations, except that the name of the destination (not its JNDI name) is the required wildcard string.

Here is an example that uses the method `Session.createQueue()` to create a queue object:

```
Queue queue = queueSession.createQueue ("Trade.*.apples");
```

Here is an example that creates an instance of the JMS Grid class `DefaultTopic`. This can either be used directly or stored in a JNDI namespace.

```
// assumes you have imported com.spirit.wave.message.DefaultTopic;
```

```
DefaultTopic topic = new DefaultTopic("Trade.>");
```

For full details of how to create a destination object See *Error! Reference source not found. on page Error! Bookmark not defined.*

4.3.2 Content Based Message Selectors

JMS Grid extends the Message Selector capabilities of JMS to allow message selectors to be defined that filter certain types of message on the basis of their payload.

Object messages that contain a serialized JavaBean object may be filtered on the basis of properties of the bean.

Text messages that contain XML documents may be filtered on the basis of the document contents. Two alternative techniques are available. The simplest is an extension of the JMS message selector syntax that allows selectors to refer to elements and attributes in an XML document hierarchy. The alternative technique allows the message selector to be defined using the powerful XPath syntax.

These additional features remove the JMS restriction that filtering criteria be placed in the message header and reduces the complexity of client code.

If you wish to use one of these extended message selector syntaxes you need to tell JMS Grid which syntax you wish to use. You do this by creating your `TopicSubscriber` or `QueueReceiver` using additional JMS Grid versions of the methods `createSubscriber()`, `createDurableSubscriber()` and `createReceiver()`:

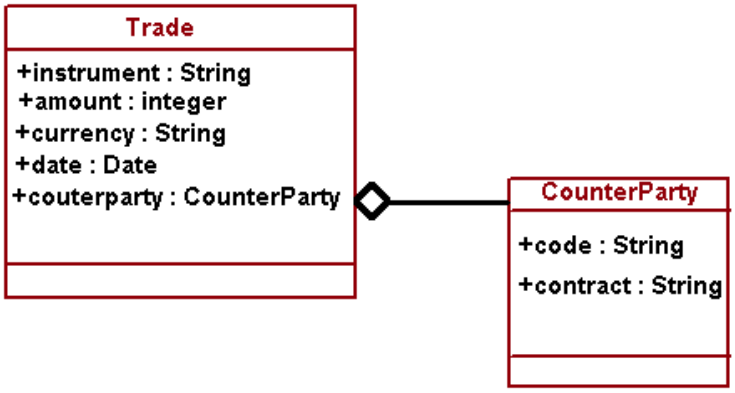
Message Selectors Based on Bean Properties

Object messages that contain a JavaBean object may be filtered on the basis of properties of the bean.

To use this feature with a topic you need to cast your `TopicSession` to a `com.spirit.wave.jms.WaveTopicSession` and then create the `TopicSubscriber` using either the method `createBeanSubscriber()` or the method `createDurableBeanSubscriber()`, depending on whether a non-durable or durable subscription is required.

If you are using a queue you need to cast your `QueueSession` to a `com.spirit.wave.jms.WaveQueueSession` and then create the `QueueReceiver` using the method `createBeanReceiver()`.

Figure 63 Bean Properties



In the following example, the message payload is a JavaBean of the class `Trade`. This bean has a number of properties, including the `integer` property amount, the `String` property `currency` and the property `counterparty`, which is another JavaBean of the class `Counterparty`. The `Counterparty` bean has a number of properties, including the `String` property code.

You will now define a message selector such that only those trades whose value is greater than GBP 1000 and whose counterparty is 'JPM' are delivered.

If you are using a non-durable subscription to a topic, the code looks like the following:

```
// assumes you have imported com.spirit.wave.jms.WaveTopicSession;

Topic   topic;
Class   beanClass;
String  selector;
Boolean noLocal;
TopicSubscriber subscriber;

topic = ctx.lookup("bank.alltrades");

// set the class of the bean to which the selector applies
beanClass = Trade.class;

selector = "amount > 1000 AND currency='GBP' AND
        counterparty.code='JPM'";
noLocal = false;

subscriber
        =(WaveTopicSession)topicSession.createBeanSubscriber
                (topic,beanClass,selector,noLocal)
```

The syntax of such message selectors is identical to that for ordinary JMS message selectors except that the message selector refers to properties of the specified JavaBean instead of properties defined in the message header. Furthermore, if a particular property refers to another bean then a 'dot' notation may be used to refer to the other bean's properties.

If you are using a durable subscription (*See Error! Reference source not found. on page Error! Bookmark not defined.*) you should use the method `createDurableBeanSubscriber()`:

```
subscriber =
(WaveTopicSession)topicSession.createDurableBeanSubscriber(topic,subs
criptionName,beanClass,selector,noLocal);
```

If you are using a queue you need to cast your QueueSession to a `com.spirit.wave.jms.WaveQueueSession` and then create the `QueueReceiver` using the method `createBeanReceiver()`:

```
// assumes you have imported com.spirit.wave.jms.WaveQueueSession;
Queue queue;
Class beanClass;
String  selector;
QueueReceiver receiver;
queue = ctx.lookup("bank.alltrades");
// set the class of the bean to which the selector applies
beanClass = Trade.class;

selector = "amount > 1000 AND currency='GBP' AND
counterparty.code='JPM'";
receiver =
(WaveQueueSession)queueSession.createBeanReceiver(queue,beanClass,sel
ector);
```

Message Selectors to Filter XML Documents using SQL-92 Syntax

Text messages that contain an XML document may be filtered using an extension of the JMS message selector syntax that allows selectors to refer to elements and attributes in an XML document hierarchy.

Consider the following simple XML document representing a trade:

```
<?xml version="1.0" encoding="UTF-8"?>
<! -- comments --->
<trade>
    <instrument>SUNQ</instrument>
    <amount>1000</amount>
    <currency>GBP</currency>
    <date>2001-10-24</date>
    <counterparty>
        <code>JPM</code>
        <contract>10030<contract>
    </counterparty>
</trade>
```

The following example shows how a message selector can be defined so that only those trades whose value is greater than GBP 1000 and whose counterparty is "JPM" are delivered.

If you are using a non-durable subscription to a topic, the code looks like the following:

```
Topic   topic;
String  selector;
Boolean noLocal;
TopicSubscriber subscriber;
topic = ctx.lookup("bank.alltrades");
selector = "amount > 1000 AND currency="GBP" AND
counterparty.code='JPM'";
selector = "xml(c" + selector +")";
noLocal = false;
subscriber = topicSession.createSubscriber(topic,selector,noLocal)
```

The syntax of such message selectors is identical to that for ordinary JMS message selectors except that the message selector can also refer to tag names in the XML

document hierarchy, with the topmost element ignored. Using a 'dot' notation it is possible to refer to a node deep on the document hierarchy. The last element of the dot notation can either refer to an attribute on the node, or simply refer to a node that has a text child node.

The "xml()" wrapper in the selector string tells JMS Grid that this is an xml selector string and not a standard JMS selector string.

If you are using a queue the code looks like the following:

```
Queue   queue;
String  selector;
queue = ctx.lookup("bank.alltrades");
selector = "amount > 1000 AND currency='GBP' AND
                                    counterparty.code='JPM'";
selector = "xml(c" + selector + ")";
QueueReceiver receiver;
receiver = queueSession.createReceiver(queue,selector);
```

Message Selectors to Filter XML Documents using Xpath Syntax

Text messages that contain an XML document may be filtered using the XPath syntax. The XPath syntax allows more complex filtering rules to be defined than is possible using the simple XML filtering described in the previous section. It does, however, require a completely different syntax to the SQL-92-based JMS Grid message selector syntax.

The following example is based on the same XML document structure as was used in the previous section, and shows how a message selector can be defined so that only those trades whose value is greater than GBP 1000 and whose counterparty is "JPM" are delivered.

If you are using a non-durable subscription to a topic, the code looks like the following:

```
Topic   topic;
String  selector;
Boolean noLocal;
TopicSubscriber subscriber;

topic = ctx.lookup("bank.alltrades");
selector="//trade[amount>1000 and currency='GBP'] and //trade/
counterparty[code='JPM']";
selector = "xpath("+selector+")";
noLocal = false;

subscriber = topicSession.createSubscriber(topic,selector,noLocal)
The syntax of such message selectors is defined in the XPath
specification,
http://www.w3.org/TR/xpath#corelib.
```

The "xpath()" wrapper in the selector string tells JMS Grid that this is an xpath selector string and not a standard JMS selector string.

If you are using a queue you need to cast your QueueSession to a `WaveQueueSession` and then create the `QueueReceiver` using the method `createXPathReceiver()`:

```
Queue   queue;
String  selector;
QueueReceiver receiver;

queue = ctx.lookup("bank.alltrades");
```

```
selector="/trade[amount > 1000 and currency='GBP'] and /trade/
counterparty[code='JPM']";
selector = "xpath(" + selector +")";
receiver = queueSession.createReceiver(queue,selector);
```

4.3.3 Subscription Listening

A JMS Grid client can listen for "subscription events" on a particular destination. A subscription event occurs whenever a client creates or closes a subscription to a topic or queue. Note that although the term "subscriber" is normally used only with topics, JMS Grid subscription events occur with both topics and queues.

Subscription events have a number of uses:

- They allow a message producer to determine whether or not any clients have subscribed to a given destination before it produces any messages to it, thus avoiding unnecessary network traffic.
- They can be used to monitor subscription patterns in a distributed environment
- When used in conjunction with a session inbox they can be used to send extra information to new subscribers. For details see The Session Inbox on page 15.

Topic Subscription Events

A client that wishes to listen for subscription events on a particular topic must first create an event listener for these events. This is an instance of a class that implements the `TopicSubscriptionEventListener` interface. This interface defines just one method, `onEvent()`, which will be called whenever a subscription event occurs:

```
// assumes you have imported
    // com.spirit.wave.TopicSubscriptionEventListener
    // and com.spirit.wave.TopicSubscriptionEvent

    TopicSubscriptionEventListener tsel =
        new TopicSubscriptionEventListener() {
            public void onEvent(TopicSubscriptionEvent tse) {
                if (tse.isStarted())
                    // A subscribe event has occurred
                    ......
                } else {
                    // An unsubscribe event has occurred
                    ......
                }
            }
        };
```

The client must then register this event listener with the connection:

```
    // waveTopicConnection is an instance of
    // com.spirit.wave.jms.WaveTopicConnection
    waveTopicConnection.addTopicSubscriptionEventListener
        (topic,tsel);
```

When a subscription event occurs, the method `onEvent(arg)` is called on the event listener. A `TopicSubscriptionEvent` object is supplied as its argument.

A subscribe event occurs either when a client creates a TopicSubscriber on the specified topic, or calls `waveTopicConnection.start()`, whichever occurs last.

An unsubscribe event occurs when a client calls `topicSubscriber.close()` to terminate a subscription to the specified topic, or when this is done implicitly by closing the session or connection.

The TopicSubscriptionEvent object has a method `isStarted()` which can be used to determine whether the event is a subscribe event or an unsubscribe event. It returns true for a subscribe event and false for an unsubscribe event.

The TopicSubscriptionEvent object also has a method `getTopic()`, which returns the Topic on which the subscription event occurred. For a full list of the available methods, see the JMS Grid javadocs.

Queue Subscription Events

A client that wishes to listen for subscription events on a particular queue must create an event listener that implements the `QueueSubscriptionEventListener` interface. This interface defines a just one method, `onEvent()`, which will be called whenever a subscription event occurs:

```
// assumes you have imported
// com.spirit.wave.QueueSubscriptionEventListener
// and com.spirit.wave.QueueSubscriptionEvent

QueueSubscriptionEventListener qsel =
    new QueueSubscriptionEventListener() {
        public void onEvent(QueueSubscriptionEvent qse) {
            if (qse.isStarted())
                // A subscribe event has occurred
                ......
                } else {
                // An unsubscribe event has occurred
                ......
            }
        }
    };
```

The client must then register this event listener with the connection:

```
// waveQueueConnection is an instance of
// com.spirit.wave.jms.WaveQueueConnection
waveQueueConnection.addQueueSubscriptionEventListener
   (queue,tsel);
```

When a subscription event occurs, the method `onEvent(arg)` is called on the event listener. A `QueueSubscriptionEvent` object is supplied as its argument.

A subscribe event occurs either when a client creates a `QueueReceiver` on the specified queue, or calls `queueConnection.start()`, whichever occurs last.

An unsubscribe event occurs when a client calls `queueReceiver.close()` to terminate a subscription to the specified queue, or when this is done implicitly by closing the session or connection.

The QueueSubscriptionEvent object has a method `isStarted()` which can be used to determine whether the event is a subscribe event or an unsubscribe event. It returns true for a subscribe event and false for an unsubscribe event.

The QueueSubscriptionEvent object also has a method `getQueue()`, which returns the Queue on which the subscription event occurred. For a full list of the available methods, see the JMS Grid javadocs.

Subscription Events from Multiple Destinations

When registering an event listener, the topic or queue specified may use wildcards to specify that the listener should receive subscription events from multiple destinations; See Wildcard Destinations on page 7.

4.3.4 The Session Inbox

JMS Grid's session inbox facility allows a message producer to send messages to a new queue receiver or topic subscriber that will be received by that consumer and no others.

The previous section described how a JMS client could listen for subscription events on a given destination. For example, a message producer can listen for subscription events on the destination it is producing messages to. When a new subscriber appears, the message producer will receive an event. In some applications, the message producer needs to be able to send a message to the new subscriber. However it can't simply send the message to the destination as it would be received by existing consumers as well. It needs a way of sending the message to the new subscriber and not to any others.

Imagine a JMS application in which a message producer publishes price updates to a particular topic. Whenever the price of some commodity changes, a price update message is published to the topic. When a new subscriber appears, the producer would like to be able to initialize the new subscriber by sending it a complete set of prices. The producer could simply publish this information to the topic. However all subscribers, not just the new one, would then receive it. What it really needs is a direct one-to-one communication link with a single subscriber. This is what the session inbox provides.

All JMS Grid sessions have a special destination that can be used by subscription event listeners to communicate messages to that client session. This is known as the "session inbox". There are two types of session inbox: the inbox for a `TopicSession` is a Topic while the inbox for a `QueueSession` is a Queue.

Producing messages to the Inbox

Before a client can send or publish messages to another client's session inbox it must first have received a subscription event from the other client.

Subscription events are described in Subscription Listening. When a subscription event occurs, the listener's `onEvent()` method is executed, with the subscription event itself passed as an argument. The inbox destination can then be obtained from the subscription event.

Publishing Messages to the Inbox for a TopicSession

If the listener is listening for subscription events on a Topic then the session inbox will also be a Topic, and may be obtained from the `TopicSubscriptionEvent` using the method `getTopicReplyTo()`:

```
// tse is an instance of TopicSubscriptionEvent
Topic inbox = tse.getTopicReplyTo();
```

Alternatively, a convenience method has been provided which allows a `TopicPublisher` to be obtained directly from the `TopicSubscriptionEvent` using the method `getReplyToPublisher()`:

```
// tse is an instance of TopicSubscriptionEvent
TopicPublisher inboxPublisher = tse.getReplyToPublisher();
```

In the following example, a client creates a subscription event listener which listens for subscription events on a particular topic. When a subscription event is received, the listener obtains the `TopicPublisher` for the subscribing client's session inbox and publishes a message to it.

```
// assumes you have imported
com.spirit.wave.TopicSubscriptionEventListener
// and com.spirit.wave.TopicSubscriptionEvent
TopicSubscriptionEventListener tsel = new
TopicSubscriptionEventListener() {
    public void onEvent(TopicSubscriptionEvent tse) {
        if (tse.isStarted())
            // A subscribe event has occurred
            // communicate a message to the new subscriber
            try {
                TopicPublisher pub=e.getReplyToPublisher();
                // TopicSession is an instance of TopicSession
                TextMessage tm = topicSession.createTextMessage();
                tm.setText("Hello from the publisher");
                pub.publish(m);
            } catch(Exception ex) {
                // error handling
            }
        }
    }
};

// waveTopicConnection is an instance of
com.spirit.wave.jms.WaveTopicConnection
waveTopicConnection.addTopicSubscriptionEventListener(topic,tse l);
```

Sending Messages to the inbox for a QueueSession

If the listener is listening for subscription events on a Queue then the session inbox will also be a Queue, and may be obtained from the QueueSubscriptionEvent using the method getQueueReplyTo():

```
// qse is an instance of QueueSubscriptionEvent
Queue inbox = qse.getQueueReplyTo();
```

Alternatively, a convenience method has been provided which allows a `QueueSender` to be obtained directly from the `QueueSubscriptionEvent` using the method `getReplyToSender()`:

```
// qse is an instance of QueueSubscriptionEvent
QueueSender inboxSender = qse.getReplyToSender();
```

Consuming Messages from the Inbox

If the session is a WaveTopicSession, a client subscribes to the session inbox using the method `createInboxSubscriber`. This returns a `TopicSubscriber`:

```
// waveTopicSession is an instance of
com.spirit.wave.jms.WaveTopicSession
TopicSubscriber inbox = waveTopicSession.createInboxSubscriber();
// messageListener is an instance of MessageListener
inbox.setMessageListener(messageListener)
If the session is a WaveQueueSession, a client creates a receiver on
the session inbox using the method createInboxReceiver(). This
returns a QueueReceiver:
// waveQueueSession is an instance of
// com.spirit.wave.jms.WaveQueueSession
QueueReceiver inbox = waveQueueSession.createInboxReceiver();
// messageListener is an instance of MessageListener
inbox.setMessageListener(messageListener)
```

4.3.5 Detecting Slow Consumers

A slow consumer is a JMS client that is taking a long time to process incoming messages, causing a backlog to build up in the JMS provider. It means that an asynchronous consumer is not returning from `onMessage()` quickly enough, or that a synchronous consumer is not calling `receive()` frequently enough.

When JMS Grid detects that a client is a slow consumer, a 'slow consumer' event is raised. The client can detect this event and take appropriate action (which might be to terminate).

Listening for Slow Consumer Events

A JMS Grid client may register to receive slow consumer events by creating an object that implements the `SlowConsumerEventListener` interface and registering it with the JMS connection using the method `addSlowConsumerEventListener()`. This is a method of the four JMS Grid classes used to implement a JMS connection: WaveQueueConnection, `WaveTopicConnection`, `WaveXAQueueConnection` or `WaveXATopicConnection` (all in the package com.spirit.wave.jms), so if your connection object is declared as one of the four JMS connection types you will need to cast it to the appropriate JMS Grid class first:

void `addSlowConsumerEventListener`

(`com.spirit.wave.SlowConsumerEventListener listener`)

The SlowConsumerEventListener interface

The `SlowConsumerEventListener` interface defines one method, `onEvent()`, which is called whenever JMS Grid detects that the client is consuming messages too slowly. Here is the full definition:

```
package com.spirit.wave;
import java.util.EventListener;
public interface SlowConsumerEventListener extends EventListener {
    public void onEvent (SlowConsumerEvent e);
}
```

The SlowConsumerEvent object

When a slow consumer event is raised, a `SlowConsumerEvent` object is supplied as an argument. This is a subclass of `java.util.EventObject`, and has only one method of interest:

```
Object getSource()
```

This returns the JMS connection object on which the event occurred.

4.4 Programming Examples

This section describes the example client programs provided with JMS Grid. You should run these examples and examine the code used to produce them. You can also use these examples as the basis for your own client programs, especially when learning to use JMS Grid.

4.4.1 How to Run the Examples

JMS Grid includes a number of simple examples that demonstrate some of its features. Each example involves running a pair of Java classes, one usually a message producing JMS client and the other usually a message consuming JMS client.

Before Running each Example

Before running any example, ensure that any server processes are running that are required by the underlying messaging product that you are using. For example, if you are using JMS Grid Message Server you should ensure a JMS Grid Message Server daemon is running. If you are using IBM MQSeries you should ensure the queue manager is running.

Running the Examples

To run each example class, open a command window, navigate to the examples directory of your JMS Grid installation, and execute the run command as follows:

```
run className
```

The run command can be found in the examples directory of your JMS Grid installation. On Windows it is the batch file run.bat; on Unix it is the executable file run.

Specifying the JNDI Provider

The File System JNDI provider which ships with JMS Grid in the `fscontext.jar` is used to run the examples.

For more information about specifying a JNDI provider See **Other JNDI Providers** on page 161.

Specifying the Connection Factory

The exampleArguments.properties file is used to specify the JNDI name of the connection factory (either a queue connection factory or a topic connection factory) to be used by this particular example client.

The JNDI name specified must be relative to the context /User. That is, if you specify `queueConnectionFactoryName=MyQueueConnectionFactory` then the examples will look for a connection factory with the name `/user/MyQueueConnectionFactory`.

If you use any connection factory names other than the default (see the following) then these names must already exist in the JNDI namespace.

The default JNDI names used by the examples to obtain connection factories are as follows:

Table 40 JNDI Names Used for Examples

Type of Example	JNDI name used
`Queue Examples`	`/user/QueueConnectionFactory`
`Topic Examples`	`/user/TopicConnectionFactory`

If these JNDI names do not already exist in the JNDI namespace then the first example you run will automatically bind connection factories as follows:

Table 41 Type of Object

JNDI Name	Type of Object
`/user/QueueConnectionFactory`	`A WaveQueueConnectionFactory configured to use JMS Grid Message Server`
`/user/TopicConnectionFactory`	`A WaveTopicConnectionFactory configured to use JMS Grid Message Server`

If you wish to use an underlying messaging product other than JMS Grid Message Server then you will need to configure the connection factories yourself, either programmatically or using the JMS Grid administration tool.

Specifying the JNDI Name of the Destination

The JNDI name specified to be used by this particular example client must be relative to the context /user. That is, if you specify queueName =MyQueue then the examples will look for a connection factory with the name /user/MyQueue.

If you use any destination name other than the default (see the following) then these names must already exist in the JNDI namespace.

If the queueName or topicName parameter is not specified then the examples will obtain queues and topics using the following JNDI names:

Table 42 JNDI Names Used for Examples

Type of Destination Required	JNDI name used
`queue`	`/user/jmsqueue`
`topic`	`/user/jmstopic`

If these JNDI names do not already exist in the JNDI namespace then the first example you run will automatically bind a queue and a topic as follows:

Table 43 JNDI Names Used for Examples Showing Default

JNDI Name	Type of Object
`/user/jmsqueue`	`A DefaultQueue`
`/user/jmstopic`	`A DefaultTopic`

Rebuilding the Examples

To recompile and rebuild the examples, navigate to the examples directory and use the build command.

The build command can be found in the examples directory of your JMS Grid installation. On Windows it is the batch file `build.bat`; on Unix it is the executable file `build`.

4.4.2 List of Examples

The following examples are provided:

- Simple Publish and Subscribe
- Simple Queues
- Durable Publish & Subscribe
- Transacted Sessions
- Message Selectors
- Subscription Events
- The Session Inbox
- The Interactive GUI

4.4.3 Simple Publish and Subscribe

About this Example

This example demonstrates how to perform publish-and-subscribe messaging using JMS Topics.

```
Consumer subscribes to a topic.
Producer publishes five messages of different types to the topic.
Consumer receives the five messages.
```

In this example, it is important to start the subscriber before you start the publisher. This is because in this example any messages that are published to the topic when there are no subscribers are lost. (A later example "Durable Publish & Subscribe" shows how this can be avoided by the use of durable subscriptions).

TestSubscriber receives messages using an asynchronous MessageListener that receives a callback whenever a message is received. There is an alternative method of receiving messages, which is to use the synchronous `receive()` method that blocks until a message is received or a timeout occurs. This is not demonstrated in this example.

Running the Example

Start up two command windows and navigate in each to the examples directory of your JMS Grid installation.

In the first window, start the subscriber using the following command:

```
run pubsub.Consumer
```

In the second window, start the publisher using the following command:

```
run pubsub.Producer
```

Expected Output

When the subscriber is started, it will display output similar to the following:

```
%JMS Grid%\examples>run pubsub.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
328  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version = x.x.x Build:bldxxx.x

Subscriber is ready to receive messages on destination jmstopic.
```

It will then wait for messages to arrive.

When the publisher is started, it will display output similar to the following:

```
%JMS Grid%\examples>run pubsub.Producer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version = x.x.x Build:bldxxx.x
Sending messages to destination jmstopic.
```

```
Producer published TextMessage
Producer published ObjectMessage
Producer published MapMessage
Producer published StreamMessage
Producer published BytesMessage
All messages published.
Producer closed.
Session closed.
Connection closed.
%JMS Grid%\examples>

The subscriber will then receive the messages, with output similar to
the following:
Received:
Text is: First text message
---------------------------------
Received:
Object is: Second message is an object
---------------------------------
Received:
Map message, property 'testvalue' is: Map message value
---------------------------------
Received:
String is: A string in a Stream message
Integer is: 5
---------------------------------
Received:
String is: A string in a Bytes message
Double is: 5.6
Expected number of messages received.
Subscriber closed.
Session closed.
Connection closed.
%JMS Grid%\examples> >
```

Variations

You can extend this test by starting multiple subscriber applications before you start the publisher. Each message published by the publisher will be received by every subscriber.

You can also try running the publisher before the receiver. You will see that with any messages published when there are no subscribers are lost. Compare this with the "Simple Queues" example.

4.4.4 Simple Queues

About this Example

This example demonstrates how to perform point-to-point messaging using JMS queues.

`Producer` publishes five messages of different types to the queue.

`Consumer` receives the five messages.

In this example, it does not matter whether the subscriber or the publisher is started first. This is because any messages that are sent to a queue when there are no

subscribers are retained by the JMS provider until a receiver is started or a timeout occurs.

`Consumer` receives messages using the synchronous `receive()` method, which blocks until a message is received or a timeout occurs. There is an alternative method of receiving messages, which is to define an asynchronous `MessageListener` that receives a callback whenever a message is received. This is not demonstrated in this example.

Amongst other things, this example demonstrates that even though the sender sends messages to the queue at a time when there is no receiver, the messages are retained by the JMS provider and delivered when a receiver is started.

Running the Example

Start up two command windows and navigate in each to the examples directory of your JMS Grid installation.

In the first window, run the sender using the following command:

```
run queues.Producer
```

In the second window, start the receiver using the following command:

```
run queues.Consumer
```

Expected Output

When the sender is started, it will display output similar to the following:

```
%JMS Grid%\examples>run queues.Producer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6
Sending messages to destination jmsQueue.

Producer sent TextMessage
Producer sent ObjectMessage
Producer sent MapMessage
Producer sent StreamMessage
Producer sent BytesMessage

All messages sent.
Producer closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

When the receiver is started, it will display output similar to the following:

```
%JMS Grid%\examples>run queues.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
```

```
282  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version = x.x.x Build:bldxxx.x

Receiver ready to receive messages on destination jmsQueue.
--------------------------------
Receiver -
Text is: First text message
--------------------------------
--------------------------------
Receiver -
Object is: Second message is an object
--------------------------------
--------------------------------
Receiver -
Map message, property 'testvalue' is: Map message value
--------------------------------
--------------------------------
Receiver -
String is: A string in a Stream message
Integer is: 5
--------------------------------
--------------------------------
Receiver -
String is: A string in a Bytes message
Double is: 5.6
--------------------------------

Expected number of messages received.

Subscriber closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

Variations

You can try running the receiver before the publisher. The results will be the same.

You can also try starting more than one receiver before starting the publisher. You should find that each message is received by one and only one receiver. However the behavior you observe will vary depending on the messaging product being used.

4.4.5 Durable Publish and Subscribe

About this Example

This example demonstrates durable subscriptions to topics.

In this example, Consumer is started and creates a durable subscription on a topic. It is then terminated by typing Ctrl-C.

`Producer` is then started and publishes five messages to the topic. Even though the subscriber is no longer running, because that subscriber had created a durable subscription on this topic, the messages are stored by the JMS provider.

`Consumer` is now re-started. This re-establishes the durable subscription and all the messages that were published while it was not running are delivered.

`Consumer` receives messages using an asynchronous MessageListener that receives a callback whenever a message is received. There is an alternative method of receiving messages, which is to use the synchronous receive() method that blocks until a message is received or a timeout occurs. This is not demonstrated in this example.

Running the Example

Start up two command windows and navigate in each to the examples directory of your JMS Grid installation.

In the first window, start the subscriber using the following command:

```
run durable.Consumer
```

This creates the durable subscription. After the subscriber has been running for a few moments, stop it by typing Ctrl-C.

In the second window, start the publisher:

```
run durable.Producer
```

This publishes five messages to the topic.

Now return to the first window and restart the subscriber:

```
run durable.Consumer
```

This reconnects to the durable subscription. The messages that were published while the subscriber was not running are then received.

Expected Output

When the subscriber is first started it will display output similar to the following:

```
%JMS Grid%\examples>run durable.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
282  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version = x.x.x Build:bldxxx.x
Durable subscriber is ready to receive messages on destination
jmstopic.
```

At this point the subscriber can be stopped by pressing Ctrl-C:

Terminate batch job (Y/N)? y

```
%JMS Grid%\examples>
```

Next the publisher is started:

```
%JMS Grid%\examples>run durable.Producer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
265  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version = x.x.x Build:bldxxx.x
Sending messages to destination jmstopic.
Producer published TextMessage
Producer published ObjectMessage
Producer published MapMessage
```

```
Producer published StreamMessage
Producer published BytesMessage

All messages published.
Producer closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

Finally the subscriber is restarted.

```
%JMS Grid%\examples>run durable.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
282  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version x.x.x Build:bldxxx.x

Durable subscriber is ready to receive messages on destination
jmstopic.
---------------------------------
Received:
Text is: First text message
---------------------------------
Received:
Object is: Second message is an object
---------------------------------
Received:
Map message, property 'testvalue' is: Map message value
---------------------------------
Received:
String is: A string in a Stream message
Integer is: 5
---------------------------------
Received:
String is: A string in a Bytes message
Double is: 5.6

Expected number of messages received.

Subscriber closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

4.4.6 Transacted Sessions

About this Example

There are in fact three separate examples:

Example 1: Transacted Send and Receive

This example demonstrates how to use a transacted session to send several messages to a queue within a single transaction. It also demonstrates how to use a transacted session to receive several messages within a single transaction.

```
Producer starts a transaction, sends five messages to a queue and
commits the transaction.
```

`Consumer` starts a transaction, receives the five messages and commits the transaction.

Example 2: Rolling back the Sender

This example demonstrates how if a transaction is rolled back by the sender none of the messages sent within that transaction are sent after all:

`RollbackProducer` starts a transaction, sends five messages to a queue and rolls back the transaction.

When `Consumer` is run a second time it does not receive any messages, showing that they were not actually sent.

Example 3: Rolling back the Receive

Finally this example demonstrates how if the transaction is rolled back by the receiver none of the messages received during the transaction are acknowledged and they will be redelivered:

`Producer` sends five messages to a queue as before.

`RollbackConsumer` starts a transaction, receives five messages and then rolls back the transaction.

When `Producer` is run a second time it receives the same five messages, showing that when the messages were rolled back they were returned to the queue so that they would be redelivered.

Running the Examples

Example 1: Transacted Send and Receive

Start up two command windows and navigate in each to the examples directory of your JMS Grid installation.

In the first window, start the transacted sender using the following command:

```
run transacted.Producer
```

In the second window, start the transacted receiver:

```
run transacted.Consumer
```

In the second window, start the transacted receiver a second time to prove that there are no more messages on the queue. After a few moments, press Ctrl-C to terminate it.

Example 2: Rolling back the Sender

In the first window, start the transacted sender whose transaction is rolled back:

```
run transacted.RollbackProducer
```

In the second window, start the transacted receiver again:

```
run transacted.Consumer
```

After a few moments, press Ctrl-C to terminate the receiver.

Example 3: Rolling back the Receive

In the first window, run the original transacted sender:

```
run transacted.Producer
```

In the second window, start the transacted receiver whose transaction is rolled back:

```
run transacted.RollbackConsumer
```

Expected Output

Example 1: Transacted Send and Receive

When the sender is started, it will display output similar to the following:

```
%JMS Grid%\examples>run transacted.Producer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version = x.x.x Build:bldxxx.x
Sending transacted messages to destination jmsQueue.
Producer sent TextMessage
Producer sent ObjectMessage
Producer sent MapMessage
Producer sent StreamMessage
Producer sent BytesMessage
Committing transaction

All messages sent.
Producer closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

When the receiver is started, it will display output similar to the following:

```
%JMS Grid%\examples>run transacted.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version = x.x.x Build:bldxxx.x

Receiver is ready to receive messages on destination jmsQueue.
--------------------------------
Receiver -
Text is: First text message
--------------------------------
--------------------------------
Receiver -
Object is: Second message is an object
--------------------------------
--------------------------------
Receiver -
Map message, property 'testvalue' is: Map message value
--------------------------------
--------------------------------
Receiver -
String is: A string in a Stream message
Integer is: 5
--------------------------------
--------------------------------
Receiver -
String is: A string in a Bytes message
Double is: 5.6
```

```
--------------------------------
Committing transaction

Expected number of messages received.

Subscriber closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

If you run the receiver a second time, it will display output similar to the following:

```
%JMS Grid%\examples>run transacted.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:5060
7) version = x.x.x Build:bldxxx.x
          Receiver is ready to receive messages on destination
jmsQueue.
```

You will then need to press Ctrl-C to terminate it.

Example 2: Rolling back the Sender

When the sender is started, it will display output similar to the following:

```
%JMS Grid%\examples>run transacted.RollbackProducer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
219  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6
```

Sending transacted messages to destination jmsQueue.

```
Producer sent TextMessage
Producer sent ObjectMessage
Producer sent MapMessage
Producer sent StreamMessage
Producer sent BytesMessage
Rolling back transaction

All messages published.
Producer closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

When the receiver is started, it will display output similar to the following:

```
%JMS Grid%\examples>run transacted.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6
```

Receiver is ready to receive messages on destination jmsQueue.

You will then need to press Ctrl-C to terminate it.

Example 3: Rolling back the Receiver

When the sender is started, it will display output similar to the following:

```
%JMS Grid%\examples>run transacted.Producer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6
Sending transacted messages to destination jmsQueue.
Producer sent TextMessage
Producer sent ObjectMessage
Producer sent MapMessage
Producer sent StreamMessage
Producer sent BytesMessage
Committing transaction

All messages sent.
Producer closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

When the receiver is started, it will display output similar to the following:

```
%JMS Grid%\examples>run transacted.RollbackConsumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Receiver is ready to receive messages on destination jmsQueue.
--------------------------------
Receiver -
Text is: First text message
--------------------------------
--------------------------------
Receiver -
Object is: Second message is an object
--------------------------------
--------------------------------
Receiver -
Map message, property 'testvalue' is: Map message value
--------------------------------
--------------------------------
Receiver -
String is: A string in a Stream message
Integer is: 5
--------------------------------
--------------------------------
Receiver -
String is: A string in a Bytes message
Double is: 5.6
--------------------------------
Rolling back transaction

Expected number of messages received.
```

```
Subscriber closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

If you run the receiver a second time, it will display exactly the same output as before.

4.4.7 Message Selectors

About this Example

This example demonstrates how message selectors can be used to filter incoming messages. It demonstrates both standard JMS message selectors (which filter messages on the basis of message properties and headers) and JMS Grid content-based selectors (which filter certain types of message on the basic of message content).

Four subscribers are started on the same topic:

`NoSelectorConsumer` creates a `TopicSubscriber` with no message selector.

`JMSSelectorConsumer` creates a `TopicSubscriber` with a standard JMS message selector that filters incoming messages on the basis of a user-defined property name.

`XPathSelectorConsumer` creates a TopicSubscriber with a XPath message selector that filters incoming text messages which contain XML documents on the basis of the contents of that document.

`XMLSelectorConsumer` creates a TopicSubscriber with a JMS Grid XML message selector that filters incoming text messages which contain XML documents on the basis of the contents of that document.

`Producer` is then started. This publishes a variety of messages to the topic, some of which satisfy the filter criteria and some which do not.

The four receivers receive the messages that satisfy their defined criteria and do not receive those that do not.

Running the Example

Start up five command windows and navigate in each to the examples directory of your JMS Grid installation.

In the first window, run the subscriber with no message selector:

```
run selectors.NoSelectorConsumer
```

In the second window, run the subscriber which has a standard JMS message selector that filters incoming messages on the basis of a user-defined property name:

```
run selectors.JMSSelectorConsumer
```

In the third window, run the subscriber with a JMS Grid XML message selector that filters incoming text messages which contain XML documents on the basis of the contents of that document:

```
run selectors.XPathSelectorConsumer
```

In the fourth window, run the subscriber with a JMS Grid XML message selector that filters incoming text messages which contain XML documents on the basis of the contents of that document:

```
run selectors.XMLSelectorConsumer
```

Finally, in the fifth window, start the publisher:

```
run selectors.Producer
```

After a few moments, press Ctrl-C to terminate the three subscribers that have not already terminated.

Expected Output

When the five subscribers are first started, they all display output similar to the following:

```
%JMS Grid%\examples>run selectors.NoSelectorConsumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
250  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Subscriber is ready to receive messages.
```

They will all then wait for messages to arrive. When the publisher is started, it will display output similar to the following:

```
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
296  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6
TextMessage sent, payload is: First text message
TextMessage sent, payload is: Hello world
TextMessage sent, payload is: <?xml version="1.0" encoding="UTF-8"
?><Top><Order customer="Fred" /></Top>
TextMessage sent, payload is: <?xml version="1.0" encoding="UTF-8"
?><Top><Order customer="Joe Bloggs" /></Top>
ObjectMessage sent, payload is: TestObject with value = testValue

All messages published.
Publisher closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

The first subscriber, selectors.NoSelectorConsumer, will then receive all the messages, with output similar to the following:

```
---------------------------------
Message received by standard subscriber with no selector:
   TextMessage received, payload is: First text message
---------------------------------
Message received by standard subscriber with no selector:
   TextMessage received, payload is: Hello world
---------------------------------
Message received by standard subscriber with no selector:
```

```
   TextMessage received, payload is: <?xml version="1.0"
            encoding="UTF-8" ?><Top><Order customer="Fred" /></Top>
---------------------------------
Message received by standard subscriber with no selector:
   TextMessage received, payload is: <?xml version="1.0"
     encoding="UTF-8" ?><Top><Order customer="Joe Bloggs" /></ Top>
---------------------------------
Message received by standard subscriber with no selector:
   ObjectMessage received, payload is:
                                  TestObject with value = testValue
---------------------------------
Message received by standard subscriber with no selector:
   ObjectMessage received, payload is:
                                TestObject with value = invalidValue
Terminating...
%JMS Grid%\examples>
```

The second subscriber, JMSSelectorConsumer, will receive the single message that has a property name set to 'testvalue':

```
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
250  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Subscriber is ready to receive messages.
---------------------------------
Message Received by standard Subscriber with selector: name =
'testvalue'
Text is: First text message

Expected number of messages received.

Subscriber closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

The third subscriber, XPathSelectorConsumer, will receive the single text message that contains an XML document with a node Order.customer with the value 'Fred':

```
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
250  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Subscriber is ready to receive messages.
---------------------------------
Message Received by XPathSubscriber with selector: xpath(//
Order[@customer="Fred"])
Text is: <?xml version="1.0" encoding="UTF-8" ?><Top><Order
customer="Fred" /></Top>

Expected number of messages received.

Subscriber closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

The fourth subscriber, XMLSelectorConsumer, will receive the single text message that contains an XML document with a node Order.customer with the value 'Fred'.

```
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Subscriber is ready to receive messages.
---------------------------------
Message Received by XMLSubscriber with selector: xml(Order.customer =
'Fred')
Text is: <?xml version="1.0" encoding="UTF-8" ?><Top><Order
customer="Fred" /></Top>

Expected number of messages received.

Subscriber closed.
Session closed.
Connection closed.

%JMS Grid%\examples>
```

4.4.8 Subscription Events

About this Example

This example demonstrates how subscription notification events can be used to notify a message producer that a new subscriber has appeared.

In this particular example, a JMS client waits until two subscribers are listening for messages on a topic before sending messages to it.

`SubNotifyProducer` listens for subscription events on a topic.

`Consumer` subscribes to this topic. This causes the `SubNotifyProducer` to receive a subscription event.

A second `Consumer` subscribes to this topic. This causes the `SubNotifyProducer` to receive a second subscription event.

The `SubNotifyProducer`, which has been keeping track of the number of subscribers to the topic, detects that the number of subscribers has reached two, and publishes some messages to the topic.

The two `Consumers` receive these messages and terminate.

Running this Example

Start up three command windows and navigate to the examples directory of your JMS Grid installation.

In the first window, start the subscription event listener:

```
run subnotify.SubNotifyProducer
```

In the second window, start a subscriber:

```
run subnotify.Consumer
```

If you look in the first window you will see that the subscription event listener has been notified about the new subscriber.

In the third window, start a second subscriber:

```
run subnotify.Consumer
```

If you look in the first window you will see that the subscription event listener has been notified about the second new subscriber and has started publishing messages. The two subscribers receive these messages.

At the end of the test the subscription event listener will still be waiting for events. Press Ctrl- C to terminate it.

Expected Output

When the subscription event listener is started, it will display output similar to the following:

```
%JMS Grid%\examples>run subnotify.SubNotifyProducer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
265  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6
Publisher awaiting 2 subscriptions...
```

When the first subscriber is started, it will display output similar to the following:

```
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
235  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Subscriber is ready to receive messages.
```

This causes the subscription event listener in the first window to wake up and display the following:

```
Received subscription event:
A subscription has started
Current number of subscribers is now 1
When the second subscription is started, it also displays output
similar to the following:
%JMS Grid%\examples>run subnotify.Consumer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
235  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Subscriber is ready to receive messages.
```

This causes the subscription event listener in the first window to wake up once again and write five messages to the topic:

```
Received subscription event:
A subscription has started
```

```
Current number of subscribers is now 2

All messages published.
```

The two subscribers then receive these messages and then terminate.

```
-----------------------------------
Received:
Text is: First text message
---------------------------------
Received:
Object is: Second message is an object
---------------------------------
Received:
Map message, property 'testvalue' is: Map message value
---------------------------------
Received:
String is: A string in a Stream message
Integer is: 5
---------------------------------
Received:
String is: A string in a Bytes message
Double is: 5.6

Expected number of messages received.

Subscriber closed.
Session closed.
Connection closed.
```

As each subscriber terminates, the subscription event listener is notified that a subscription has ended:

```
Received subscription event:
A subscription has ended
Current number of subscribers is now 1
Received subscription event:
A subscription has ended
Current number of subscribers is now 0

Terminating...
Publisher closed.
Session closed.
Connection closed.
```

4.4.9 The Session Inbox

About this Example

This example demonstrates how JMS Grid's session inbox facility can be used to allow a message producer to send messages to a new subscriber that will be received by that subscriber and no others.

`Producer` listens for subscription events on a topic.

`Consumer1` first subscribes to its session inbox. It then subscribes to the topic.

The `Producer` receives a subscription notification event and sends a message to the session inbox of the new subscriber.

The `Consumer1` receives this message.

Running the Example

Start up two command windows and navigate to the examples directory of your JMS Grid installation.

In the first window, start the subscription event listener:

```
run inbox.Producer
```

In the second window, start a subscriber:

```
run inbox.Consumer1
```

At the end of the test both clients will still be waiting for events. Press Ctrl-C to terminate each client.

Expected Output

When the subscription event listener is started, it will display output similar to the following:

```
%JMS Grid%\examples>run inbox.Producer
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
281  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6
Establishing topic subscription event listener...
Waiting indefinitely for messages ...
```

When the subscriber is started, it will display output similar to the following:

```
%JMS Grid%\examples>run inbox.Consumer1
Initializing JNDI Objects...
0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid Client
version = x.x.x Build:bldxxx.x
297  [main] INFO  com.spirit.jmq.JMQConnection  - Successfully
connected to JMS Grid Message Daemon (daemon=ln-dev-gx260-kd-50607 @
tcp://ln-dev-gx260-kd:50607) version 6.0.6 Build:bld606.6

Subscriber is ready to receive messages.
```

This causes the subscription event listener in the first window to wake up and display the following:

```
Received a subscription event =
Sending a message to the subscriber's session inbox
```

The subscriber receives this message in its session inbox and displays:

```
A message has been received in the inbox
Text is: hello from the publisher
```

Variation

After you have started the InboxSubscriber, leave it running and start a second Consumer2 in another window. When this subscribes to the topic the Producer will again receive a subscription event and publish a message to its session inbox. Notice that this message is received by the new subscriber but not by the original subscriber. This demonstrates that the session inbox conveys messages to a single subscriber only.

4.4.10 The Interactive GUI

About this Example

This interactive example implements an easy to use GUI to enable the User to run variations on many of the examples described above. The GUI allows the User to run both publish/subscribe and send/receive JMS models specifying various different configurations such as whether or not the messages are to be persistent or transacted, whether or not the consumer should be durable or use selectors and whether or not it should receive synchronously or asynchronously. Multiple producers and consumers can also be configured at will.

Summary of Commands

Figure 64 Tabs - What They Do

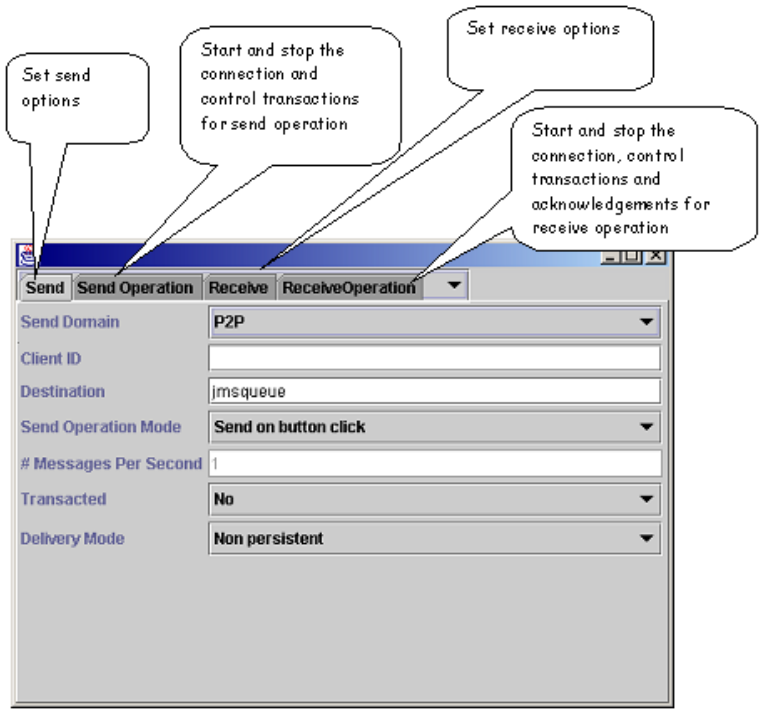


Figure 65 The Send Panel

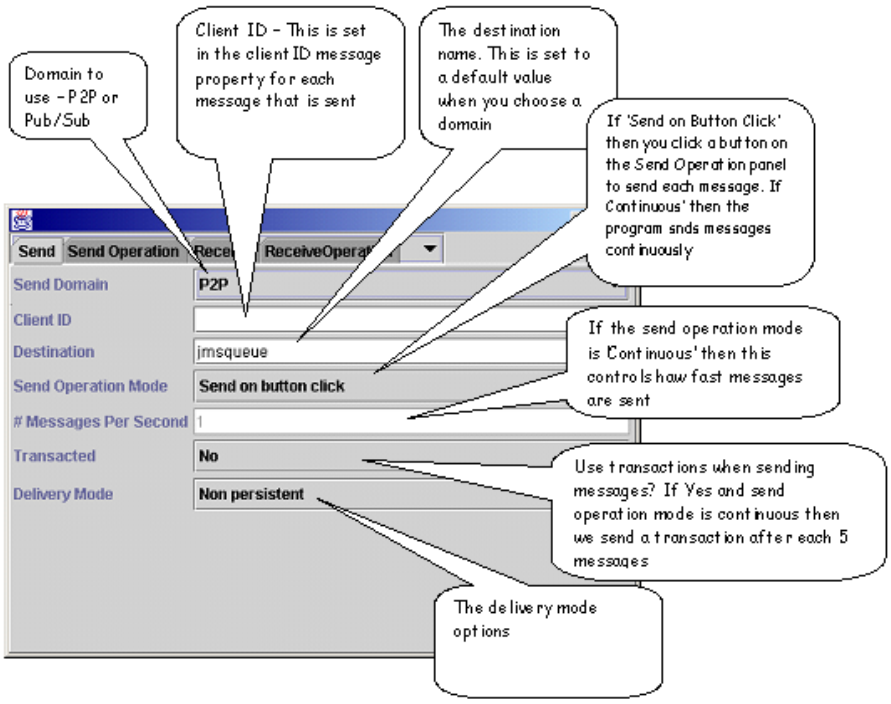


Figure 66 The Send Operation Panel

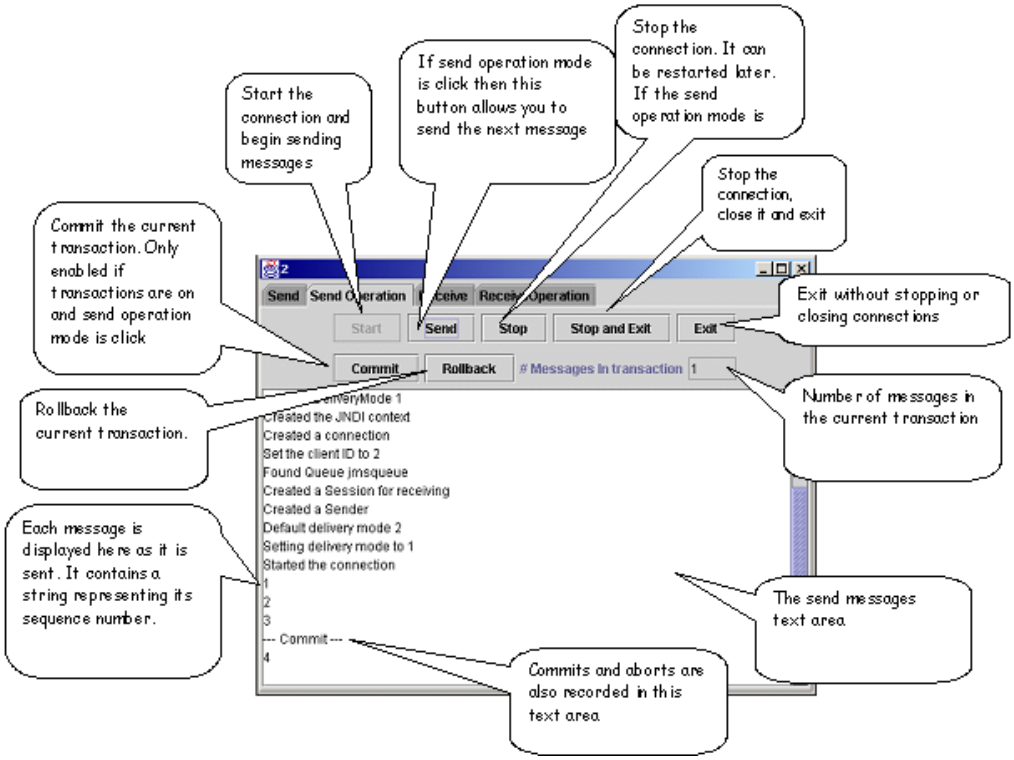


Figure 67 The Receive Panel

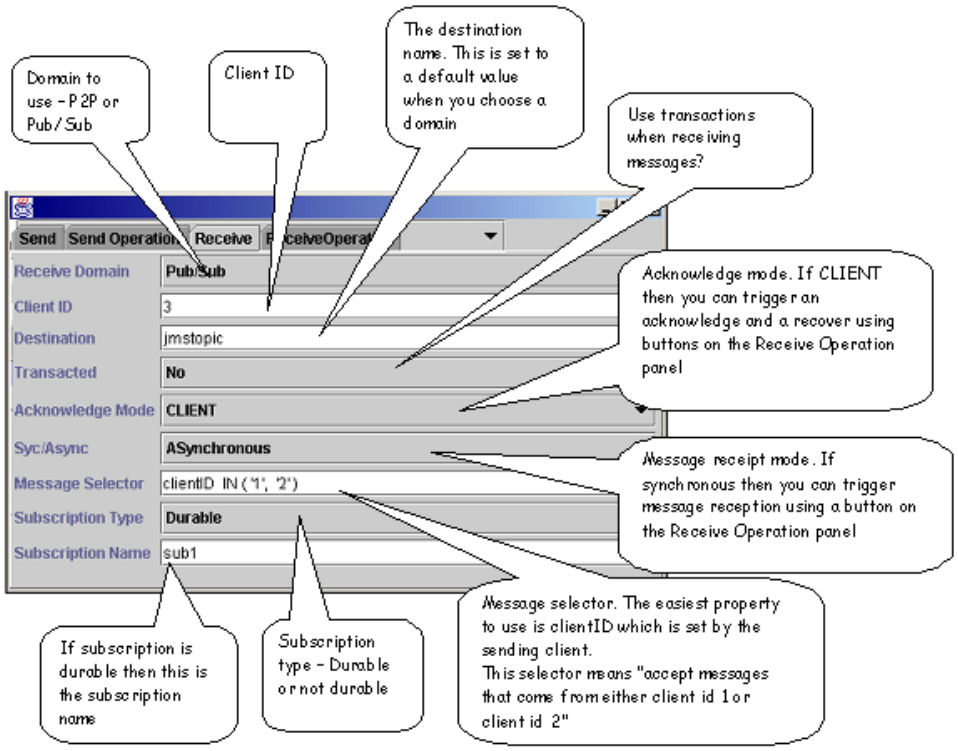
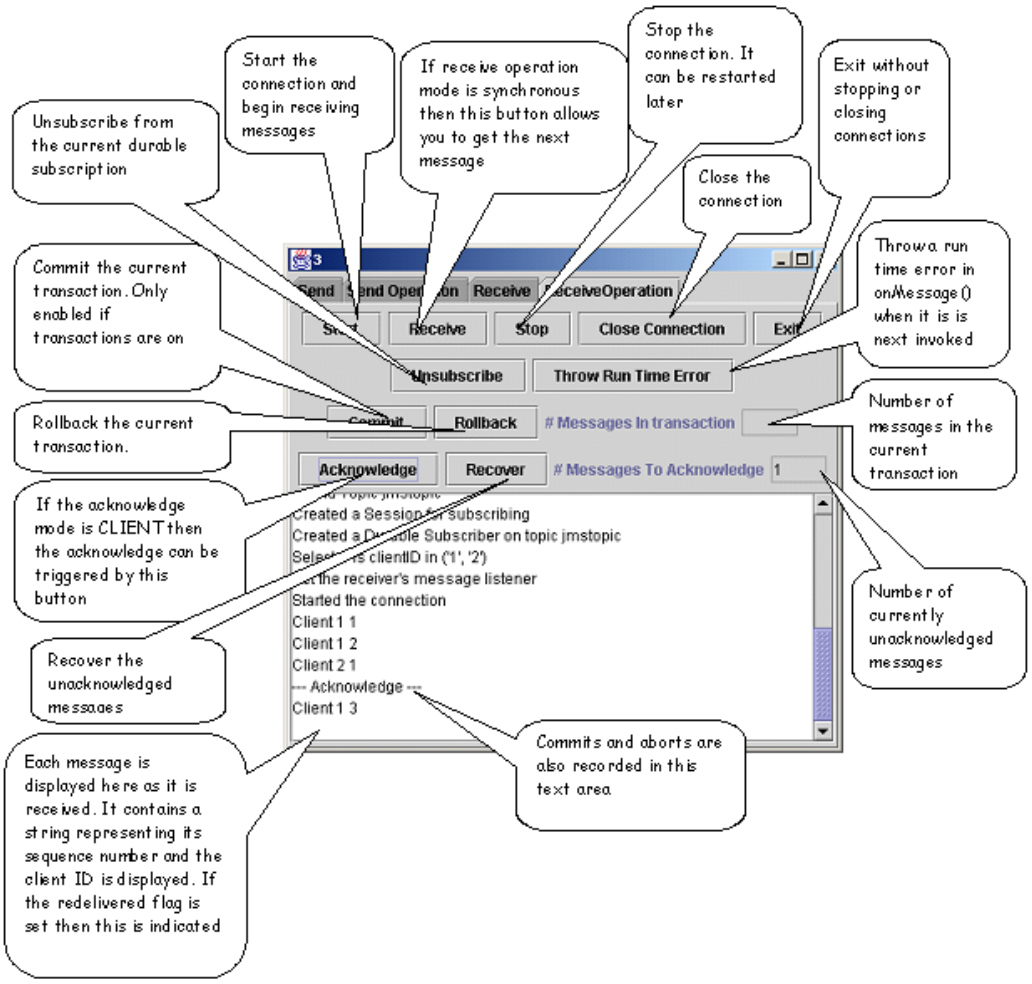


Figure 68 The Receive Operation Panel



Running the Example

There are many different configurations that can be run using the GUI. Here are 2 examples to get you started.

Example 1: Continuous publish/subscribe

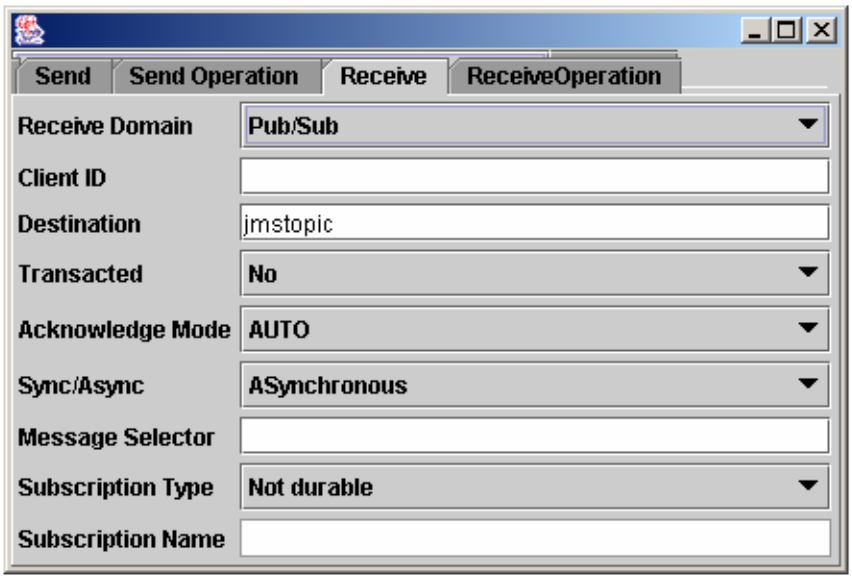
Start up two command windows and navigate to the examples directory of your JMS Grid installation.

In the first window, start a subscriber:

```
run gui.JMSGUI
```

This will fire up a GUI:

Figure 69 Receive Tab - Pub/Sub



Click on the "Receive" tab and select "Pub/Sub" from the "Receive Domain" drop-down.

Click on the "ReceiveOperation" tab and select start.

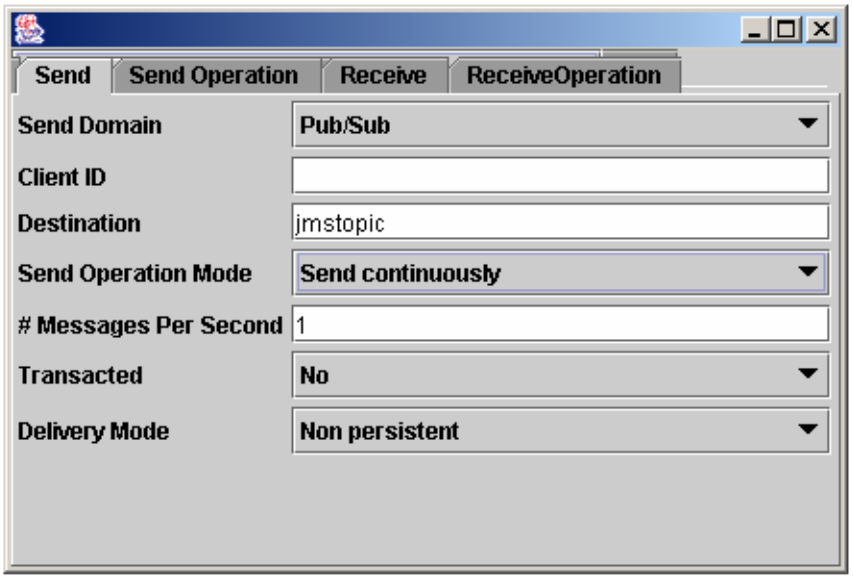
Figure 70 Receive Operation Tab

In the second window, start a publisher:

```
run gui.JMSGUI
```

This will fire up a second GUI:

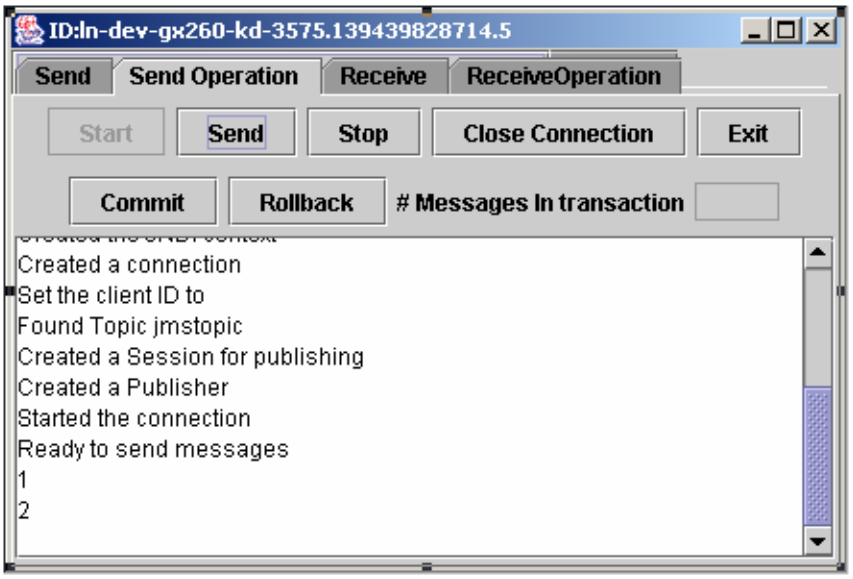
Figure 71 Send Tab



Click on the "Send" tab and select "Pub/Sub" from the "Send Domain" drop-down and select "Send continuously" from the "Send Operation Mode" drop-down.

Click on the "Send Operation" tab and select start.

Figure 72 Send Operation Tab

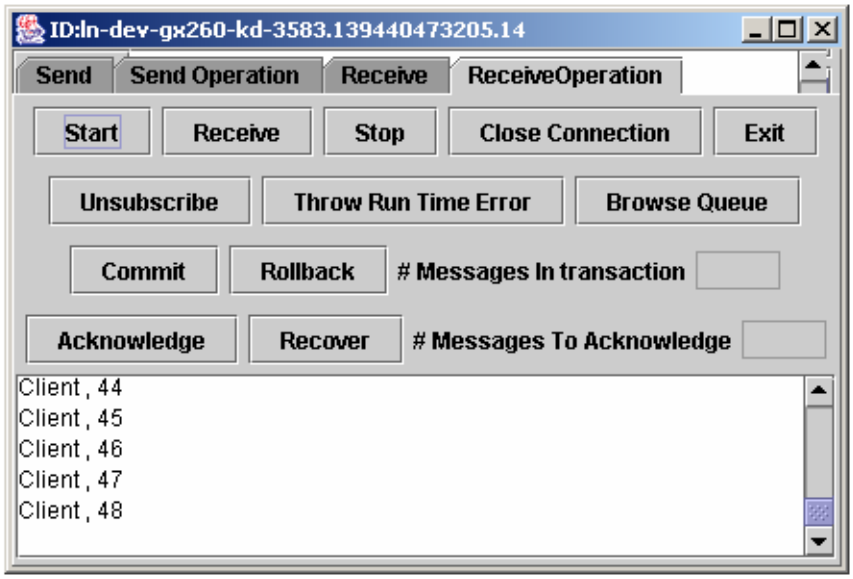


Expected Output

The GUI running the Producer will display the message number as each message is sent.

The GUI running the Consumer will similarly display the message numbers of each message received:

Figure 73 Receive Operation Messages Numbers



Example 2: Non-continuous Synchronous send/receive

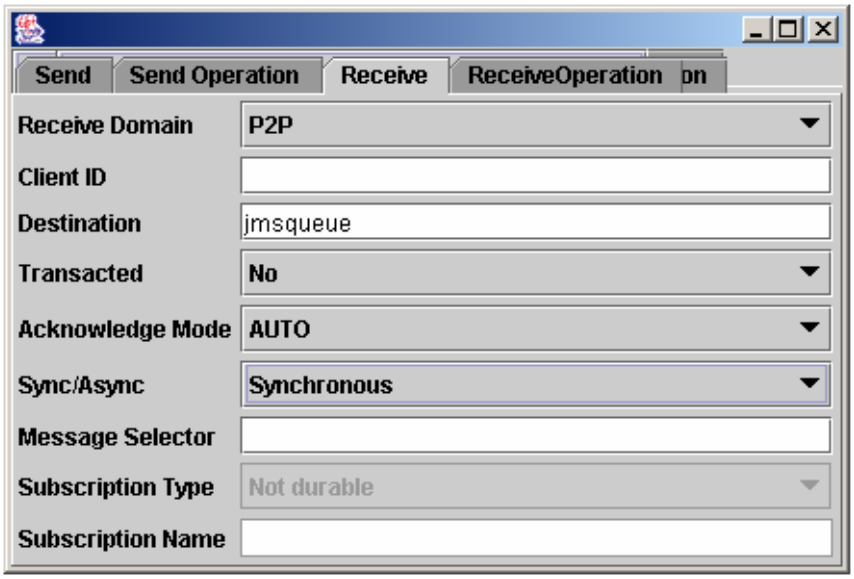
Start up two command windows and navigate to the examples directory of your JMS Grid installation.

In the first window, start a subscriber:

```
run gui.JMSGUI
```

This will fire up a GUI:

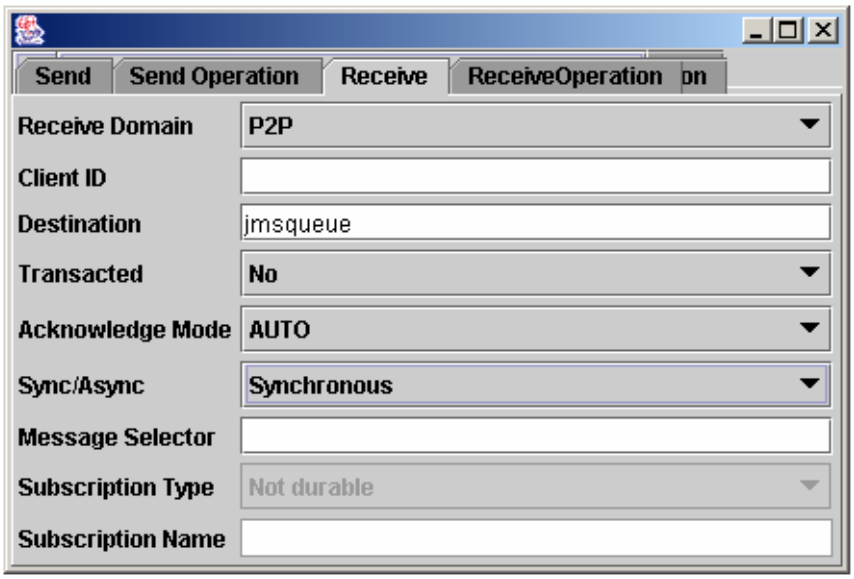
Figure 74 Synchronous Receive



Click on the "Receive" tab and select "P2P" from the "Receive Domain" drop-down and "Synchronous" from the "Sync/Async" drop-down.

Click on the "ReceiveOperation" tab and select start.

Figure 75 Receive Operation Sync

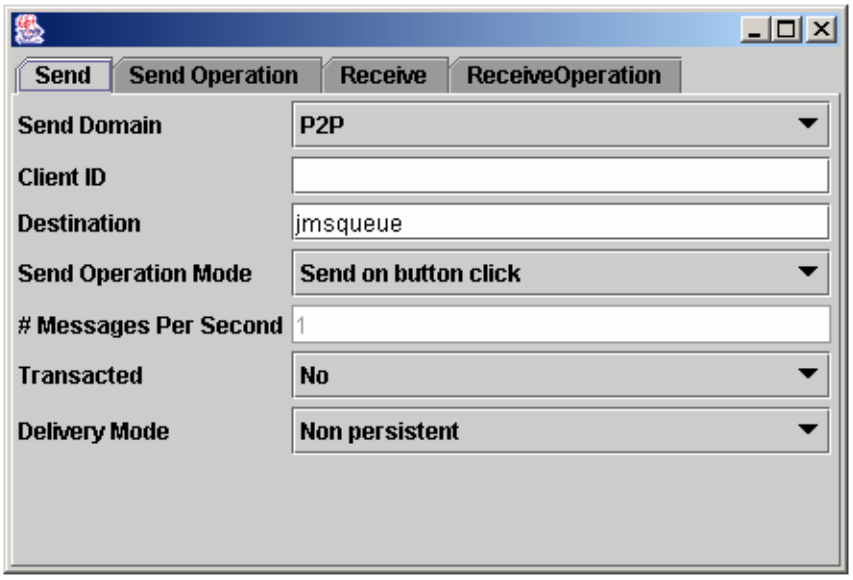


In the second window, start a publisher:

```
run gui.JMSGUI
```

This will fire up a second GUI:

Figure 76 Second Send



Click on the "Send" tab and select "P2P" from the "Receive Domain" drop-down and "Send on button click" from the "Send Operation Mode" drop-down.

Click on the "Send Operation" tab and select start.

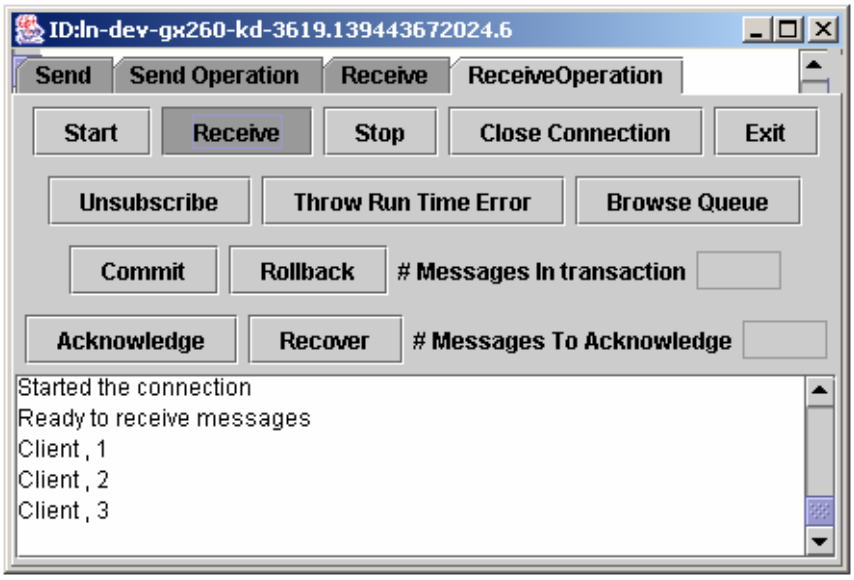
Figure 77 Second Send Operation

Click on send to send a message:

Figure 78 Send the Message

Click on receive to receive a message:

Figure 79 Receive the Message



4.5 References

Textbooks

Java Message Service by Richard Monson-Haefel and David A Chappell O'Reilly 2001. ISBN 0-596-00068-5.

Professional JMS Programming by Paul Giotta and 6 others. Wrox Press 2000. ISBN 1-961004-93-1

Online Resources

Java Message Service Specification **http://java.sun.com/products/jms/docs.html**

Java Message Service Tutorial **http://java.sun.com/products/jms/tutorial/index.html**

Chapter 5

JMX Management

This Chapter describes the JMX compliant management and monitoring capabilities of JMS Grid. It contains the following sections:

- What is JMX - a basic introduction to key JMX terminology
- Management Architecture Overview - describing how the management components interact
- Running the JMS Grid Management Console
- Using the JMS Grid Management Console
- Using the JMS Grid Management Commands
- The Management Object Model - a list of all the MBeans attributes and operations implemented by JMS Grid Message Server and accessible via the jmxConsole.

5.1 What is JMX?

The Java Management eXtensions (also called the JMX specification) define an Architecture, the design patterns, the API's, and the services for application and network management in the Java programming language. It provides the means to instrument Java code, create smart Java agents, implement distributed management middleware and managers, and integrate these solutions smoothly into existing management systems using either HTML or industry standards such as SNMP and WBEM. For more information see **http://java.sun.com/products/JavaManagement/.**

5.1.1 JMX Concepts

This section briefly defines the main JMX concepts and terminology used in this guide.

Manageable Resource

A JMX manageable resource can be an application, an implementation of a service, a device, a user, and so forth. It is developed in Java, or at least offers a Java wrapper, and has been instrumented so that it can be managed by JMX-compliant applications.

The JMS Grid Message Server includes the following manageable resources:

- **Wave Message Daemon MBean** (general management - threads, memory etc.)

- **The Message resource** (the message processing and routing engine)
- **The Store resource** (the message store files for persistent messages)

Management Bean (MBean)

A Management Bean or MBean is the Java object through which management of resources is effected. Its interface makes available all the information that is needed for an application to manage it. JMX defines four types of MBean, and each type is defined by, for instance, implementing certain interfaces, or following certain naming conventions. The four types of MBeans are: standard, dynamic, open and model. Each of these corresponds to different management needs and are appropriate for a particular situation.

- Standard MBeans are the simplest to design and implement; their management interface is described by their method names and the fact that they implement an interface whose name ends in "MBean". These are suitable where the management interface of a resource is stable.
- Dynamic MBeans must implement the DynamicMBean interface and expose their operations and attributes at run time by providing callers with metadata to describe them. These are useful for wrapping non-conformant resources, or where the management interface is changing.
- Open MBeans are dynamic MBeans, which only use the primitive wrapper types and a few others in their operations and attributes. These can be discovered and used by any client at runtime and can be used without needing extra jar files.
- Model MBeans are also dynamic MBeans that are fully configurable and self-described at run-time; they provide a generic MBean class with default behavior for dynamic instrumentation of resources.

All types of MBean expose Attributes and Operations to allow the resource represented by the MBean to be controlled by management applications.

Management Server (MBean Server)

The Managed Bean server, or MBean server for short, is a registry for objects that are exposed to management operations in an agent (see next section for description of an agent). Any object registered with the MBean server becomes visible to management applications - both in the same Java Virtual Machine (JVM), and externally. The MBean Server only exposes an MBean's management interface, never its direct object reference.

Management Agent

A JMX Management Agent is a management entity that runs in a JVM and acts as the liaison between the MBeans and the management application. A JMX agent is composed of an MBean server, a set of MBeans representing managed resources, a minimum number of agent services implemented as MBeans, and typically at least one protocol adaptor or connector. The agent services are usually MBeans themselves and add manageability to the agent.

Management Application

A Management Application is a JMX client application that interacts with manageable resources via JMX Management Agents in order to control or monitor the resources. The JMS Grid Management Console is a browser based management application. The JMS Grid Management commands are simple standalone programs which run one MBean operation each.

Attributes

Attributes are the fields or properties of an MBean that are in its management interface. Attributes are discrete, named characteristics of the MBean which define its appearance or its behavior, or are characteristics of the managed resource that the MBean instruments.

Attributes are always accessed via method calls on the object that owns them, and the method calls are derived from the attribute names with get- and set- prefixes.

Suppose you have an MBean with an attribute myAttribute. If it is readable, there will be a method called `getMyAttribute`; if it is writable, there will be a setter method `setMyAttribute` to update it. Either or both of these may be present. If the attribute is boolean then the getter may alternatively be called `isMyAttribute`.

Operations

Operations are methods that have been exposed in the MBean interface of a JMX MBean. Any method that is not a “getter“ or “setter“ for an attribute is considered to be an operation. They are used to control a manageable resource - to start, stop or pause it, for example.

Domain

Each object managed by an MBean Server is allocated a namespace according to its domain name. How the domain name is structured is application dependent. The JMS Grid Management API uses domain names of the form:

`WMS.<clusterid>.<daemonid>`

The JMS Grid Management Console constructs the correct domain name from the name and type of the resource being managed.

Notification Model

The notification model is a part of JMX which allows MBeans to broadcast management events such as state changes or error conditions; these are called notifications. Management applications and other objects register as listeners with the broadcaster MBean. The MBean notification model of JMX enables a listener to register only once and still receive all different events that may occur in the broadcaster. The JMX notification model only covers the transmission of events between MBeans within the same JMX agent - external communication to remote management applications is not specified. Sun’s management tools use JMS as the external notification mechanism,

publishing messages on well-known management topics. These are named `com.spirit.channel.management`.

5.1.2 Additional JMS Grid Concepts

JMS Grid Message Server's operational management facilities augment JMX with some further capabilities which are described below.

Advisory Messages

When particular events take place, JMS Grid Message Server may publish advisory messages on well-known management topics. Management tools can subscribe to these topics perhaps using message selectors to filter out specific message types - and react to the events.

Management topics

A management topic is a pre-defined topic reserved for use by Sun products such as the JMS Grid Message Server. This topic acts as a channel for management information messages, for example errors, alerts, notifications and metrics. These messages are issued on well known topics using the topic hierarchy "`com.spirit.channel.management.*`"

Metrics

The JMS Grid Message Server uses metric objects for collecting runtime performance information about the running system. Each metric is an MBean, which uses JMS to distribute change notification events to any active listeners. Examples of metrics include:

- Number of Active Clients
- Message throughput
- Memory utilization
- Thread utilization
- Queue depths
- Topic depths

5.2 Management Architecture Overview

5.2.1 Uses of JMX in JMS Grid

The Management API is used to monitor and control single or clustered JMS Grid Message daemons. Examples of the operations supported include:

- Retrieve a list of durable subscribers by subscription name
- Add/remove durable subscribers to/from a daemon
- Browse messages (a snapshot) by subscription name
- Browse a message, by message Id
- Remove a message, by message Id
- List all current JMS clients, where they are, how long they have been running
- Close down a daemon, or all daemons, now or in a specified time
- Force the re-connect of a client to another daemon

Details of all supported operations can be found in the Javadocs, and in the tables in Management Commands below.

5.2.2 Distributed Architecture

Each JMS Grid daemon contains a set of JMX MBeans that expose the basic operations and attributes for each managed resource.

A remote MBeanServer resides on the client and allows remote access to the MBeans used to manage a JMS Grid daemon instance. Utilizing the JMS-JMX infrastructure available with JMS Grid, the remote MBean Server can be used to group multiple daemon instances - i.e. a cluster of daemons. For convenience, the JMS Grid management API uses concrete classes to manage remote MBeans in the JMS Grid Message daemon, by using the same remote MBeanServer infrastructure. This enables a client (i.e. the servlet based management console) to connect to a single point in the JMS Grid network, and manage multiple cluster instances at the same time. The agent can also be used to provide a gateway portal for integration with enterprise management tools such as Tivoli, HP OpenView or CA Unicenter through Simple Network Management Protocol (SNMP).

The agent and the daemons communicate with each other using JMS publish/subscribe messaging over various management topics. The topic used depends on the scope of the operation (here meant in its broader sense): operations directed at a single daemon use one topic, while those directed at a cluster use a different one rooted at "`com.spirit.channel.management.*`" Operations are dispatched to one or more daemons, addressed using a JMX domain name of the form:

`WMS.<clustername>.<daemonid>`.

The daemons also automatically send metric data, and logging data when requested. Any JMS client can subscribe to the appropriate topic, so it is possible for multiple management consoles - or logging/alerting agents - to listen in to the data being published. See Metrics for more details.

5.3 Running the JMS Grid Management Console

This chapter describes the installation and running of the JMS Grid Management Console. The management console is a Web based GUI that requires a servlet container to be installed and running. This section explains how to use the embedded Servlet container in the JMS Grid Message Daemon. As an alternative you can work through the installation of the JMX management console in a standalone Tomcat Java Web Server.

5.3.1 Using the Servlet Container in a JMS Grid Daemon

This provides the quickest, simplest approach for accessing the jmxConsole. The default behavior of the embedded jmxConsole is to deploy the jmxConsole on port 8080. For example the following will start the JMS Grid Message Daemon and deploy the jmxConsole on the default port.

```
startserver -c (Unix)

startserver /c (Windows)
```

To configure the jmxConsole to run a non-standard port, pass the port value in as the final argument to the wmd script. For example to configure the embedded Servlet Container is use port 80 use:

```
startserver -c 80 (Unix)

startserver /c 80 (Windows)
```

5.3.2 Installing the Management Console in a Web Server

This section describes how to set up the Management Console from its web archive (WAR) file.

Prerequisites

Installation of a servlet engine, this document assumes you are using Tomcat version 4.0 or later. Application servers such as WebLogic have a servlet engine built into them, but you need to consult the documentation for instructions on how to deploy in these environments.

Secure Socket Layer (SSL) must be installed and a certificate made available for the web or application server. If you have not installed this software the section Installing and Configuring SSL will help you. Alternatively, the section Running without SSL explains how to switch this off.

The management console web archive (WAR), jmxConsole.war. In order to run the application you also need a running JMS Grid Message Daemon, wmd.

Note: *It is a "feature" of Tomcat that if a context is defined for a web application then Tomcat does not unpack a war file with that context name. This applies even if the*

application has not been deployed before, i.e. in the absence of a directory with the context name in the webapp directory.

This is justified as necessary to prevent inadvertent overwrites of existing contexts. However, it means that an extra step is introduced into the installation, as unpacking the WAR and creating a context have to be done separately.

1 Deploy the WAR File.

- If Tomcat is running shut it down.
- Copy the WAR file, jmxConsole.war, into Tomcat's web application directory, called webapp, which is a direct child of the Tomcat installation directory. If you are reinstalling the management console delete the jmxConsole directory.
- If you are reinstalling the management console delete the jmxConsole directory and comment out the jmxConsole context from the server.xml file in the conf directory. This directory is a direct child of the Tomcat installation directory. You can comment out using the XML comment delimiters <!-and -->.
- Restart Tomcat, which will then unpack the WAR file into a directory called jmxConsole.
- Stop Tomcat again.

2 Create or Restore a Context for the Web Application.

For this step you need to edit the server.xml file which resides in the conf directory, also a direct child of the Tomcat installation directory. You need to add some lines into the default virtual host's configuration. Look for:

```
<Host name="localhost" debug="0"" appBase="webapps" unpackWARs="true">
```

This occurs at the start of this section. A little further down you will see several tags for contexts. Add these lines after the other Context tags, that is after the last

```
</Context> :
<!-- jmxConsole Context -->
<Context path="/jmxConsole" docBase="jmxConsole" debug="0"
reloadable="false" >
<Logger className="org.apache.catalina.logger.FileLogger"
prefix="localhost_jmxConsole_log." suffix=".txt" timestamp="true"/>
</Context>
```

This declares the existence of the jmxConsole context and arranges for all logging output from the web application to go to localhost_jmxConsole_log.<date>.txt, a file in the logs directory.

3 Restart Tomcat

Once started the web application will now write to its log file `localhost_jmxConsole_log.<data>.txt`.

Some elements of the application are loaded when the server starts so there will be some entries made in the file before you do anything.

If anything goes wrong during start up, for instance if you place the context declaration lines in the wrong place in server.xml, Tomcat will fail to start up again. All the XML files used by Tomcat are validated against a DTD and will be rejected if invalid. Tomcat writes any errors in a file named "`catalina_log.<date>.txt`" in the logs directory.

Check here if there are any problems reported during Tomcat start up. Also, check the console's own log file for any error reports there.

Installing and Configuring SSL

Complete instructions for installing the SSL extension are given in the Tomcat documentation. Go to "Tomcat Documentation" and then the "SSL Config How-to" link. There are quick start instructions at the top of the page, but more extensive ones about half way down under the banner "Configuration."

You should use the self-signed certificate as recommended in these instructions. The documentation for the keytool utility will help you if you wish to import a certificate you already have.

Running without SSL

If you are happy to run without using SSL all you need to do is comment out the security section of the web application's deployment descriptor. This file, `web.xml`, is located in the `webapp/jmxConsole/WEB-INF` directory in the Tomcat installation tree. The security section looks like this:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Login Screen</web-resource-name>
        <url-pattern>/jsp/login</url-pattern>
        <url-pattern>jsp/Login.jsp</url-pattern>
            <http-method>DELETE</http-method>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
            <http-method>PUT</http-method>
    </web-resource-collection>
    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>
</security-constraint>
```

Special Note for Running The Management Console On Unix Variants

The JFreeChart software you have used for drawing the metric graphs has a dependency on the graphical capabilities of the operating system. In particular, it can depend on there being an X server running in order to work. You might see an error such as:

```
Can't connect to X11 window server using ':0.0' as the value of the
DISPLAY variable.
```

Where machines have no graphics capabilities, or there is no X server running, there is a straightforward work around for this. You need to set the system property `java.awt.headless` to be true on the command line of the java virtual machine where the console web application is running.

If you are running Tomcat, the startup scripts have a special variable provided to set extra JVM arguments, `JAVA_OPTS`. To do this for C shell:

```
setenv JAVA_OPTS '-Djava.awt.headless=true'
```

or for `Bourne/bash` shells:

```
JAVA_OPTS='-Djava.awt.headless=true'
```

Running the Management Console

The web application's welcome page has been set to be the log in page for the management console, so you only need point your browser at the web application root to start it. If the application is running on the local host and on the standard Tomcat port this will be **http://localhost:8080/jmxConsole**. Alternatively you can go to the log in page explicitly, **https://localhost:8443/jmxConsole/jsp/Login.jsp**

In more general terms the URL for the management console is to be found at:

http://<hostname>:<port>/jmxConsole

The first screen you see is the Login screen, at which you must enter:

- **Username** - Administrator User name (same as for the Admin tool)
- **Password** - Administrator's password
- **Daemon** - URL of the daemon to be used as an entry point into the network

As the User name and password are passed across the network it is important that you use SSL if the network in which you use the Management Console is unsecured as otherwise these may be compromised. This URL assumes you are using SSL.

5.4 Using the JMS Grid Management Console

You can monitor and control the runtime behavior of a JMS Grid Message Daemon using the JMS Grid Management Console.

Note: *The Management Console allows you to observe the behavior of a JMS Grid Message Daemon and its components, and it allows you to control that behavior 'on the fly,' by setting attribute values or invoking operations on individual MBeans. The changes made directly to attributes or because of invoking an operation will be persistent.*

Figure 80 Overview of the Management Console

(PLACE HOLDER FOR FIGURE 1)

The JMS Grid Management Console is divided into two main views (frames) - the Navigation Tree frame or view, which is on the left, and the information frame or view, which appears on the right. The possible contents of the information frame are discussed below.

5.4.1 Navigation View

The Navigation View allows you to select clusters and daemons within the message network. The Navigation view represents the network topography as a tree. The root of the hierarchy is always shown as "Management Console". If you open this it shows a

node for each of its children, which are all the clusters in the network. Opening a cluster shows first its manageable resources, classified by the MBean type, which is used to manage them.

Each cluster has a generic view, which can be used to manage every daemon in a JMS Grid message cluster as one logical unit. Each cluster generic view has child nodes, one for each daemon in the cluster.

Opening a node shows its manageable resources, again classified by the managing MBean type, or agent, message or store.

You can refresh the Navigation View using the browser's refresh button. The list of managed daemons and clusters is then updated. The Navigation View has also a "QuickSearch" option that allows direct navigation to named components, it uses a full-string match. For example if you wish to navigate to an existent daemon called "ClusteredDaemon1" input the name into the search text field and press "Go", the navigation tree will expand at the node representing "ClusteredDaemon1".

The navigation tree will collapse if no item is found.

What Daemons and Clusters are Visible from the Console?

The Management Console shows all the connected clusters to the entry point (the daemon you connect to). The Management Console can be used to stop message daemons, but not start them.

5.4.2 Information Views

The Information views allow you to access runtime information and invoke management bean operations. Selecting the MBean type, whether in the cluster or daemon scope in the Navigation view, will change the current Information view.

The Information view has a standard layout. It shows a series of tabs, where each tab corresponds to a particular type of management activity applicable to the target (daemon or cluster) for the MBean type (agent, message or store). For instance, if you select the Agent node under a cluster the information view will change to one with two tabs, Operations and Logging. The Management Model section below discusses in detail the management activities available in each target/MBean context.

Each tab in the Information view also has three standard buttons, "Previous" to go back to the previous page in the viewing history, "Next" to go to the next page in the viewing history and "Help" which launches context sensitive help in a separate browser window. (Not included in this release). These buttons appear across the bottom of the tab.

Attributes View

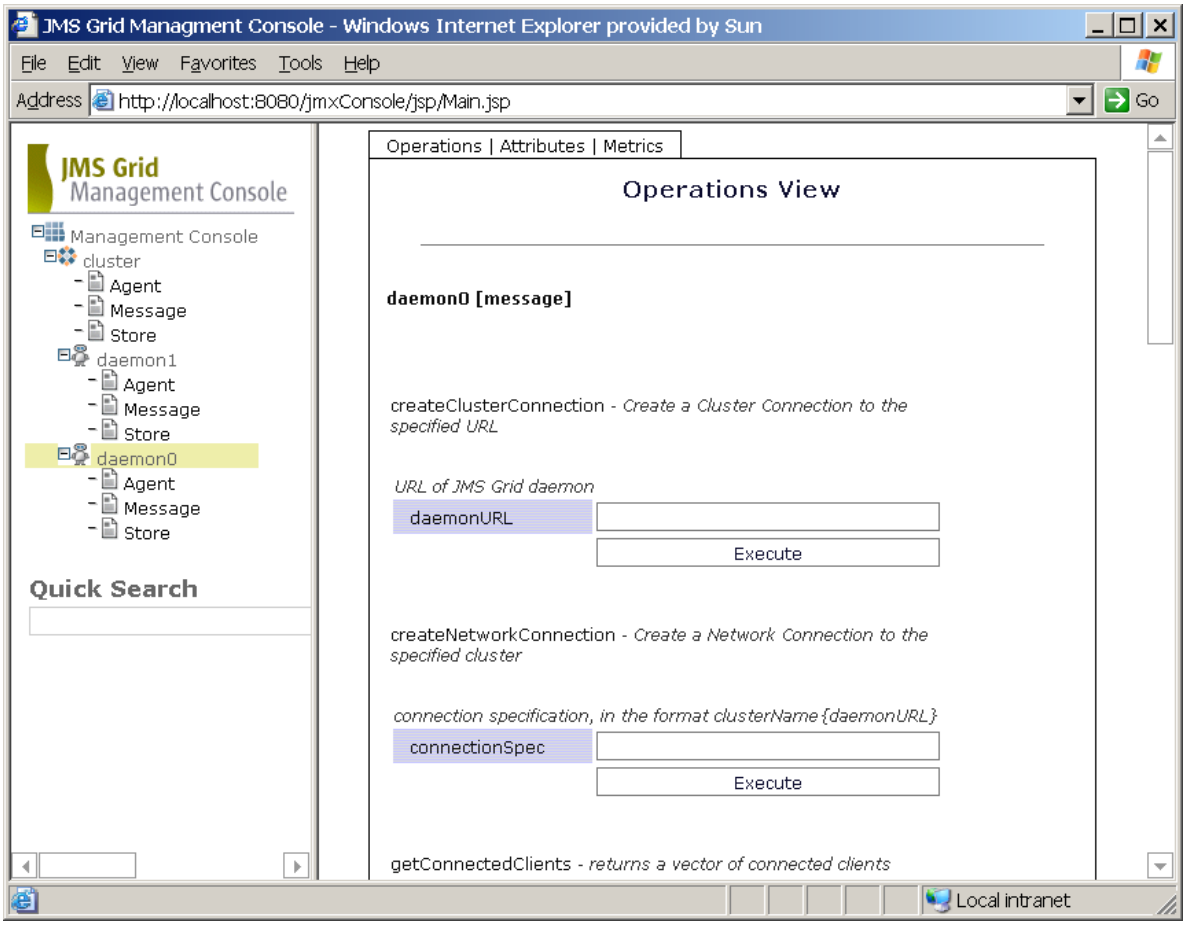
The Attributes view lists all the values for readable JMX attributes associated with the selected MBean, and allows new values to be set for any writable attributes. You can read more about the specific attributes for each MBean in the section The Management Model.

Note: *You can apply changes to multiple attribute values in a single operation, but there is no guarantee as to the order in which those changes are made, and the changes are not batched into a single transaction. If a change to an attribute results in an exception being raised, other changes already made are not rolled back, and subsequent changes are not applied.*

Operations View

The Operations View allows you to invoke operations on the selected MBean. The initial view displays a list of all possible operations. If you select an operation this may result in a more detailed view for the specific operation with buttons representing further operations that are available in that context, or a simple page displaying the status of execution of the operation. You can find out more about the operations available for each resource in the section The Management Model.

Figure 81 Typical Operations View Screen



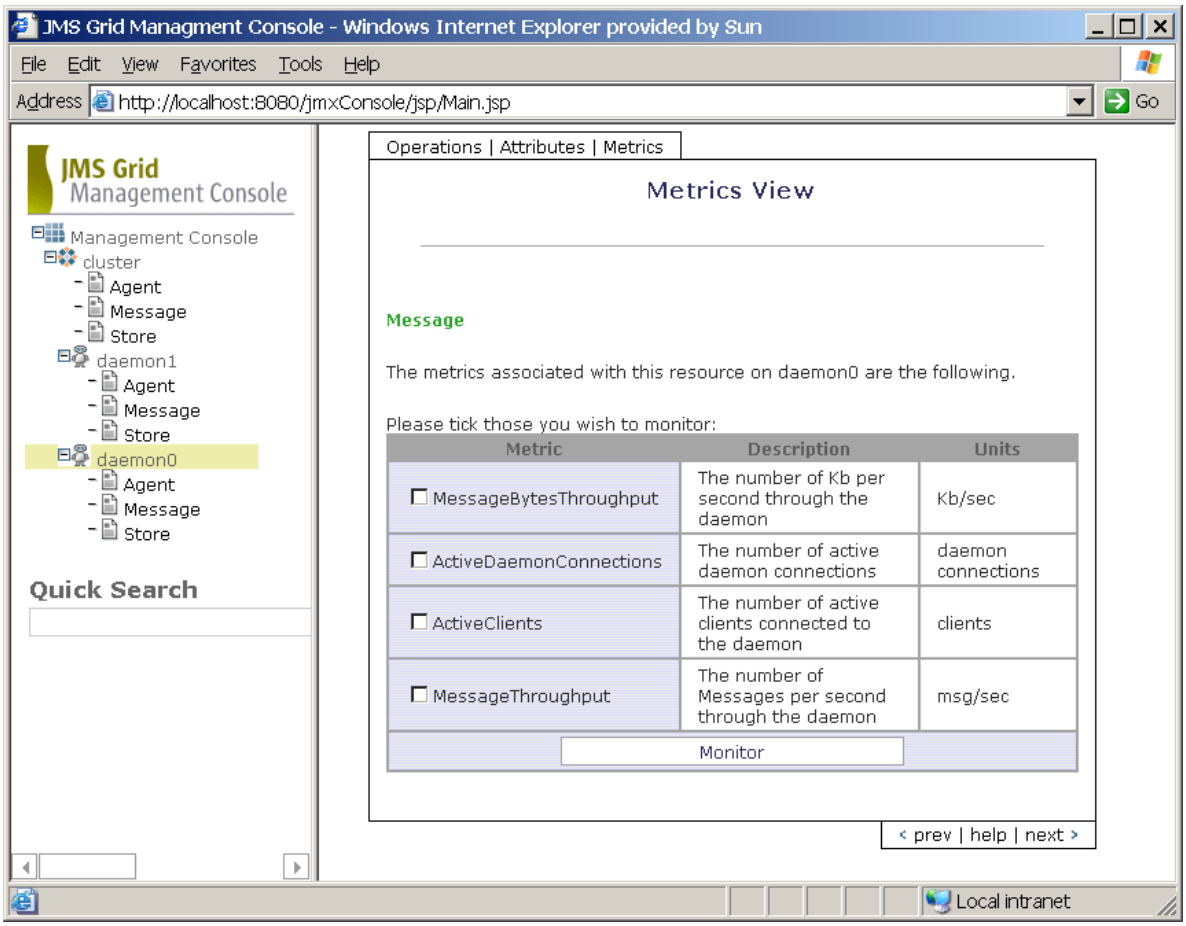
Metrics View

There are two screens in the metrics view: Select Metrics and View Metric.

Select Matrics Screen

This view allows you to select from a list those metrics should be displayed.

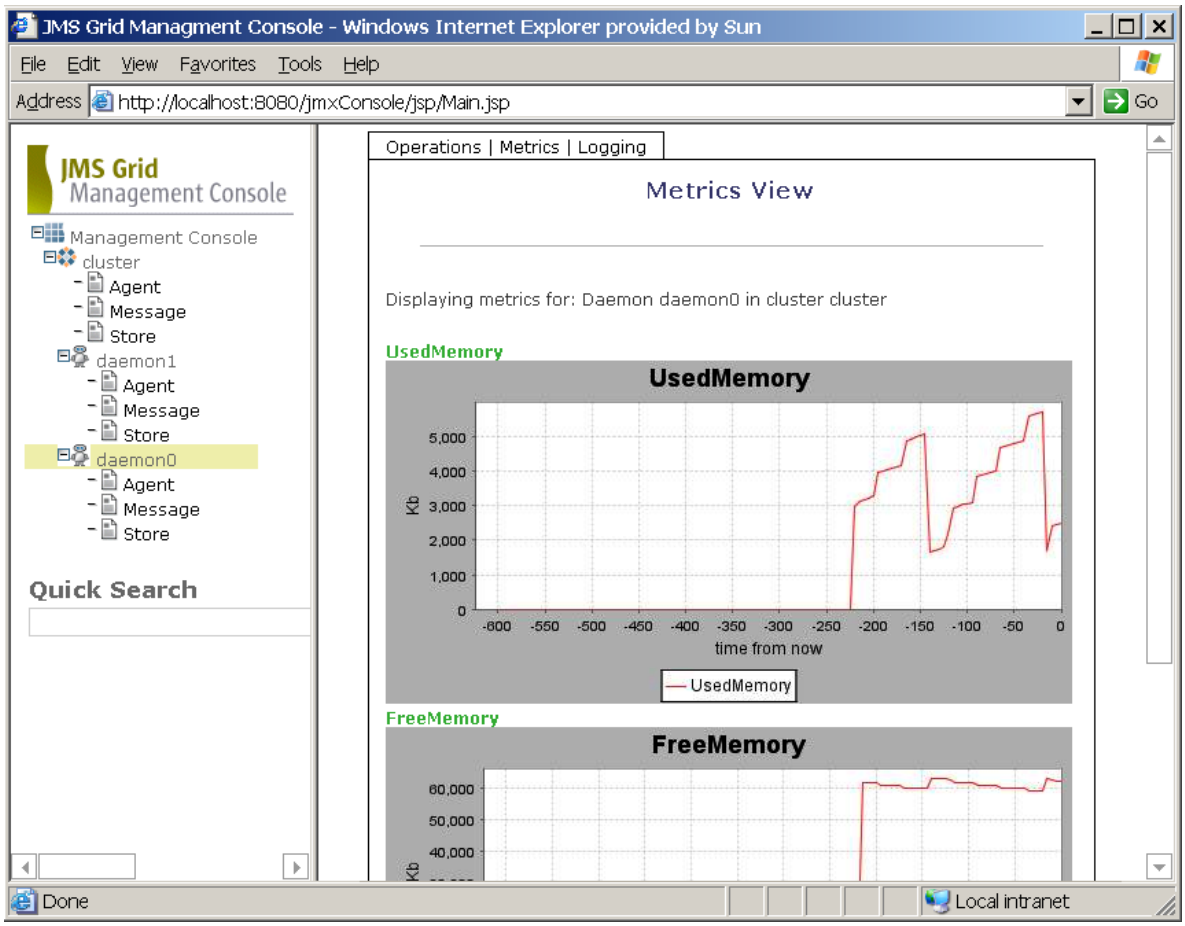
Figure 82 Typical Metric Selection Screen



View Metrics screen

The Metrics view displays the information graphically for each of the metrics you selected on the Select Metrics screen such as memory utilization.

Figure 83 Typical Metric Viewing Screen



Logging View

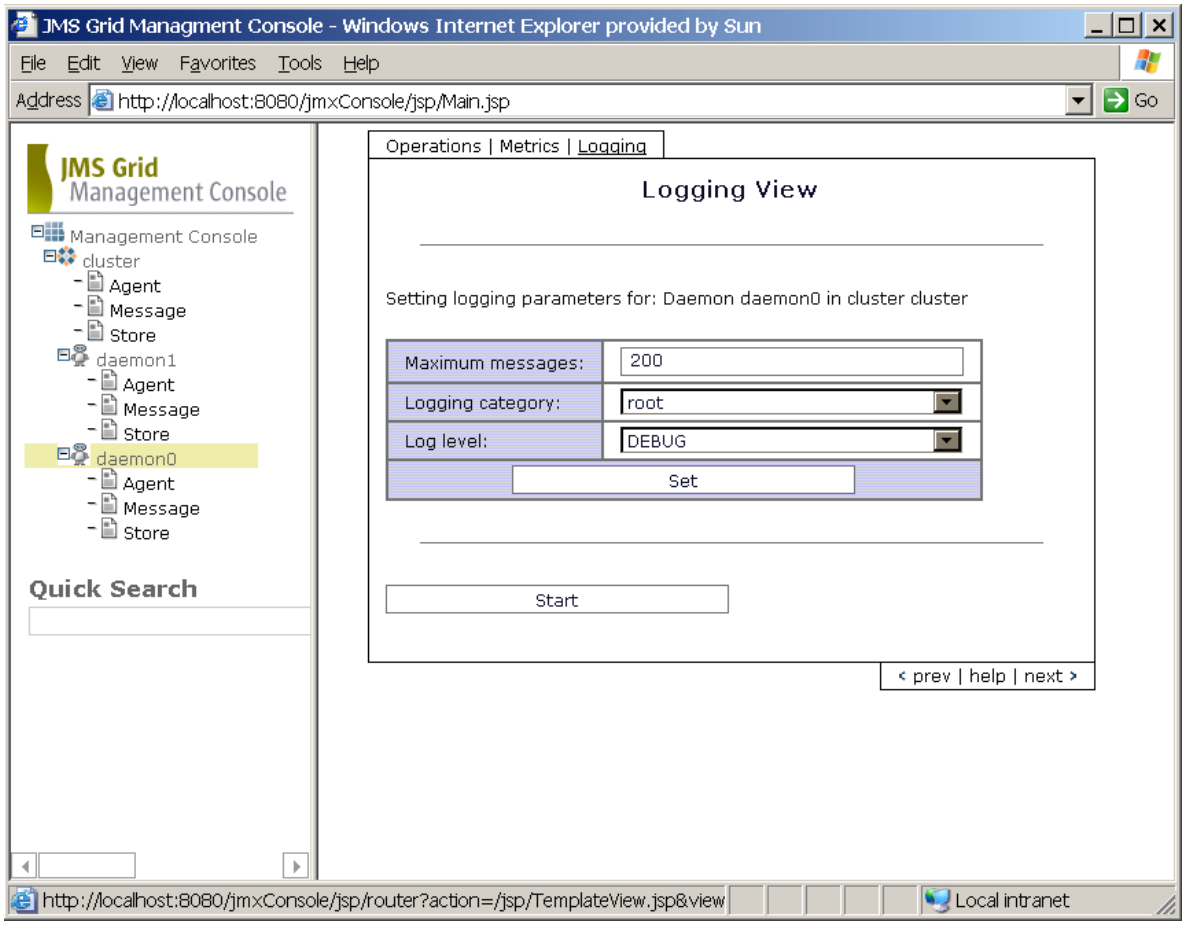
The screens available in the logging view depend on whether you are logging a cluster or a daemon. You can also set the logging parameters on the Set Logging Parameters screen before starting to log. On this screen it is also possible to set your maximum messages preference.

All daemons and clusters share a tree of logging category names, and each category has an associated priority. This screen allows you to alter the priority, or log level, associated with a particular category. The view also allows you to set the maximum number of messages preference. This determines how many messages are visible at a time in the view logging screen, in effect the number of rows in the table. This value is saved in a cookie at the client and will persist from session to session. As shown in the screen shot below this view presents you with a list of logging categories.

Selecting a category shows its current priority in the log level list. The value in the log level list can be changed. There is also a field for setting the maximum number of visible messages. The “Set“ button validates the parameters, and if valid makes these changes to the daemon you selected and acknowledges the action. If your input is

invalid you will be presented with some explanatory messages to help you resolve the problem.

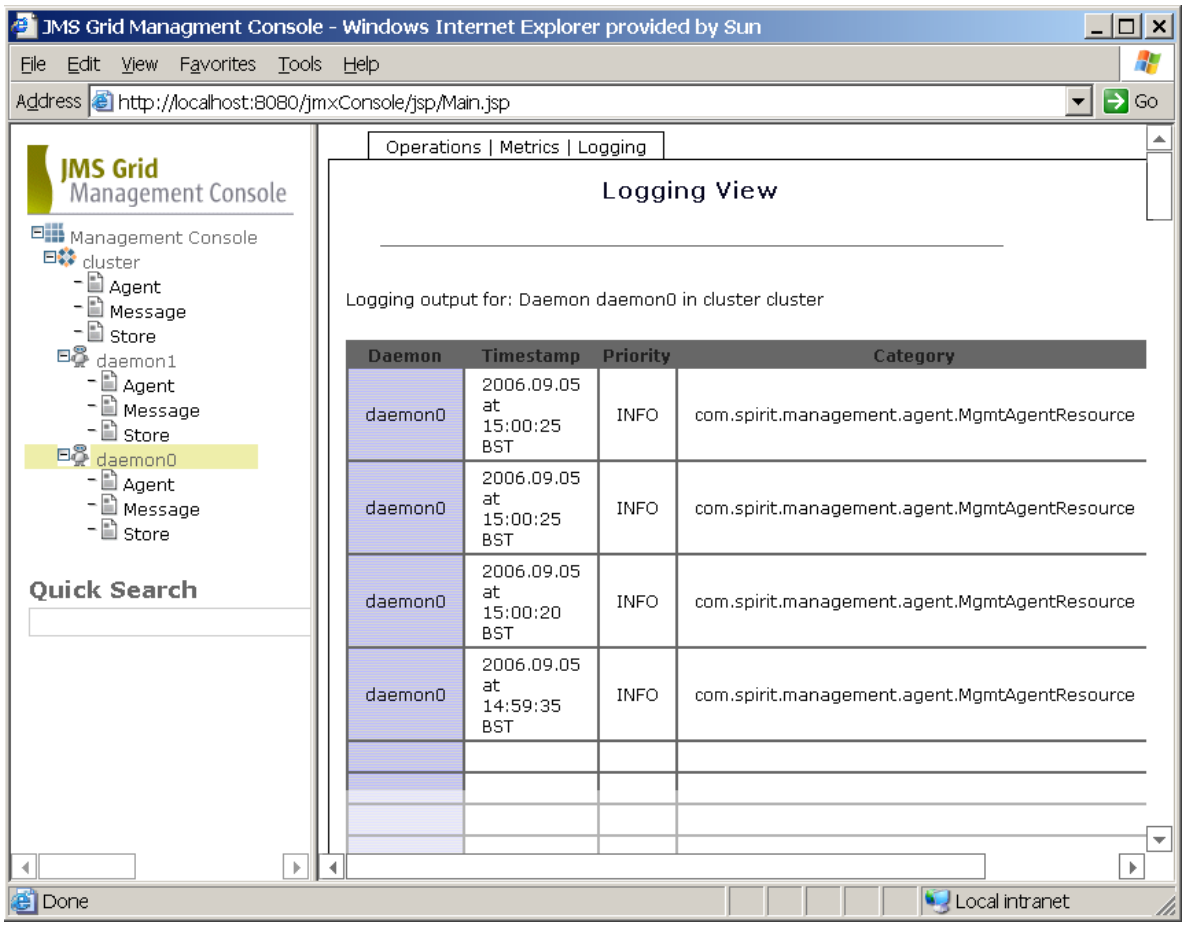
Figure 84 Setting Logging levels in the Set Logging Parameters Screen



View Logging screen

This view consists of a table, which is continuously updated, showing you the logging messages from a particular daemon. Logging is not retrospective, i.e. this view only displays log messages that arrive after the view is initiated. Messages are displayed in reverse chronological order, and by priority. Ordering by priority means that if an event generates several messages only the one(s) with the highest priority are presented. The message list will accommodate up to your preferred maximum number of messages.

Figure 85 View Logging Screen for a Cluster with no Message Activity



5.5 Management Commands

This section of the Management Guide discusses the commands that can be run from the command line rather than from the Management Console. Many of these operations are available from within the console but you may find it more convenient to run them this way.

5.5.1 Notes on Command Syntax

Note: *You describe the syntax of the commands using a notation similar to Backus Naur form. Here is a description of the symbols and what they mean:*

::= is the usual BNF notation meaning 'is expressed as'

| separates alternatives

[ ] items in square brackets are optional

* after an element means it may appear not at all, once or many times

{ } curly brackets are used to group alternatives separated by |

anon any text in bold should be entered in the command line exactly as it appears.

Apart from the last items these symbols are part of the grammar and do not form part of the command input.

Common Features

All commands when run standalone have the same template for the command line.

This is:

```
Command::= CmdOption CommandLine
CmdOption::= atsub | attr | btime | ccc | clc | filter | gc | killd |
lcc | lq | lqm |lqnm | lsub | lt | ltm | metric |qsize | qstat | rcc |
rmq | rmqm | rmtm | rmtsub | showqm | showtm | sms |substat | tsize |
tstat | uc
CommandLine::= CommandArgsOnLine | CommandArgsInFile
CommandArgsOnLine::= ConnectionSpecification ContextSpecification
[Arguments]
CommandArgsInFile::= FileSpecification
```

The following four sections discuss these command elements, giving a grammar for each. There then follows a description for each command, describing what it does and what it expects for the `Arguments` element if required.

Note: *There should be no spaces between the values in the various command segments.The examples will make this clearer.*

Connection Specification

For standalone commands the connection parameters must be specified on the command line. The connection specification is used to say which daemon you want to connect to, though it need not be the one about which you are carrying out a query. A grammar for the connection specification is:

```
ConnectionSpecification::= -connect Hotspot,UserSpecification
Hotspot::= hotspotURL | props
UserSpecification::= username,password | anon
```

The special values props and anon are:

props: specifies that the hotspot should be read from the system properties, specifically messageChannels.

anon: instead of specifying username and password this indicates that anonymous log in should be attempted.

If you want to connect to a daemon running on port 50609 on your local machine and use anonymous log in then the connection specification would appear:

```
-connect tcp://localhost:50607,anon
```

In contrast if that daemon is running on a machine with IP address 129.167.100.4 and it has security on so anonymous login won't work, a connection specification would look like:

```
-connect tcp://129.167.100.4:50607,admin,admin
```

assuming that you have a user admin with password admin.

Context Specification

This allows the User to specify the cluster or daemon on which the command is to be carried out. Where a daemon is specified its owning cluster forms the first part of the context. The context appears:

```
ContextSpecification::= -context ClusterOnly | Daemon
ClusterOnly::= cluster
Daemon::= cluster.daemon
```

In most cases the context will require a Daemon-type specification. Where this is not the case this will be pointed out.

If you have one default daemon, that is started just by running startserver with no arguments, running on a machine called philip then the context specification for it would look like:

```
-context default.philip-50607
```

The cluster for a default daemon is always 'default' and the default daemon's name is derived from the machine name plus the port number on which it runs. However, if you have a daemon named D1 forming part of a cluster C1 then its context specification would be:

```
-context C1.D1
```

Arguments

This part varies from command to command and may for some be absent entirely. It allows the user to specify arguments to a command in a comma separated list. It has the form:

```
Arguments::= -args ArgList
ArgList::= ArgListItem* ArgItem
ArgListItem::= ArgItem,
```

The precise nature of ArgItem varies from command to command and is discussed with each command in later sections.

Note about Spaces and Shell Interpretation of 'Special' Characters

In some cases you might need to specify a selector, and this will be an expression that may include embedded spaces or characters which have a special meaning for the shell you are using. Some commands use curly brackets which have a special meaning to Unix shells. In this case you can either put the particular `ArgItem` in inverted commas, or the whole of `ArgList`. This will prevent the shell and shell script used by the commands from interpreting wrongly. Here is an example. Assume you want to list messages on a topic. The command says you must supply topic name, client id, subscriber name and selector in the `ArgList`. The argument part can be written as follows:

... ... -args jmstopic,MyClientId,MySubscriberName,"JMSPriority > 2"

or as follows:

... -args "jmstopic,MyClientId,MySubscriberName,JMSPriority > 2"

FileSpecification

It is possible to provide the arguments to the command as a file. This is done:

FileSpecification::= **-file** filename

The content of the file follows the `CommandArgsOnLine` format.

5.5.2 Command Descriptions

The following sections describe each of the available commands. For simplicity the grammar for providing arguments in a file is not discussed. As mentioned above the content of the file should follow the non-file format. In most cases the `ConnectionSpecification` and `ContextSpecification` need no special comment. The `ContextSpecification` will almost always include a cluster and daemon name.

Add a Topic Subscriber - atsub

Adds a subscriber to a topic.

Grammar:

```
atsub ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args topicName,clientID,subscriberName,{selector|NONE}
```

See Values of Attributes - attr

Shows values of attributes of a daemon.

Grammar:

```
attr ConnectionSpecification ContextSpecification
```

The `Arguments` element is not required for this command.

Set Time Socket Blocked before Closing - btime

Set time socket blocked before closing.

Grammar:

```
btime ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args timeInMilliseconds
```

Create Connection - ccc

Create a cluster or network connection: incorporate daemons into a cluster or clusters into a network.

Grammar:

```
ccc ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args {ClusterSpec | NetworkSpec}
ClusterSpec ::= cluster,daemonURL
NetworkSpec ::= network,clusterName{daemonURL}
```

For a cluster connection let's assume you have two single daemons running on your local machine called philip, one on port 50607 and one on 50608. You want to make these into a cluster. You can do it like this:

```
ccc -connect tcp://localhost:50607,anon -context default.philip-50607
-args cluster, tcp://localhost:50608
```

or like this:

```
ccc -connect tcp://localhost:50608,anon -context default.philip-50608
-args cluster, tcp://localhost:50607
```

As cluster connections are bidirectional it does not matter which way around you issue the command.

When you come to network connections you have to be more careful as a network connection is unidirectional: topic messages flow only from the target cluster to the one which originates the connection. Lets say you have two clusters, A and B, with A running on a machine called preston and B on a machine called kendal. A has two daemons D1 and D2 and B has D3 and D4 and these use ports 50607 through to 50610. Assuming you want A to receive topic messages from B: this means A has to subscribe to messages from B by originating the network connection. A command to do this might appear:

```
ccc -connect tcp://preston:50607,admin,admin -context A.D1 -args
network,B{tcp://kendal:50609}
```

Note: *You connect to a daemon in A and give as arguments the cluster B and the URL of a daemon in that cluster. If on the other hand you want B to receive topic messages from A you swap things around:*

```
ccc -connect tcp://kendal:50609,admin,admin -context B.D3 -args
network,A{tcp://preston:50608}
```

See also the filter command, which can be used to restrict the topics whose messages are sent over a network connection. See also the note above about the interpretation of 'special' characters as curly brackets can cause problems on Unix.

Close clients - clc

Closes clients in various contexts.

Grammar:

```
clc ConnectionSpecification ContextSpecification [Arguments]
```

where

```
ContextSpecification ::= -context cluster[.daemon]
Arguments ::= -args clientURL
```

The following three variations of this command are available:

1. Close all clients on a cluster. For this specify only the cluster name.
2. Close all clients on a daemon. For this specify the cluster and daemon names for the daemon whose clients are to be closed.
3. Close a particular client on a daemon. For this specify the cluster and daemon name where the client is connected and the client's URL. You can find client URLs using the **lcc** command described below.

Set Network Connection Topic Filters - filter

Set network connection topic filter. When a network connection is created you specify which topics' messages are to be forwarded to the remote cluster. This command allows you to set this list.

Grammar:

```
filter ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args clusterName{commaSeparatedTopicList}
```

If you have a network connection to cluster C2 and you only want messages from topics WidgetOrders and WidgetFaults to be forwarded then the Arguments would appear:

```
-args C2{WidgetOrders,WidgetFaults}
```

See also the ccc command which describes how to create a network connection. See also the note above about the interpretation of 'special' characters as curly brackets can cause problems on Unix.

Collect Garbage - gc

Causes the garbage collector to be invoked on a daemon's virtual machine.

Grammar:

```
gc ConnectionSpecification ContextSpecification
```

The `Arguments` element is not required for this command.

Shut Down a Daemon - killd

Causes the daemon to shut down immediately. Be aware that the immediate shut down prevents all the daemon threads from terminating normally. It is better to use the **sms** command to leave the daemon in a consistent state.

Grammar:

```
killd ConnectionSpecification ContextSpecification
```

The `Arguments` element is not required for this command.

List Connected Clients - lcc

List clients connected to a daemon.

Grammar:

```
lcc ConnectionSpecification ContextSpecification
```

The `Arguments` element is not required for this command.

List all Queues - lq

Lists all queues available on a daemon.

Grammar:

```
lq ConnectionSpecification ContextSpecification
```

The `Arguments` element is not required for this command.

List queue messages - lqm

Lists undelivered messages on a queue.

Grammar:

```
lqm ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args queueName,{selector | NONE}
```

List all Subscribers - lsub

Lists all durable subscribers.

Grammar:

```
lsub ConnectionSpecification ContextSpecification
```

The `Arguments` element is not required for this command.

List all Topics - lt

This command shows all topics which have at least one durable subscription.

Grammar:

```
lt ConnectionSpecification ContextSpecification
```

The `Arguments` element is not required for this command.

List Topic Messages - ltm

Lists the undelivered messages on a topic for a subscriber meeting criteria set in the selector.

Grammar:

```
ltm ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args topicName,clientID,subscriberName,{selector|NONE}
```

Values for client ID and subscriber name can be obtained using the **lsub** command.

Display Values of a Metric - metric

Shows a continuously updating series of values for a metric. This terminates when the user presses control-C.

Grammar:

```
metric ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args {resourceName,metricName | LIST}
```

All metrics are associated with a resource, whose possible values are WaveMessageDaemon, MessageStore and MessageCore. The LIST option will give a list of the resources and their associated metrics.

Show Queue Size - qsize

Shows how many undelivered messages there are on a queue, or on all known queues.

Grammar:

```
qsize ConnectionSpecification ContextSpecification [Arguments]
```

where

```
Arguments ::= -args queueName
```

If no Arguments are specified the output gives the number of messages unconsumed on all queues, otherwise only on the queue named in `Arguments`.

Show Statistics about Queues - qstat

This command shows statistics about a queue or all queues. The statistics shown are message count, timestamp of the newest and oldest unconsumed message on the queue, and the number of receivers.

Grammar:

```
qstat ConnectionSpecification ContextSpecification [Arguments]
```

where

```
Arguments ::= -args queueName
```

Reconnect Clients - rcc

Reconnects clients in various contexts.

Grammar:

```
rcc ConnectionSpecification ContextSpecification [Arguments]
```

where

```
ContextSpecification ::= -context cluster[.daemon]
Arguments ::= -args [ClientSpecification,]destinationURL
ClientSpecification ::= clientID{clientURL}
```

The following three variations of this command are available:

1. Reconnect all clients on a cluster to one of its daemons. For this specify the cluster name in the context and the destination URL in arguments. The destination URL is the URL of the daemon in the cluster to which the clients should reconnect. An example might appear:

```
rcc -connect tcp://localhost:50607,admin,admin -context C1 -args
tcp://localhost:50609
```

2. Reconnect all clients from one daemon in a cluster to an alternative one in the same cluster. For this specify the cluster and daemon names in the context and the destination URL in arguments. The destination URL is the URL of the daemon in the cluster to which the clients should reconnect. An example might appear:

```
rcc -connect tcp://localhost:50607,admin,admin -context C1.D1 -args
tcp://localhost:50609
```

3. Reconnect a particular client from one daemon to another. For this specify the cluster and daemon names in the context the client's ID and URL and the destination URL in arguments. The destination URL is the URL of the daemon in the cluster to which the client should reconnect. An example might appear:

```
rcc -connect tcp://localhost:50607,admin,admin -context C1.D1 -args
CID{tcp://localhost:3520},tcp://localhost:50609
```

See also the note above about the interpretation of 'special' characters as curly brackets can cause problems on Unix.

Remove a Queue - rmq

Removes a queue.

Grammar:

```
rmq ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args queueName
```

Remove a Queue Message - rmqm

Removes a message from a queue.

Grammar:

```
rmqm ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args queueName,messageID
```

Values for message ID can be obtained using the **lqm** command.

Remove a Topic Message - rmtm

Removes a message from a topic or subscription.

Grammar:

```
rmtm ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args {RemoveFromTopicArgs | RemoveFromSubArgs}
RemoveFromTopicArgs ::= topicName,messageID
RemoveFromSubArgs::=
topicName,clientID,subscriberName,{selector|NONE},messageID
```

If just a topic name and message identifier are supplied then that message is removed from all subscriptions which might receive it. If all the arguments are supplied then it is only removed from that subscription. Values for clientID, subscriberName, selector and messageID can be obtained using the **ltm** and **lsub** commands.

Remove a Topic Subscriber - rmtsub

Removes a subscriber from a topic.

Grammar:

```
rmtsub ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args topicName,clientID,subscriberName
```

Values for client ID and subscriber name can be obtained using the lsub command.

Display a Message from a Queue - showqm

This command shows some properties of a queue message and also its body. If the message is encrypted, the body is not shown.

Grammar:

```
showqm ConnectionSpecification ContextSpecification Arguments
```

where

```
Arguments ::= -args queueName,messageID
```

The values for `messageID` can be obtained using the lqm command.

Display a message from a topic - showtm

This command shows some properties of a topic message and also its body. If the message is encrypted, the body is not shown.

Grammar:

showtm *ConnectionSpecification ContextSpecification Arguments*

where

```
Arguments ::= -args topicName,messageID
```

The values for messageID can be obtained using the **ltm** command.

Shutdown Message Server - sms

This command causes the daemon to shut down normally.

Grammar:

```
sms ConnectionSpecification ContextSpecification [Arguments]
```

where

```
Arguments ::= -args timeBeforeShutdownInSeconds
```

Show Statistics about Subscriptions - substat

This command shows statistics about a subscription or all subscriptions. The statistics shown are message count, and the timestamp of the newest and oldest unconsumed message on the subscription.

Grammar:

```
substat ConnectionSpecification ContextSpecification [Arguments]
```

where

```
Arguments ::= -args topicName,clientID,subscriberName,{selector|NONE}
```

Show Number of Unconsumed Messages on a Topic - tsize

Shows how many unconsumed messages there are on topics. There are three variations on this command as described below.

Grammar:

```
tsize ConnectionSpecification ContextSpecification [Arguments]
```

where

```
Arguments ::= -args {TopicSpecification | SubscriptionSpecification}
TopicSpecification ::= topicName
SubscriptionSpecification ::=
topicName,clientID,subscriberName,{selector|NONE}
```

The three variants on this command do the following:

1. No Arguments: returns the number of unconsumed messages on all topics. This value will reflect the maximum number of unconsumed messages over all subscribers to each topic.

2 Named topic: returns the number of unconsumed messages on that topic. This value will reflect the maximum number of unconsumed messages over all subscribers to the topic.

3 All arguments supplied: shows the number of unconsumed messages on a topic for a particular subscriber. Values for client ID and subscriber name can be obtained using the lsub command.

Show Statistics about Topics - tstat

This command shows statistics about a topic or all topics. The statistics shown are message count, timestamp of the newest and oldest unconsumed message on the topic and the number of subscribers.

Grammar:

```
tstat ConnectionSpecification ContextSpecification [Arguments]
```

Where

```
Arguments ::= -args topicName
```

Update Configuration - uc

Reloads a cluster's configuration from Admin Store.

Grammar:

```
uc ConnectionSpecification ContextSpecification
```

where

```
ContextSpecification ::= -context cluster
```

The `Arguments` element is not required for this command.

5.6 The Management Model

The operations and attributes (methods and properties) of the MBeans used to manage the JMS Grid Message Daemon are described in the tables in the following sections.

There are two ways in which operations can be performed and attributes set. First of all, one can use the various `invoke()` and `setAttribute[s]()` methods on the MBeans.

These are completely generic and are mandated by the JMX specification. As an example, if you want to invoke the `retrieveQueueSize` operation on a queue called '`jmsqueue`' you can call `invoke()` on the Store bean passing the string argument '`retrieveQueueSize(jmsqueue)`'.

The JMS Grid JMX API has also added Java helper classes - representing the model directly. These classes allow you to call a method in what might seem a more natural way. If you want to call `retrieveQueueSize` then this is available as a method in its own right on the helper class, in this case `MessageStoreResource`. These classes are described in more detail in the browsable Javadocs for the JMS Grid Management API.

5.6.1 WaveMessageDaemon Resource

The WaveMessageDaemon resource provides access to agent services, some metrics and to logging of a particular daemon.

N.B. The WaveMessageDaemon Resource is still labeled Agent in the Management Console.

Operations

The following operations are available for the WaveMessageDaemon resource.

Table 44 Operations for WaveMessageDaemon Resource

Operations	Parameter	Description
`collectGarbage`	None	Request JVM collects garbage from memory.
`shutdownNow`	None	Quick shutdown of the application.
`getLogPriority`	String {CategoryName}	Get the log priority of a single Category.
`setLogPriority`	String{Category}, String {priority}	Set log priority on a single category.
`getLogPriorities`	None	Get the log priorities of all categories.
`retrieveActiveThreads`	None	Get the active threads.

Metrics

The following metrics can be accessed from the WaveMessageDaemon resource.

Table 45 Metrics from WaveMessageDaemon Resource

Metrics	Description
`ActiveThreads`	The number of threads currently running.
`FreeMemory`	The current free memory available to the JVM.
`UsedMemory`	The memory currently used in the JVM.

5.6.2 MessageCore Resource

The MessageCore resource is the component of the JMS Grid daemon that handles cluster formation, client and network connections, and message routing.

Attributes

The Boolean attributes shown, as Bool in the table, are Boolean primitives and can take values of true or false. All Attributes are read/write.

Table 46 Boolean Attributes

Attribute	Type	Description
autoDiscoveryAllowed	Bool	Determines whether the daemon will register as a service for clients to automatically discover, and whether the daemon itself will use multicast discovery to locate other daemons.
connectionLoadBalancing	String	The load balance strategy - either RandomLoading or LeastUsedLoading.
daemonConnectionRetriesTimeout	int	The time to wait before trying to reconnect to a peer in a cluster/network.
maxDaemonConnectionRetries	int	The maximum attempts to connect to a peer in a cluster/network.
maxInternalQueueSize	int	The maximum size of all transient queues in the daemon (in bytes).
maxTopicDispatchQueues	int	The maximum number of dispatch queues for transient Topic messages.
pingEnabled	Bool	Enable keep alive protocol for Sockets.
pingTimeout	int	Time between sending a keep alive packet down a socket (in ms).
networkConnectionQueueFilters	String	Queue filter between networked clusters.
networkConnectionTopicFilters	String	Topic filter between networked clusters.
startTime	long	The time the daemon was started.
timeSocketBlockedBeforeClosing	int	max time in milliseconds that a client is allowed to be blocked receiving traffic before being closed.

Operations

The MessageCore resource supports the following operations, which unless otherwise stated have a void return type:

Table 47 Message Core Resource Operations

Operations	Parameter	Description
createClusterConnection	String {daemonURL}	Create a Cluster Connection to the specified URL.
createNetworkConnection	String{networkURL} networkURL := clusterName{daemonURL}	Create a Network Connection to the specified cluster.
getConnectedClients	None	Return a java.util.Vector of connected clients.
closeClient	String{clientID}	Close a client connected to the message daemon by it's clientID.

Operations	Parameter	Description (Continued)
closeAllClients	None	Close all clients attached to the message daemon.
reconnectClient	String {clientID} String {daemonURL}	Reconnect a client to a another message daemon.
reconnectClientByDaemonName	String {clientID} String {daemonName}	reconnect a client to a another message daemon
reconnectAllClients	String{daemonURL}	Reconnect all clients to a another message daemon.
reconnectAllClientsByDaemonName	String{daemonURL}	Reconnect all clients to another message daemon.
shutdownMessageDaemon	int{timeBeforeShutdown in seconds}	Shutdown the message server.
updateConfiguration	None	Update the daemon's configuration from the configuration store.

Metrics

The following metrics can be accessed from the MessageCore resource.

Table 48 Metrics from the MessageCore Resource

Metrics	Description
ActiveClients	The number of clients connected to the daemon.
ActiveDaemonConnections	The number of active daemon connections connected to the daemon.
MessageBytesThroughput	The number of Kbytes per second through the daemon.
MessageThroughput	The number of messages per second through the daemon.

5.7 MessageStore Resource

The MessageStore resource manages the SpiritDB persistence component for the JMS Grid Message daemon.

Operations

The MessageStore resource supports the following operations, which unless otherwise stated have a void return type:

Table 49 MessageStore Resource Operations

Operations	Parameter	Description
`retrieveAllQueues`	None	Retrieve all durable Queues from the Message Store - returns a java.util.Vector of Queue names.
`retrieveAllSubscribers`	None	Retrieves a java.util.Vector of durable Topic subscribers.
`removeQueue`	String{queueName}	Removes a Queue.
`removeTopicSubscriber`	String{Topic Name}, String{clientID}, String{subscriberName}	Remove a durable Topic Subscriber.
`addTopicSubscriber`	String{Topic Name}, String{clientID}, String{subscriberName}, String{selector}	Add a durable Topic Subscriber.
`retrieveAllTopicMessages`	String{Topic Name}, String{clientID}, String{subscriberName}, String{filter to apply to returned messages}	Retrieve all Topic Messages from a subscriber.
retrieveAllQueueMessages	String{Queue Name}, String{filter to apply to returned messages}	Retrieve all Messages for a named queue.
retrieveSubscriberSize	String{Topic Name}, String{clientID}, String{subscriberName}, String{filter to apply to returned messages}	Return the number of unconsumed messages for a subscriber.
retrieveQueueSize	String{queueName}	Return the number of unconsumed messages for a Queue.
retrieveQueueHeadAge	String (queueName)	Return the age in milliseconds of the oldest message in the queue.
retrieveQueueTailAge	String (queueName)	Return the age in milliseconds of the newest message in the queue.
retrieveTopicSize	String{Topic Name}	Return the number of unconsumed messages for a Topic.
removeMessagesFromQueue	String{Queue Name}, String{messageIDs}	Remove message(s) from a Queue. A string comma separated list can be passed as messageIDs.

Operations	Parameter	Description (Continued)
removeMessagesFromTopic	String{Topic Name}, String{clientID}, String{selector}, String{messageIDs}	Remove message(s) from a Topic Subscriber. A string comma separated list can be passed as messageIDs.

5.7.1 Metrics

The following metrics can be accessed from the MessageStore resource:

Table 50 Metrics from the MessageStore Resource

Metrics	Description
`MessageStore`	The current size of the message store in MBytes
`SD_<clientID/SubscriberName>`	Subscriber depth for a durable Topic subscriber - identified by clientID and subscriberName
`QD_<QueueName>`	Queue depth for a named queue

5.8 Example JMX Program

This section will develop a complete example using the JMS Grid JMX classes to illustrate how you can make use of them in your own organization. You should have a basic familiarity with the concepts explained in the JMX Management Guide chapter 2 and the JMX architecture described in Chapter 3.

The program you will develop monitors the depth of a topic and should it reach a pre-defined limit will send an email to an administrator. You might want to do this for example where the fact that a topic is becoming over long could indicate a problem with consumers or the network.

5.8.1 A Note about Documentation

The javadocs for all the management classes, some of which are used in this example, can be found in your wave installation in docs/javadoc. Management classes are in the package tree starting at com.spirit.management.

5.8.2 Making a Connection

Before you can do anything with JMX you need to connect to the network of management agents. You do this using the WaveManager class (from com.spirit.management.wave):

```
WaveManager waveManager = new WaveManager();
waveManager.init(hotspot, username, password);
```

This code snippet creates a WaveManager instance and then uses it to connect to the management infrastructure. The three arguments you have here are:

Table 51 Three Arguments

Arguments	Description
`hotspot`	The URL of any daemon in your JMS Grid network. It will look something like: `tcp://localhost:50607`
`username`	The administrator's username
`password`	The administrator's password

5.8.3 Finding the Agent

Once you have successfully completed the previous step you have logged in to the management infrastructure. In order to do anything useful you need to interact with an MBean which manages the resource you are interested in.

As explained in the JMX Management Guide each daemon has an MBeanServer which holds the registry of MBeans. Each MBeanServer has an agent which allows applications to contact it from "outside" and make use of its MBeans. The next step in the process is therefore to make contact with the agent so you can find the right MBean.

In JMS Grid JMX you use proxies to communicate with the agents. There are DaemonClusterProxies and DaemonAgentProxies. The latter provide access to the MBeans on one daemon, the former are a sort of "directory" of proxies available in a cluster. They can also provide some information about the cluster as a whole.

This is how you connect to the DaemonAgentProxy, with error handling omitted:

```
String clusterName = "cluster1";
String daemonName = "daemon1";
DaemonClusterProxy dcp =
waveManager.getDaemonClusterProxy(clusterName);
DaemonAgentProxy dap = dcp.getDaemonAgentProxy(daemonName);
```

As topics are replicated across all daemons in a cluster it does not really matter which daemon agent you use to find the topic depth. You have just chosen an arbitrary daemon in your cluster.

5.8.4 Finding the Manageable Resource

Now you have the DaemonAgentProxy you can interact with the MBean which can tell you about topic depth. As explained in Chapter 7 of the JMX Management Guide there are two ways to interact with MBeans: using the generic `invoke()` and `setAttributes()` or by using JMS Grid JMX helper classes. You will use the helper classes in this example.

There are three manageable resources in a daemon: store, core and the agent itself. The store has information about topic depths. You therefore need to use the helper class which wraps interactions with the store MBean and allows you to manage the store resource:

```
MessageStoreResource resource = dap.getMessageStoreResource();
```

5.8.5 Finding the Topic Size

The method on MessageStoreResource which tells you about topic depth is `retrieveSubscriberSize()`. As its name suggests it actually tells you how many messages remain unconsumed for a particular consumer. Therefore to use it you need to supply information about the particular subscriber:

```
String topicName = "jmstopic";
String clientID = "Durable";
String subName = "Test";
String selector = "";
Integer noMessages = resource.retrieveSubscriberSize(
topicName, clientID, subName, selector);
```

5.8.6 Comparison: using MBean `invoke()`

In the interest of comparison the next code snippet shows how you would use the MBean `invoke()` method without using the resource helper class:

```
// as before...
DaemonAgentProxy dap = dcp.getDaemonAgentProxy(daemonName);
Integer noMessages = (Integer)dap.invoke(
"retrieveSubscriberSize(jmstopic,Durable,Test,)");
```

5.8.7 Complete Example

The following section shows the complete code for the topic depth monitor:

```
import java.util.*;
import java.text.*;
import java.io.IOException;
import javax.mail.*;
import javax.management.*;
import com.spirit.management.shell.*;
import com.spirit.management.wave.*;
/**
 * Example program showing how to monitor a topic and send an
 * email if it reaches a defined size.
 */
public class TopicMonitor extends TimerTask
{
    /**
     * these are all parameters needed for making a connection,
     * authentication and describing the subscription to be
     * monitored, etc. These could be given as command line
     * arguments or read from a property file
     */

    // connection
    private final String hotspot = "tcp://localhost:50607";
    private final String username = "admin";
    private final String password = "admin";

    // context
    private final String clusterName = "cluster1";
    private final String daemonName = "daemon1";
```

```
    // subscription - in the case of the queue you would need
    // the queue name and selector instead
    private final String topicName = "jmstopic";
    private final String clientID = "Durable";
    private final String subName = "Test";
    private final String selector = "";

    // limit of number of message
    private final int limit = 20;

    // mail properties
 private final String originator = "topicwatch@someorg.com";
 private final String recipient[] = {"administrator@someorg.com"};

    // obviously, can make other parts of message configurable
private final String messageBase = "Topic jmstopic for subscriber
Test reached size limit \"{0}\" at \"{1}\"";

private final String messageTitle = "Subscriber \"{0}\" reached size
limit";
private final DateFormat dateFormat = new
SimpleDateFormat("HH:mm:ss dd/MM/yy");

    // interval between monitoring requests - 10 s
    private final long interval = 10000;

    private Timer timer;
    private MessageStoreResource resource;

    /**
     * sets up connections etc. before the timer starts to do
     * periodic monitoring
     */
    public TopicMonitor() throws JMException
    {
        WaveManager waveManager = new WaveManager();
        waveManager.init(hotspot, username, password);

DaemonClusterProxy dcp =
waveManager.getDaemonClusterProxy(clusterName);
        if (dcp == null)
        {
System.out.println("TopicMonitor: unrecognised cluster " +
clusterName);
throw new JMException();
        }

DaemonAgentProxy dap = dcp.getDaemonAgentProxy(daemonName);
        if (dap == null)
        {
System.out.println("TopicMonitor: unrecognised daemon " +
daemonName);
throw new JMException();
        }

        resource = dap.getMessageStoreResource();
    }
    /**
     * creates the timer and sets self to be run periodically
     */
    public void start()
    {
        timer = new Timer();
        timer.schedule(this, 0, interval);
```

```
    }
    /**
     * does the work of the command - checks the topic size and
     * emails if too large
     */
    public void run()
    {
        try
        {
Integer noMessages = resource.retrieveSubscriberSize(
topicName, clientID, subName, selector);
if (noMessages.intValue() > limit)
            {
Object[] args1 = {subName};
String title = MessageFormat.format(messageTitle,
args1);

                String msgDate = dateFormat.format(new Date());
                Object[] args2 = {new Integer(limit), msgDate};
String message = MessageFormat.format(messageBase,
args2);

    // com.spirit.management.shell.ShellUtil
                ShellUtil.sendEmail(originator,
                          recipient,
                          title,
                          message);

    // comment out this line if you want monitoring to go
    // on and an email sent every time until the topic size
    // goes down.
                cancel();
            }
        }
        catch (JMException jme)
        {
System.out.println(
"TopicMonitor: Error with connection to JMX, exiting...");
cancel();
        }
        catch (MessagingException me)
        {
System.out.println(
"TopicMonitor: Error with email connection, exiting..." + me);
cancel();
        }
        catch (IOException ioe)
        {
System.out.println(
"TopicMonitor: Error with input/output, exiting..." + ioe);
cancel();
        }
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        try
        {
            Properties props = System.getProperties();
            props.setProperty("mail.smtp.host", "sean");
            TopicMonitor tm = new TopicMonitor();
            tm.start();
```

```
            }
            catch (JMException jme)
            {
System.out.println(
"TopicMonitor: unable to set up connection");
            }
        }
```

Chapter 6

Configuration and Tuning

This chapter provides an overview of the configuration mechanism of JMS Grid and how to update configurations of running daemons.

A brief overview of some of the internals of JMS Grid is presented and the use of different configuration parameters to improve performance is explained.

6.1 Configuration Overview

6.1.1 The Properties Directory

The properties directory contains the override configuration file - `overrideprops.cfg` - (see usage below), and example certificate and Log4J properties files. If using the licensed product, then this is where the `licence.properties` file should be placed. This directory will also contain a `jndi.properties` file, which is created by the Administration tool, once the JNDI repository has been selected.

6.1.2 The Working Directory Structure

A JMS Grid Message Daemon requires a directory structure for persistent message stores; log files, configuration files and temporary data files. If the directory structure does not exist, the Message Daemon will automatically try to create one.

The working directory structure looks this:

WDIR/

conf/- snapshots of running Message Daemon configurations

data/- persistent message store(s)

logs/- running Message Daemon file logs

6.1.3 How Configuration Works

The JMS Grid Message Daemon uses an internal configuration bean for all its internal configuration information. The configuration can be initialized either from a JNDI

repository, such LDAP or the Directory Service that is part of the JMS Grid product or from a properties file.

When the JMS Grid Message Daemon is started, one of the command line parameters it can take is its name that must be unique. If the name is presented on the command line, then it is assumed that the JMS Grid Message Daemon must retrieve its configuration information from JNDI. The location of the JNDI repository is defined in the jndi.properties file that is located in the properties directory in the JMS Grid installation.

As an alternative, a configuration properties file can be used to initialize the Message Daemon. If the name of the daemon is not set on the command line, then only the properties file will be used. If the `name` property is not set in the configuration file, then a unique one is automatically generated.

The Message Daemon saves its working configuration in the conf/ directory under the working directory. If no configuration is supplied, or the configuration information cannot be retrieved from JNDI at start-up, the last property values saved to this configuration file are used.

6.1.4 Locally Overriding the Configuration

If a JMS Grid Message Daemon is initialized to use JNDI, and is supplied with a properties configuration file on the command line, then the properties set in the latter have precedence on the properties retrieved from JNDI. This can be useful for locally changing the configuration of a daemon. By default, a Message Daemon is initialized to retrieve override properties from the `overrideprops.cfg` file in the installation properties directory. This file contains a subset of all the possible configuration properties, which are commented out.

6.1.5 Dynamically Changing the Configuration for a Running Daemon

You can dynamically change the configuration of a running daemon in a number of ways:

- By using the JMX Management Console (see the *JMX Management* chapter)
- By the management command line scripts (see the *JMX Management* chapter)
- By JMX directly using a 3rd party tool
- Through changing the JNDI configuration store (e.g. by using the Administration Console - see the JMS Grid Administration Chapter)
- By changing the `overrideprops.cfg`
- By changing the cached configuration in the **conf/** directory

JMS Grid uses the open source JMX product MX4J, so any management tools compatible with MX4J should be able to interact with running Message Daemons.

Apart from changes through JMX, all other dynamic updates rely on polling (i.e. configuration files or JNDI).

The configuration property `doConfigurationPolling` should be set to *false* (the default is true) to prevent polling. By default, the Message Daemon will poll its configuration for updates every 30 seconds. This timeout is also configurable - the property is called `configurationPollingTimeoutInSeconds`.

When a change occurs through JMX or locally through the cached configuration or `overrideprops.cfg` file, then these changes will be reflected back into the JNDI store, if it is being used. If you require keeping these changes local to the running Message Daemon (for debugging applications, for example), then you need to set the configuration property `keepConfigurationUpdatesLocal` to *true*.

Configuration updates are also propagated to clients where applicable (such as compression levels, network timeouts, logging levels etc). You can prevent configuration changes being propagated to clients by setting the property `overrideClientConfiguration` to *false*.

6.2 Configuring JMS Grid for Fast Throughput

This chapter explains some of the configuration parameters that can be changed when performance tuning JMS Grid for your applications.

6.2.1 Message Delivery Overheads

Timestamp Computation

There is a small overhead in computing the timestamp for a message and in the space taken in the on-the-wire format for transporting this value. You can disable the computation of message timestamps by calling the method `setDisableMessageTimestamp(false)` on the `javax.jms.MessageProducer`

Checking for durable subscribers

If you are sending messages to a topic and are not using durable subscribers then JMS Grid will not persist the message. However JMS Grid will nevertheless bear a small overhead for every message because of the need to check whether there are any valid durable subscribers. You can avoid this overhead by setting the message to be non-persistent. This is done by calling the method `setDeliveryMode(DeliveryMode.NON_PERSISTENT)` on the `javax.jms.MessageProducer`.

Note: *The method set`DeliveryMode()` on the `javax.jms.Message` is for internal use only, and is overridden by the delivery mode of the `javax.jms.MessageProducer`.*

6.2.2 Message Listeners

JMS Grid uses a push-based model for message delivery. This can result in very fast throughout, where messages are delivered ahead of time and cached on the client. Internally a JMS Grid JMS client caches messages into a high-water queue, which is limited by the amount of memory used to hold the messages. Calling using a `receive()` on a `javax.jms.MessageConsumer` consumes messages from the internal high-water queue.

The overhead of using the high-water queue inside the JMS Grid client is very small. However, using a `receive()` is slower than using an asynchronous `javax.jms.MessageListener`, because of the context switch involved between the thread consuming messages from the Message Daemon and the thread calling the `receive()`. Providing the application using the Message Consumer is fast enough, it is recommended that for fast throughput a Message Listener be used.

6.2.3 Asynchronous Message Dispatch From the Daemon

By default, transient messages (i.e. non-persistent messages, or messages dispatched to non-durable consumers) are dispatched as fast as possible to message consumers. Messages received into the Message Daemon are routed and dispatched using the same thread. There are pros and cons to this approach:

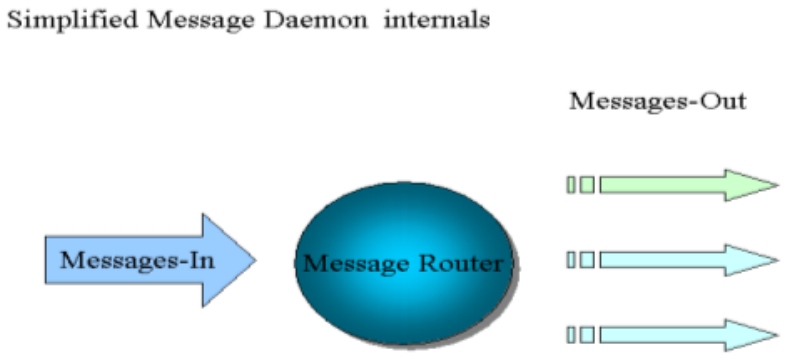
Pros:

- Latency is extremely small
- Low message overhead

Cons:

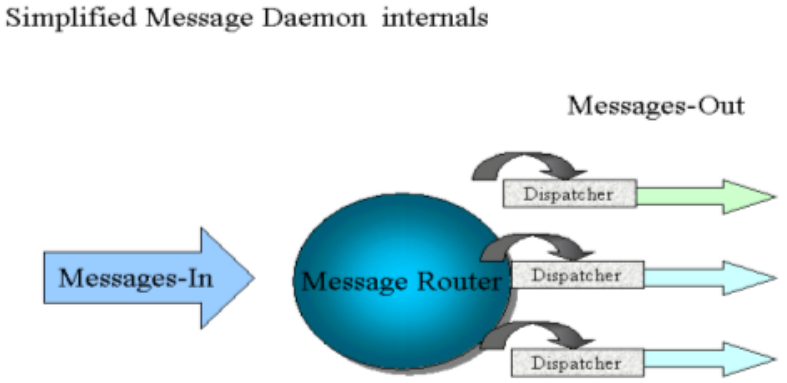
- Message throughput is driven by the slowest consumer (not asynchronous)
- Not scalable

Figure 86 Asynchronous Message - Default



In order to ensure asynchronous dispatch, the Message Daemon can be configured to use internal dispatch queues for transient messages. Each dispatch queue has a separate thread.

Figure 87 Asynchronous Dispatch Queues



To configure the Message Daemon with asynchronous dispatch queues for Publish/Subscribe, set the configuration property `maxTopicDispatchQueues` to the number of dispatch queues required. Consumers are load balanced internally across these dispatch queues, so that a single dispatch queue can service more than one message consumer.

6.2.4 Asynchronous Client Message Dispatch

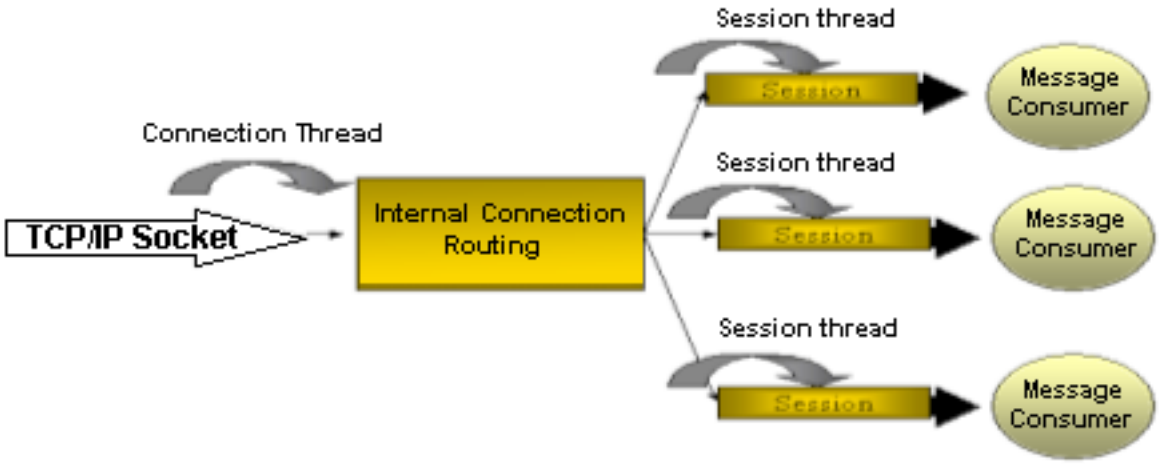
When using a JMS Grid JMS Client to connect to the Message Daemon across a slow network (such as a WAN), it can be beneficial to asynchronously dispatch published messages from the client. The client-side property to enable this is: `produceDispatchThread` (default value is *false*).

For a single JMS Client that is servicing many message consumers, an option is available on the client-side to use a separate internal queue for dispatching into the consuming sessions. This option is `consumeDispatchThread` (default value is *false*).

Both these options can increase latency and decrease throughput for normally operating clients that are using the Message Daemon on the LAN, so should be used with caution.

By default, on the client there is one thread per connection that services the TCP/IP socket connection to the daemon to consume messages. Messages are then dispatched to interested Sessions associated with the Connection. Each Session will have its own internal dispatch queue and associated thread, as depicted in the following figure.

Figure 88 Threads Used Within a JMS Grid JMS Client



6.2.5 Thread Priorities

It is important to understand the threading priorities used by a JMS Grid. The running priority of the internal threads used increases down stream of the publisher. The table below describes the different priorities and their default values.

Table 52 Priority Defaults

Parameter Name	Description	Default Value
`socketReceiverPriority`	The priority of the thread used to receive messages for both the JMS Connection and the Message Daemon	5
`clientSessionThreadPriority`	The priority of the JMS Session thread	6
`receiverThreadPriority`	The expected priority of application threads calling receive() on `MessageConsumers` N.B. JMS Grid will try and temporarily set this priority on the application thread, if the application thread is of a lower priority	7
`daemonDispatchQueuePriority`	The priority of the threads in the daemon for transient message dispatch	6
`daemonPersistentDispatchPriority`	The priority of the threads used for persistent message dispatch	6

6.2.6 In-Memory Messaging using an Embedded Daemon

When you create a JMS Grid client connection, you can specify that instead of connecting to a remote JMS Grid daemon it should create and use a daemon that is embedded into the same JVM as the client. The embedded Message Daemon behaves in the same way as a normal Message Daemon (allowing clustering, networks, fail-over etc).

To use the embedded driver, the client-side parameter `driverName` should be set to `JMSGridEmbedded`. The properties used to initialize the `javax.jms.ConnectionFactory` (whether from JNDI or directly) are the same ones expected to initialize a running Message Daemon.

When using more than one connection in the same JVM, providing the embedded Message Daemon has the same `bindAddresses` property, then the same instance will be shared.

There are pros and cons of using an embedded Message Daemon:

Pros:

- Messages passed to/from the embedded daemon from the JMS Connection using it are passed straight through without being transformed into an on-the-wire format and do not incur the overhead of being sent via a TCP/IP Socket
- Many JMS Connections in the same JVM can share the same embedded instance
- The embedded daemon behaves in the same way as an off process Message Daemon

Cons:

- Impact on an existing application of running a multi-threaded message-processing engine should not be ignored
- Could make application debugging harder

6.2.7 Flow Control

JMS Grid internally monitors the resource usage of message daemons and connected JMS clients. On the clients the amount of memory consumed while queuing messages is monitored, while on the message daemons memory usage and disk usage by the message store is measured.

Resource Utilization on the Client

JMS Grid use of a push-based model for message delivery ensures that messages are delivered as quickly as possible to the clients. Internally messages are queued into an internal dispatch queue, using much the same mechanism that is used in the Message Daemon. A single dispatch queue is associated with every JMS Session used on the client. The client-side parameter that defines how much memory that can be used by the internal queues is `maxInternalQueueSize`, which is the value in bytes. The default value is 8388608 (8 MB).

This limit applies to a JMS Connection. This means that if you are using multiple JMS Connections within the same JVM, the value of `maxInternalQueueSize` may need to be decreased accordingly.

There are two further parameters which control the number of messages that a daemon will push (pre-send) to a client:

`marWindowSize` (default = 100) applies to both queues and topics, and specifies the maximum number of unacknowledged messages that the daemon will dispatch from an individual queue or durable subscription. When this limit is reached the daemon will not send any further messages from that queue or durable subscription until previous messages have been acknowledged or committed.

This limit is per queue or durable subscription, not per consumer, which means that in the case of queues (which may have more than one consumer), this parameter controls the total number of unacknowledged messages than will be dispatched across all consumers.

`maxQueuePresend` (default = 10) applies to queues only, and specifies the maximum number of unacknowledged messages that the daemon will send to a queue receiver. When this limit is reached, the daemon will not send any further messages to the receiver until previous messages have been acknowledged or committed.

In the case of queues, both of these parameters are used and whichever limit is reached first will apply.

Note that the parameters `marWindowSize` and `maxQueuePresend` have the effect of limiting the number of messages that can be consumed in a single transaction.

Resource Utilization on the Message Daemon

The maximum amount of memory that messages can consume when in transit is determined by the configuration parameter `maxInternalQueueSize`. The default is 33554432 (32 MB). The configuration parameter that limits the amount of space utilized by persisted messages in the message store is `spiritDBMaxFileSizeInMbytes`. The default size is 512 MB.

Flow Control Strategies

The following paragraphs outline the different flow control strategies used by JMS Grid. Flow control strategies affect single Message Daemons, clusters and networks of clusters.

Slow Client Consumers of Persisted Messages

Messages that are persisted in the message store are dispatched independently of transient (i.e. not persisted) messages. If a client reaches the pre-fetch limit, the dispatcher will wait until the client is ready to consume some more messages. When messages are being produced faster than they are consumed, the size limit on the message store can be reached. This will result in message producers controlling their rate of message delivery to the daemon.

Slow Client Consumers of Transient Messages

When the Message Daemon detects that a client is reaching its internal memory limit (as defined by the `maxInternalQueueSize` parameter) it will move messages to be dispatched to the client to a separate slow dispatch queue. In this way, slow consumers do not impact the delivery of messages to every other client. However, if the client is consuming messages slowly for a long time, then it can impact the amount of memory utilized for message processing by the Message Daemon. It is advisable for applications that are known to consume messages slowly to use a persisted message delivery strategy instead.

Message Producer Throttling

When a memory of disk space utilization on the Message Daemon reaches its limit, Message Producers are throttled by pausing for a configurable amount of time. This is determined by the configuration parameter `throttleTimeouts`, which is a string of comma separated values. The default is: "10000,5000,1000,500,50,5,0,0,0,0" which corresponds to the following table:

Table 53 Throttle Timeouts

Daemon Utilization %	100	90	80	70	60	50	40	30	20	10
Timeout (ms)	10000	5000	1000	500	50	5	0	0	0	0

6.2.8 Using Selectors and Destination Hierarchies

JMS Grid supports hierarchies for destinations (both Topics and Queues) and wildcards for selecting - see the JMS Grid Programming chapter for more information. This type of message selection supplements the standard JMS message selectors, and the content based selectors that JMS Grid provides.

Using destination hierarchies will in general always be quicker than selectors. Although all routing and message selection occurs in the daemon, using message selectors incurs the additional overhead of deserializing the message from the *on-the-wire* format used between JMS Clients and Message Daemons.

A mixture of destination hierarchies and wildcards can also be a good combination, where fine-grained routing of messages is required. The only disadvantage of using destination hierarchies and wildcards is that they are non-standard.

6.2.9 Compression

Message compression can improve performance for medium to large messages. There are a number of parameters that are used to control compression, as outlined in the table below:

Table 54 Compression Parameters

Parameter Name	Description	Default Value
`doCompression`	A flag which determines if compression is used at all	true
`doCompressLimit`	The size of message payload (in bytes) above which compression is used, if doCompression is enabled	4096
`doCompressBufferSize`	The size of the internal write buffer used for compression. For large messages, increasing this size can boost performance of compression	1024
`compressionLevel`	The level of compression to be used (0 - 9 where 0 is no compression and 9 is maximum compression)	1

Compression of messages occurs at the client, where they are published. Compression strategies can be applied for the client (client-side configuration) or at the Message Daemon. The only other place where a compression strategy is applied to message payloads is when they leave a message cluster to pass over a network.

It is possible to have different compression strategies for clients and daemons (for network connections) by ensuring daemon configuration changes are not propagated to clients by setting the parameter `overrideClientConfiguration` to *false*.

6.3 Fail-over and Fault Tolerance

This section covers usage of clusters, client load balancing, client failover, and fault tolerance.

6.3.1 Usage of Clusters

JMS Grid supports fully replicated clustering for Message Daemons, where every persistent message is stored on every node in the cluster. Some of the features that JMS Grid offers through its clustering are truly unique. They include the following:

- Automatic Recovery of Message Daemons - Message Daemons can be removed from a cluster (through hardware failure or by design), moved between clusters or joined to a cluster without affecting message integrity or involve management of clients.
- Extremely high availability
- Complete automated fail-over for clients
- Automated load balancing of clients
- True location transparency for both Topics and Queues
- Combined with non-blocking I/O - high scalability

To completely guarantee message integrity for full message replication between cluster nodes, a two-phase commit protocol is used.

6.3.2 Client Load Balancing

JMS Grid supports two types of load balancing for Clients:

1. Random (the default)
2. Least Loaded

A JMS Grid client is initialized with a comma separated string of possible URLs it can connect to by the `messageChannels` parameter. By default, this value is `tcp://localhost:50607`. By default the JMS Grid client will choose one of the URLs supplied by this parameter to connect to at random. This behavior is determined by the client's randomConnection property, which is true by default. If it is set to false then the client will connect to the first URL in the list.

On the daemon, if the daemon parameter connectionLoadBalancing is set to LeastUsedLoading then whenever a new client connects to that daemon, and another daemon exists in the cluster with fewer client connections, then the client connection will be relocated to that daemon. This connection strategy ensures that clients are evenly-loaded across the cluster.

If the daemon parameter connectionLoadBalancing is set to Random Connection (which is the default) then the newly-connected client will not be redirected to other daemons. Note that this value does not in itself cause random load balancing.

When a client connects to a Message Daemon, it is informed of the location of all the Message Daemons of that cluster and is kept informed of changes in cluster membership. This ensures that the client has an accurate list of possible Message Daemons to fail-over too if its local Message Daemon should fail.

If a new or recovered Message Daemon joins an existing cluster, it can be useful to have connected clients redistribute themselves evenly across all the Message Daemons. To enable this feature, set the configuration flag `reloadClientsOnNewClusterDaemon` to *true*.

6.3.3 Client Fail-Over

If a JMS Grid client loses connectivity with a Message Daemon, it will try and reconnect to another Message Daemon in the cluster. The client will block all active threads sending messages until after connectivity can be established.

When connectivity is established, JMS Grid determines that current state of the client, to ensure that all message integrity and order are maintained. For any application using JMS Grid, fail-over is non-intrusive and seamless. To ensure that messages are not lost, the last few sent are cached on clients that are publishing. These cached messages are flushed to the Message Daemon on reconnection and the Message Daemon discards duplicates.

The number of attempted retries is determined by the client-side parameter `defaultConnectionRetries` which by default is 10. After an unsuccessful retry, the client will sleep for `defaultConnectionRetriesTimeout` (default is 30secs) before trying reconnect again.

It is only after `defaultConnectionRetries` all are exhausted that the client will throw a JMSException.

6.3.4 Fault Tolerance

The default message store used by the Message Daemon writes messages to files locally to disk. There is always a small risk that the integrity of the message store could be compromised if there was a sudden catastrophic disk failure. The automatic recovery algorithms used for JMS Grid clusters can usually recover these failures. To ensure that every message store operation is written to disk and not lost in a file buffer, then the parameter flag `useSync` should be set to *true*.

6.4 Configuration Parameters for Daemons

Table 55 Parameters for Message Daemons

Parameter Name	Description	Default Value
autoDiscoveryAllowed	Determines whether the daemon will register as a service for clients to automatically discover, and whether the daemon itself will use multicast discovery to locate other daemons.	false
bindAddresses	urls the daemon will attach to	tcp://localhost:50607
closeClientsOnNetworkConnectionFailure	close all connections on a network connection failure	false
clusterCompleteTimout	timeout (seconds) awaiting for the cluster to complete.	30
clusterID	unique id for the cluster the daemon belongs to	
configurationPollingTimeInSeconds	The time in seconds before polling for updates to the configuration	30
connectionLoadBalancing	Either RandomLoading (default) or LeastUsedLoading	RandomLoading
daemonClusteredConnections	URLS of daemons in the cluster to connect with	
daemonConnectionRetriesTimeout	timeout in seconds before retrying to establish connection to another daemon	5000
daemonNetworkConnections	URLS of daemons in the hierarchy to connect with	
dataBackupDir	absolute path of backup directory for data blocks	
dataBlockSizeInBytes	size of the SPIRITDB message block extent. Can improve performance on some platforms	10485760
doConfigurationPolling	Allow for polling for changes to the configuration	true
doDaemonRecovery	daemons sync databases on connection in a cluster	true

Table 55 Parameters for Message Daemons (Continued)

Parameter Name	Description	Default Value
exceptionOnNoQueueReceiver	throw an exception on client of no receiver for a Queue	false
indexBlockSizeInBytes	size of the SPIRITDB index blocks. Can improve performance on some platforms	1048576
initialPooledDBConnections	initial number of JDBC connections per daemon	1
isSecure	is authentication and authorization enabled on the JMS Grid Message Server	false
jmxTimeout	The maximum time a synchronous JMX client will wait for a response from this daemon. This value is defined on the daemon and passed to the client when it first connects.	30000
keepConfigurationUpdatesLocal	For changes made locally to configuration or from JMX - do not update JNDI	false
marWindowSize	Specifies the maximum number of unacknowledged messages that the daemon will dispatch from an individual queue or durable subscription. When this limit is reached the daemon will not send any further messages from that queue or durable subscription until previous messages have been acknowledged or committed. This limit is per queue or durable subscription, not per consumer, which means that in the case of queues (which may have more than one consumer), this parameter controls the total number of unacknowledged messages than will be dispatched across all consumers. Note: The term "MAR" relates to the internal "Mark As Read" message used to handle message acknowledgement.	100
maxDaemonConnectionRetries	number of retires attempting to connect to another daemon	2147483647
maxInternalQueueSize	set the amount of VM memory allowed to be used by the daemon's message dispatch Queues (in bytes)	33554432

Table 55 Parameters for Message Daemons (Continued)

Parameter Name	Description	Default Value
maxLogBackupIndex	max number of backup log files that will be created	10
maxLogFileSize	the maximum size of the log file before it is archived - it's moved to a <log file name>.<number>	2000000
maxQueuePresend	Specifies the maximum number of unacknowledged messages that the daemon will send to a queue receiver. When this limit is reached, the daemon will not send any further messages to the queue receiver until previous messages have been acknowledged or committed.	10
maxThrottleTimeout	maximum time in milliseconds before a client will attempt to send another message to the JMS Grid message server which has reached capacity (either in memory consumption or disk space)	5000
maxTopicDispatchQueues	maximum number of separate dispatch queues used for distributing publish/ subscribe (Topic) messages	0
name	unique name for the message server	
netConnectPoolSessions	The number of ServerSessions used on a Network Connection to a remote cluster	2
networkConnectionQueueFilters	filter used to restrict queue propagation across cluster boundaries	null
networkConnectionTopicFilters	filter used to restrict topic propagation across cluster boundaries	null
overrideClientConfiguration	Override client configuration from the daemon configuration	true
password	password for firewall proxy server	null
pingEnabled	use ping protocol to determine network outage	false
pingTimeout	time (ms) before next ping	5000
producerHistorySize	number of MessageProducers the JMS Grid Message Server will cache information for (like last sent message)	2048

Table 55 Parameters for Message Daemons (Continued)

Parameter Name	Description	Default Value
producerHistoryTimeout	number of msg IDs to record for each producer	0
proxyHost	Hostname or IP address of firewall proxy server	null
proxyPort	port for firewall proxy	0
reloadClientsOnNewClusterDaemon	Reload clients across the cluster if a new daemon joins	false
restrictedNetworkTopology	Restrict message flow to enable hub/spoke topology for networks of clusters	true
serviceDiscoveryChannel	Multicast channel used for automatic discovery of daemons. Only used if the allowAutoDiscovery parameter is set	multicast://224.0.0.4:3495
spiritDBInitialFileSizeInBytes	size of the SPIRITDB message block extent. Can improve performance on some platforms	10485760
spiritDBMaxFileSizeInMbytes	maximum size of the SPIRITDB.	512
spiritDBThrottleThresholdPercentage	high water mark for publisher throttling.	80
spiritDbDispatchThreads	Number of threads used to dispatch messages from the message store. There will be this number of threads dispatching from queues and the same number of threads dispatching from topics.	
storeName	JNDI name (relative to parent context) under which the configuration of this object is stored. Not normally changed except through admin tool.	null
tcpNoDelay	enable/disable Nagle's algorithm for tcp sockets	false
tunnelSSLServerCertFilename	SSL Server Certificate	used to authenticate Server to Client
tunnelSSLServerCipher	Cipher used during crypto session with an SSL Client	SSL_DH_anon_WITH_DES_CBC_SHA
tunnelSSLServerPrivateKeyFilename	Encrypted Private Key	

Table 55 Parameters for Message Daemons (Continued)

Parameter Name	Description	Default Value
`tunnelSSLServerPrivateKeyPassword`	Password for encrypted Private Key	
`tunnelSSLServerRootCACertFilename`	Client Certificate must be signed by this RootCA Certificate	
`tunnelSSLServerTrustedCAs`	Trusted CA Certificates	
`tunnelSSLdoClientAuthentication`	Request Client Authentication using an SSL Certificate	false
`useDataBackup`	If true, old data blocks will be moved to a backup directory rather than be deleted	false
`useSync`	If set to true, daemon will sync file after each message write	false
`username`	User name for proxy	null

6.5 Configuration Parameters Common to Daemons and Clients

Table 56 Parameters for Message Daemons and Clients

Parameter Name	Description	Default Value
`clientSessionThreadPriority`	The default priority of JMS Session threads	6
`compressionLevel`	set the compression level (0-9) - the default is 1 (quickest compression)	1
`compressionStrategy`	the strategy used (default is DEFAULT : value = 0)- other strategies are: `"FILTERED value = 1 and HUFFMAN_ONLY value = 2`	0
`daemonDispatchQueuePriority`	The priority of threads for internal wave daemon message dispatch queues	6

Table 56 Parameters for Message Daemons and Clients (Continued)

Parameter Name	Description	Default Value
daemonPersistentDispatchPriority	The priority of threads for dispatching persistent messages from the message store	6
defaultPooledThreadPriority	The default priority of threads used by the default thread pool	5
doCompressBufferSize	size of buffer used when compressing network messages	1024
doCompressLimit	the size of a message (in bytes) above which compression is used	4096
doCompression	flag to indicate compression should be used for messages	true
flushMessageOnPublish	when using TCP/IP as the transport - enable flushing of packets on to the underlying TCP/IP Socket - This may improve performance of request/reply semantics on some platforms (e.g. VMS).	false
networkStreamBufferSize	size of the underlying socket buffer (bytes)	131072
networkTimeout	The maximum time a synchronous call will wait for a response	120000
receiverThreadPriority	The default priority of JMS client receivers	7
serviceDiscoveryChannel	Multicast channel used for automatic discovery of daemons. Only used if the allowAutoDiscovery parameter is set	multicast://224.0.0.4:3495
socketReceiverPriority	The priority of threads used to read data from sockets	5
socketSoTimeout	default SoTimeout for Sockets	30000
throttleTimeouts	sleep times (ms) for publisher according to availability of resource values are for 0 to 100% in 10% intervals	10000,5000,1000,500,0,..

Table 56 Parameters for Message Daemons and Clients (Continued)

Parameter Name	Description	Default Value
waveCloseConnectionOnSlowConsumer	determines if Wave should close Connection of a slow consumer	false
waveCloseSessionOnSlowConsumer	determines if Wave should close the session for a slow consumer	false
waveConsumerMessageQueueMaxSize	maximum internal cardinality of session queue	200

6.6 Configuration Parameters for Clients

Table 57 Parameters - Client Side

Parameter Name	Description	Default Value
autoDiscoveryAllowed	Use multicast discovery to find a Message Daemon to connect to (this parameter needs to be enabled on Message Daemon as well)	false
consumeDispatchThread	Consume messages from the Socket Thread in a separate dispatch thread	false
defaultConnectionRetries	Number of attempted retries to connect to a cluster	10
defaultConnectionRetriesTimeout	Time to pause (in seconds) before retrying to connect to a cluster	30
initialConnectionTimeout	Time in milliseconds to wait for initial acknowledgement when a connection is being set up. If the time expires with no acknowledgement the connection fails with a JMSException. The default is 30 minutes to allow time for recovery to take place if necessary. This is because during recovery no acknowledgement will be received.	1800000 (30 minutes)
maxInternalQueueSize	Maximum amount of memory to be used by JVM consuming JMS Messages	8388608
messageChannels	Connection URLs to the cluster	tcp://localhost:50607

Table 57 Parameters - Client Side (Continued)

Parameter Name	Description	Default Value
`pingEnabled`	Keep connection alive protocol	false
`pingTimeout`	Timeout (ms) before ping sent on socket if no traffic	5000
`produceDispatchThread`	Use a separate thread to asynchronously dispatch messages to a cluster	false

Chapter 7

Integrating JMS Grid with Application Servers

The J2EE (TM) Connector Architecture (JCA) specification sets out how application servers can be connected to heterogeneous external systems. This fully specifies the contract between an application server and an external system so that provided each has fulfilled its side of the contract the application server can connect to any compliant external system and vice versa. In concrete terms the external system vendor provides a resource adapter, which is a special archive with suffix .rar. This has to be placed somewhere in an application server's file system. In practice, other application server specific files or file entries usually need to be made to create a fully functional application.

The latest release of JMS Grid comes with a JCA 1.5 compliant resource adapter that allows you to plug it into any compliant application server and run applications using JMS Grid queues and topics. The following sections describe the additional things you need to do for each application server.

7.1 Using JMS Grid with the JBoss Application Server

The following example consists of a Message Driven Bean (MDB) which listens for messages on a queue and echoes them back on a second queue.

This example illustrates the files and file entries needed to get an application working.

Important: *The examples described here have been tested with version 4.0.4-GA of JBoss, using JDK versions 1.4.2 and release 5.1.2 of JMS Grid.*

There are four files which are important for the integration:

- The resource adapter archive. This can be found in the JMS Grid distribution in the jmsjca directory and is called `rawave.rar`.
- A JBoss specific file which defines a new data source: a JMS Grid data source. You have provided an example of such a file in the JBoss example. It is called `jmsgrid-jms-ds.xml`. Note that data source descriptor files must be named `<something>-ds.xml`.
- The applications deployment descriptors, `ejb-jar.xml` and `jboss.xml` need to provide information specific to indicate the use of a 'foreign' JMS provider.

There are interrelationships between these files which mean that names must match or the application will not work. These interrelationships are indicated in detail in the following sections.

You can find all the example code, descriptors and a build file for this integration in the examples directory jboss-integration.

Step 1 – Add the Resource Adapter and Data Source Descriptor

The steps in this section deal with global configuration, to make JMS Grid available to any application that wishes to use it.

The resource adapter file can be deployed without any change, but the data source descriptor may need to be changed. This file leads to the creation of connection factories whose attributes are described in the resource adapter. The precise meaning of all the elements in this file are beyond the scope of this document.

See `http://www.jboss.org/j2ee/dtd/jboss-ds_1_5.dtd` for more details.

However, to take an example from the sample file:

```
    <tx-connection-factory>
       <jndi-name>jmsgrid/QueueConnectionFactory</jndi-name>
       <xa-transaction/>
       <track-connection-by-tx/>
       <rar-name>rawave.rar</rar-name>
       <connection-definition>javax.jms.QueueConnectionFactory</
connection-definition>
    </tx-connection-factory>
```

This sets up a connection factory which supports transactions (`<tx-connection-factory>`) which will be bound at `jmsgrid/QueueConnectionFactory`, supports XA transactions and will be an instance of `javax.jms.QueueConnectionFactory`. The concrete class which implements the connection factory will be found in the resource adapter named (`<rar-name>`). A connection will be dedicated to one transaction until it concludes (`<track-connection-by-tx>`).

You also need to create any destinations which will not be created automatically when the application is deployed. For your application this is the outbound queue and this is created via a management bean (MBean):

```
   <mbean code="org.jboss.resource.deployment.AdminObject"
          name="jmsgrid.queue:name=mdb.outbound">
       <attribute name="JNDIName">jmsgrid/queue/outbound</attribute>
       <depends optional-attribute-name="RARName">
            jboss.jca:service=RARDeployment,name='rawave.rar'
       </depends>
       <attribute name="Type">javax.jms.Queue</attribute>
       <attribute name="Properties">Name=mdb.outbound</attribute>
    </mbean>
```

The MBean of type `org.jboss.resource.deployment.AdminObject` is responsible for creating a queue called `mdb.outbound`. In the same way as for the connection factory it finds the concrete class implementing `javax.jms.Queue` in the resource adapter.

Once you have set up your data source descriptor you can deploy these files:

Copy `rawave.rar` to the deploy directory of your JBoss server. If you are using the default server type, that would be `$JBOSS_HOME/server/default/deploy`.

Copy the data source descriptor for JMS Grid to the `deploy/jms directory` of your JBoss server. If you are using the default server type, this would be `$JBOSS_HOME/server/default/deploy/jms`.

When you start up JBoss or add these files to a running system, you should see some messages confirming that the deployments are successful:

```
13:53:58,859 INFO  [ConnectionFactoryBindingService] Bound
ConnectionManager 'jboss.jca:name=jmsgrid/
QueueConnectionFactory,service=ConnectionFactoryBinding' to JNDI name
'java:jmsgrid/QueueConnectionFactory'
13:53:58,875 INFO  [ConnectionFactoryBindingService] Bound
ConnectionManager 'jboss.jca:name=jmsgrid/
TopicConnectionFactory,service=ConnectionFactoryBinding' to JNDI name
'java:jmsgrid/TopicConnectionFactory'
13:53:58,953 INFO  [AdminObject] Bound admin object
'com.spirit.wave.message.DefaultQueue' at 'jmsgrid/queue/outbound'
```

You can also use the JBoss management console to check the various MBeans which are associated with the connection factories and destinations and to see what has been bound into the JNDI name space.

The connection factories have MBeans in the `jboss.jca domain`, and the queue you created shows up in the `jmsgrid.queue` domain, as you defined in your descriptor. You can check the JNDI name space by using the `list()` operation on the JNDIView MBean. This bean is in the jboss domain. This shows that your connection factories are bound in the java: namespace and the queue is in the global name space.

Step 2 – The Source Code

The MDB for this example is in the file `EchoingBean.java`. This is a simple Message Driven Bean implementation. The only things to take particular note of here are the JNDI binding names of the connection factory and outbound destination that are used. These names appear in the deployment descriptors discussed in the next section:

Connection Factory at `java:comp/env/jms/MyQueueConnectionFactory`

Outbound queue at `java:comp/env/jms/mdbOut`

A simple client which sends messages onto the inbound queue of the MDB and listens for the echoes on its outbound queue is in SimpleClient.java.

Step 3 – Write the Deployment Descriptors

There are two deployment descriptors you need to provide:

1. `ejb-jar.xml`

This is the standard descriptor for describing the Enterprise Beans which exist in an application. The first thing you will notice about this is the series of elements enclosed by `<activation-config>`. The elements you need to put in here are determined by the descriptor of the resource adapter. This is `ra.xml`, which is available in the `jmsjca` directory of the JMS Grid distribution. The part of `ra.xml` which is important here is that which starts with `<inbound-resourceadapter>` and in particular this part:

```
<activationspec>
  <activationspec-class>
      com.stc.jmsjca.wave.RAWaveActivationSpec
  </activationspec-class>
  <required-config-property>
    <config-property-name>destination</config-property-name>
  </required-config-property>
  <required-config-property>
    <config-property-name>destinationType</config-property-name>
  </required-config-property>
  <!--
```

The following settings are optional

```
     connectionURL          : default: from ra.xml
     options                : default: form ra.xml
     userName               : default: form ra.xml
     password               : default: form ra.xml
     subscriptionDurability : either Durable or NonDurable
     subscriptionName       : required if Durable
     cliendId               : default: none / auto-generated if
necessary
     messageSelector        : default: none
     concurrencyMode        : CC, Serial, ...; default: Serial
     endpointPoolMaxSize    : default: 15 message driven beans
     contextName            : default: none
     mBeanName              : default: none
   -->
</activationspec>
```

You have to supply one `<activation-config-property>` name and value for each `<required-config-property>` and the same for any of the optional ones given here. Some have defaults, as indicated, and these appear at the start of `ra.xml` in the `<config-property>` entries. For this example, you will use all the defaults, but just add entries for the compulsory attributes destination and destination type.

Note: *The value given for destination is the name of the physical destination, and is not a JNDI name.*

You add an entry for the connection factory which your MDB will use to send outbound messages:

```
<resource-ref>
  <res-ref-name>jms/MyQueueConnectionFactory</res-ref-name>
  <res-type>javax.jms.QueueConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

The JNDI name is the same as the one set in the MDB code. Finally you need an entry for the outbound queue it will use:

```
<message-destination-ref>
  <message-destination-ref-name>jms/mdbOut</message-destination-ref-
name>
  <message-destination-type>javax.jms.Queue</message-destination-
type>
  <message-destination-usage>Produces</message-destination-usage>
  <message-destination-link>mdb.outbound</message-destination-link>
</message-destination-ref>
```

There are two things to note here: the JNDI name matches the one used in the MDB code and the destination named here is the same one you created in the data source descriptor.

2 `jboss.xml`

This file maps everything you have created in `ejb-jar.xml` to objects bound in JNDI, the resource adapter and configures a container in which all this will run. Taking each item in turn:

```
<enterprise-beans>
  <message-driven>
    <ejb-name>EchoingEJB</ejb-name>
    <resource-adapter-name>rawave.rar</resource-adapter-name>
    <configuration-name>JMS Grid Message Driven Bean</configuration-
name>
    <resource-ref>
      <res-ref-name>jms/MyQueueConnectionFactory</res-ref-name>
      <resource-name>queuefactoryref</resource-name>
    </resource-ref>
  </message-driven>
</enterprise-beans>
```

This section is for you MDB, indicating its resource adapter file and indicating that you need to reference a resource with the name `jms/MyQueueConnectionFactory`. The next part but one indicates exactly what that is and where it is bound in JNDI.

```
<assembly-descriptor>
  <message-destination>
    <message-destination-name>mdb.outbound</message-destination-name>
    <jndi-name>jmsgrid/queue/outbound</jndi-name>
  </message-destination>
</assembly-descriptor>
```

This refers back to the outbound queue you referenced in `ejb-jar.xml`, and indicates where it is to be found in JNDI. This is another reference to the destination you created in the data source descriptor.

```
    <resource-manager>
        <res-name>queuefactoryref</res-name>
        <res-jndi-name>java:/jmsgrid/QueueConnectionFactory</res-
jndi-name>
    </resource-manager>
```

This binds your resource reference to a JNDI name where your Connection Factory will be bound.

Note: *The JNDI name is the same as the one you set in the data source descriptor.*

The `<invoker-proxy-bindings>` section relates to how messages are delivered from the service provider to the MDBs that want to consume them. The JCA specification defines a MessageEndpoint interface, to be implemented by the application server, as the mechanism to do this. The application server supplies the resource adapter with a MessageEndpointFactory which it uses to create MessageEndpoints to which it can deliver messages when they are received.

The `<container-configurations>` part then defines a container in which the MDBs will run. This will get its messages via the `MessageEndpointFactory` you defined in `<invoker-proxy-bindings>`.

Step 4 - Building and Running the Application

The example is provided with an ant build file which can be used to create the ear file and deploy it to JBoss. Assuming that you have ant in your path and `$JBOSS_HOME` is set then:

```
ant assemble-mdb
```

will build the ear file and

```
ant deploy-mdb
```

will deploy it to JBoss. Whether JBoss is already running or whether you start JBoss after this step you will see some output:

```
09:37:19,468 INFO  [EARDeployer] Init J2EE application: file:/C:/
jboss-4.0.4.GA/server/default/deploy/EchoingEJB.ear
09:37:19,765 INFO  [EjbModule] Deploying EchoingEJB
09:37:19,921 INFO  [EJBDeployer] Deployed: file:/C:/jboss-4.0.4.GA/
server/default/tmp/deploy/tmp41838EchoingEJB.ear-contents/
echoingejb.jar
09:37:20,156 INFO  [EARDeployer] Started J2EE application: file:/C:/
jboss-4.0.4-GA/server/default/deploy/EchoingEJB.ear
```

Note: *Once JBoss is running with the deployed MDB, you must also have a JMS Grid daemon running to make a connection.*

To run the client you need to have JMSGRID set and then simply type:

```
ant run-client
```

You will see the following output:

Buildfile: `build.xml`

```
run-client:

     [java] 0    [main] INFO  com.spirit.jmq.JMQConnection  - JMS Grid
client version = 7.0.0 Build:DEV-PHILIP-20060530-1500
     [java] 1500 [main] INFO  com.spirit.jmq.JMQConnection  -
Successfully connected to JMS Grid Message Daemon (daemon=philip-
50607 @ tcp://philip:50607) version 7.0.0 Build:DEV-PHILIP-20060530-
1500
     [java] Sending message: This is message 1
     [java] Sending message: This is message 2
     [java] Sending message: This is message 3
    [java] SimpleMessageClient: Message received: EchoingBean This is
message 1
    [java] SimpleMessageClient: Message received: EchoingBean This is
message 2
    [java] SimpleMessageClient: Message received: EchoingBean This is
message 3
     [java] All messages received
     [java] 8062 [Thread-0] WARN  com.spirit.jmq.JMQConnection  - JVM
calling sh
utdown on Connection
```

Chapter 8

Tools

This chapter describes a number of tools which may be useful when using JMS Grid. These tools may be found in the tools directory under your JMS Grid installation.

8.1 Tools to Dump and Create Durable Subscription Definitions

This section describes a pair of commands which can be used to transfer the definitions of durable subscriptions from one daemon's message store to another. This can be useful if you want to create an empty message store which contains the same durable subscriptions as some other message store.

The `dumpsubs` command creates a file containing definitions of all the durable subscriptions that exist in the message store of a specified daemon.

The `buildsubs` command can then be used to read this file and create a new message store which is empty apart from these durable subscriptions. Note that the durable subscriptions thereby created are empty: this mechanism does not copy messages.

8.1.1 Dumping Durable Subscription Definitions to a File

The `dumpsubs` command dumps the definitions of all durable subscriptions that exist in the specified daemon to the specified file. Only the definitions of durable subscriptions are dumped: subscription name, clientID and topic name. Individual messages are not dumped.

This file is intended to be used by the `buildsubs` command only. Its format may change without notice in future releases of JMS Grid.

Syntax:

```
dumpsubs [-w workingDirectory -n <daemonName> -o <dumpfile>]

where
```

-w specifies the daemon working directory (usually `wdir` under the JMS Grid installation)

-n specifies the name of the daemon whose message store should be read

-o specifies the name of an output file; if not specified, the name `dumpsubs daemonName.xml` will be used and the file will be created in the current directory

-h syntax help

Note: *You must either specify both the -w and -n parameters or specify neither of them. If -w and -n are not both supplied then it is assumed that the message store is in in the current directory.*

Example:

```
dumpsubs -w ../wdir -n daemon_alpha -o dumpsubs_alpha.xml
```

This command starts scanning of all `st_*` files residing in the `../wdir/data/daemon_alpha` directory and stores the resulting XML file as

`./dumpsubs_alpha.xml.`

8.1.2 Building Durable Subscriptions from a File

The buildsubs command creates an empty message store containing the durable subscriptions defined in the specified file. The file must be in the format used by the `dumpsubs` command. The message store that will be created will be for the specified daemon in the specified working directory.

Syntax:

```
buildsubs -f <dumpfile> [-w<dirname> -n <daemonName>]

where
```

-f specifies the name of a file containing definitions of durable subscriptions

-w specifies a JMS Grid daemon working directory (usually wdir under the JMS Grid installation)

-n specifies a daemon name whose message store should be created

-h syntax help

Note: *You must either specify both -w and -n parameters or specify neither of them. If -w and -n are not both supplied then the durable subscriptions will be created in the current directory.*

Example:

```
buildsubs -f dumpsubs-alpha.xml -w ../wdir -n daemon_beta
```

This command recreates a durable subscription message store, stored in the file `dumpsubs-alpha.xml`, in the directory `../wdir/data/daemon_beta`.

Index

K

M

N

O

P

Q

R

S

T

U

V