Sun Java System Message Queue 4.1 Release Notes

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–7753–10
September 2007

070919@18741

Contents

◆ ◆ ◆ CHAPTER 1

Sun Java System Message Queue 4.1 Release Notes

Version 4.1

Part Number 819-7753

These release notes contain important information available at the time of release of Sun Java™ System Message Queue 4.1. New features and enhancements, known issues and limitations, and other information are addressed here. Read this document before you begin using Message Queue. These release notes also contain information about the 4.0 release of Message Queue; see "About Message Queue 4.0" on page 16 for information about features introduced in that release.

The most up-to-date version of these release notes can be found at the Sun Java System Message Queue documentation web site. Check the web site prior to installing and setting up your software and then periodically thereafter to view the most up-to-date release notes and product documentation.

These release notes contain the following sections:

Third-party URLs are referenced in this document and provide additional, related information.

Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Release Notes Revision History

The following table lists the dates for all 4.x releases of the Message Queue product and describes the main changes associated with each release.

TABLE 1–1 Revision History

Date	Description of Changes
May 2006	Initial release of this document for version 4.0 of Message Queue.
January 2007	Initial release of this document for Beta version 4.1 of Message Queue. Adds description of JAAS support.
April 2007	Second release of this document for Beta version 4.1 of Message Queue. Adds high availability feature.
September 2007	Third release of this document for customer ship. Adds description of support for Java Enterprise System Monitoring Framework, fixed C ports, bug fixes, and other features.

About Message Queue 4.1

Sun Java System Message Queue is a full-featured message service that provides reliable, asynchronous messaging conformant to the Java Messaging Specification (JMS) 1.1. In addition, Message Queue provides features that go beyond the JMS specification to meet the needs of large-scale enterprise deployments.

Version 4.1 of Message Queue adds support for high availability, for the Java Authentication and Authorization Service (JAAS), for the use of fixed C ports, and for Java Enterprise System Monitoring Framework. It also adds minor enhancements, and bug fixes. This section includes the following information.

- "What's New in The 4.1 Release" on page 7
- "Hardware and Software Requirements" on page 16

For information about features introduced in Message Queue 4.0, see "About Message Queue 4.0" on page 16.

What's New in The 4.1 Release

Message Queue 4.1 introduces high availability (data and service availability) broker clusters, JAAS support, and various other minor features. This section describes these features and provides further references for your use.

High Availability

Message Queue 4.1 introduces high availability clusters, which provide data availability as well as service availability. If a client loses its connection to a high availability broker, it is automatically reconnected to another broker in a cluster. The broker that provides the new connection takes over the failed broker's persistent data and state, and continues to provide uninterrupted service to the failed broker's clients. You can run high availability brokers over a secure connection.

High availability brokers require the use of a highly available database (HADB). If you do not have such a database or if data availability is not important to you, you can continue to use conventional clusters, which offer automatic reconnection and service availability.

Configuring high availability brokers is simple: you specify the following kinds of broker properties for each broker in the cluster.

- *Cluster membership properties,* which specify that the broker is part of a high availability cluster, the id of the cluster, and the id of the broker.
- *Highly available database (HADB) properties*, which specify the model for persistent messages (JDBC), the name of the HADB vendor, and vendor-specific configuration properties for the database.
- *Failure detection and takeover properties*, which specify how broker failure should be detected and handled.

To use this feature, you must do the following:

1. Install a highly available database.
2. Install the JDBC driver's .jar file.

3. Create the database schema for the highly available persistent store.
4. Set those properties that are related to high availability for each broker in the cluster.
5. Start each broker in the cluster.

For a conceptual discussion of high availability and how it compares to conventional clusters, see Chapter 4, "Broker Clusters," in *Sun Java System Message Queue 4.1 Technical Overview*. For procedural and reference information about high availability, see Chapter 8, "Broker Clusters," in *Sun Java System Message Queue 4.1 Administration Guide* and "Cluster Configuration Properties" in *Sun Java System Message Queue 4.1 Administration Guide*.

If you were using an HADB database with Message Queue version 4.0 and want to use a high availability cluster, you can use the `dbmgr` utility to upgrade to a shared HADB store. See "Broker Clusters" on page 42 for more information.

JAAS Support

In addition to the file-based and LDAP-based built-in authentication mechanisms, Message Queue also supports the Java Authentication and Authorization Service (JAAS), which allows you to plug a variety of services into the broker to authenticate Message Queue clients. This section describes the information that the broker makes available to a JAAS-compliant authentication service, and it explains how you configure the broker to use such a service.

It is beyond the scope of this document to describe the JAAS API. Please consult the following sources if you need to know more.

- For complete information about the JAAS API, please see the *Java Authentication and Authorization Service (JAAS) Reference Guide*.

  `http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/JAASRefGuide.html`

- For information about writing a LoginModule, please see the *Java Authentication and Authorization Service (JAAS) LoginModule Developer's Guide*.

  `http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/JAASLMDevGuide.html`

The JAAS API is a core API in J2SE and therefore it is an integral part of Message Queue's runtime environment. JAAS defines an abstraction layer between an application and an authentication mechanism, allowing the desired mechanism to be plugged in with no change to application code. In the case of the Message Queue service, the abstraction layer lies between the broker (application) and an authentication provider. By setting a few broker properties, it is possible to plug in any JAAS-compliant authentication service and to upgrade this service with no disruption or change to broker code.

You can use JMX clients to manage the broker if you are using JAAS-based authentication, but you must manually set up JAAS support (by setting JAAS-related broker properties) before you start the broker. You cannot use the JMX API to change those properties.

Elements of JAAS

Figure 1–1 shows the basic elements of JAAS: a JAAS client, a JAAS-compliant authentication service, and a JAAS configuration file.

- The JAAS client is an application that wants to perform authentication using a JAAS-compliant authentication service. It communicates with this service using a LoginModule and it is responsible for providing a callback handler that the LoginModule can call to obtain the user name, password, and other relevant information.
- The JAAS-compliant Authentication Service consists of one or more LoginModule and of logic that performs the needed authentication. The LoginModule may include the authentication logic, or it may use a private protocol or API to communicate with a module that provides that logic.
- The JAAS configuration file is a text file that the JAAS client uses to locate the LoginModule(s) needed to communicate with the JAAS-compliant service.

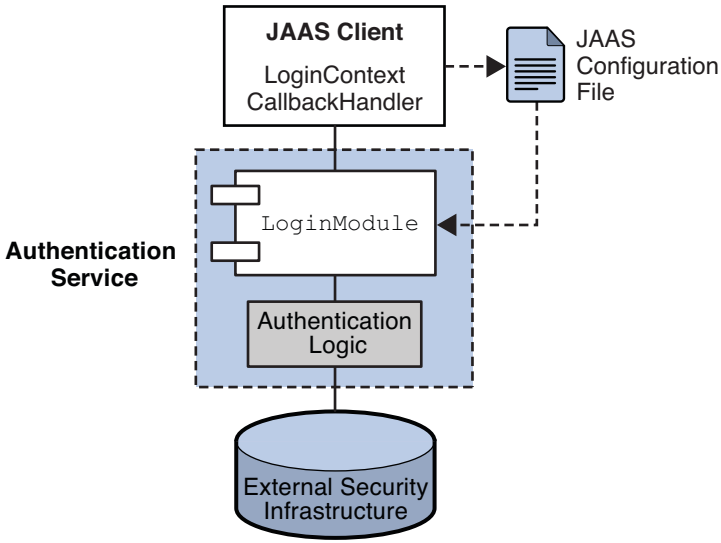

FIGURE 1–1 JAAS Elements

The next section explains how the Message Queue service uses these elements to provide JAAS-compliant authentication.

JAAS and Message Queue

The next figure shows how JAAS is used by the Message Queue broker. It shows a more complex implementation of the JAAS model shown in the previous figure.

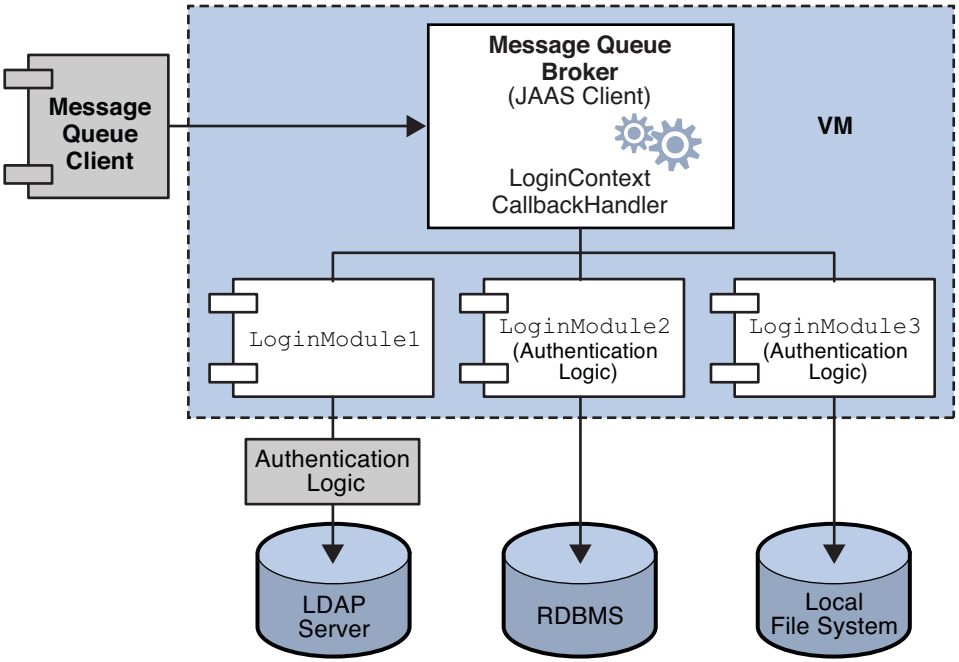

FIGURE 1–2 How Message Queue Uses JAAS

As was shown in the simpler case, the authentication service layer is separate from the broker. The authentication service consists of one or more login modules (LoginModule) and of additional authentication modules if needed. The login modules run in the same Java virtual machine as the broker. The Message Queue broker is represented to the login module as a `LogInContext` and it communicates with the login module by means of a `CallBackHandler` that is part of the broker runtime code.

The authentication service also supplies a JAAS configuration file that contains entries to the login modules. The configuration file specifies the order in which the modules are to be used and some conditions for their use. When the broker starts up, JAAS locates the configuration file by the Java system property `java.security.auth.login.config` or the Java security properties file. It then selects an entry in the JAAS configuration file, according to the value of the broker property `imq.user_repository.jaas.name`. That entry specifies which login modules will be used for authentication. As the figure shows, it is possible for the broker to use more than one login module. (The relation between the configuration file, the login module, and the broker is shown in Figure 1–3.)

The fact that the broker uses a JAAS plug-in authentication service remains completely transparent to the Message Queue client. The client continues to connect to the broker as it did before, passing a user name and password. In turn, the broker uses a callback handler to pass this information to the authentication service, and the service uses that information to

authenticate the user and return the results. If authentication succeeds, the broker grants the connection; if it fails, the client runtime returns a JMS security exception that the client must handle.

After the Message Queue client is authenticated, if there is further authorization to be done, the broker proceeds as it would normally; it consults the access control file to determine whether the authenticated client is authorized to perform the actions it undertakes: accessing a destination, consuming a message, browsing a queue, and so on.

Setting up JAAS-Compliant Authentication

Setting up JAAS-compliant authentication involves setting broker and system properties to select this type of authentication, to specify the location of the configuration file, and to specify the entries to the login modules that are going to be used.

This section illustrates how the JAAS client, the login modules, and the JAAS configuration file are related and then describes the process required to set up JAAS-compliant authentication. The next figure shows the relation between the configuration file, the login module, and the broker.

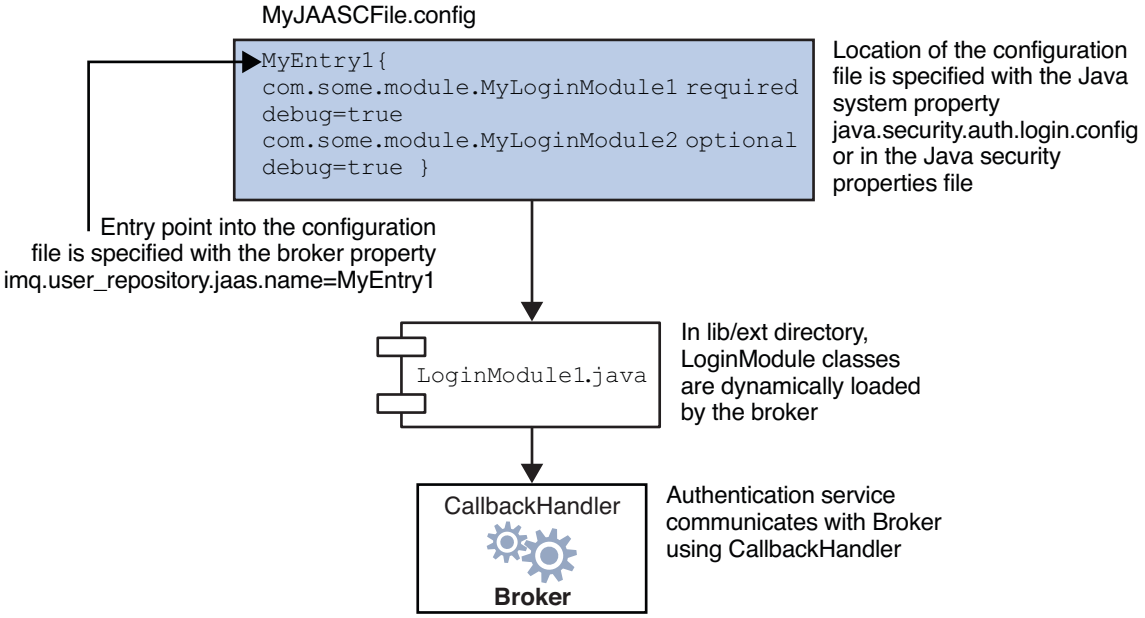

FIGURE 1–3 Setting Up JAAS Support

As shown in the figure, the JAAS configuration file, `MyJAASCFile.config` contains references to several login modules, grouped in an entry point. The broker locates the configuration file by consulting the Java system property `java.security.auth.login.config` or by consulting the

Java Security properties file. The login modules to be used are determined by consulting the broker property `imq.user_repository.jaas.name`, which specifies the desired entry in the configuration file. The classes for those modules are found in the `lib/ext` directory.

To set up JAAS support for Message Queue, you must complete the following steps. (In a development environment all these steps might be done by the developer. In a production environment, the administrator would take over some of these tasks.)

1. Create one or more login module classes that implement the authentication service. The JAAS callback types that the broker supports are listed below.

   `javax.security.auth.callback.LanguageCallback`
   The broker uses this callback to pass the authentication service the locale in which the broker is running This value can be used for localization.

   `javax.security.auth.callback.NameCallback`
   The broker uses this callback to pass to the authentication service the user name specified by the Message Queue client when the connection was requested.

   `javax.security.auth.callback.TextInputCallback`
   The broker uses this callback to specify the value of `imq.authentication.type` to the authentication service when the `TextInputCallback.getPrompt()` is `imq.authentication.type`. Right now, the only possible value for this field is `basic`. This indicates Base-64 password encoding.

   `javax.security.auth.callback.PasswordCallback`
   The broker uses this callback to pass to the authentication service the password specified by the Message Queue client when the connection was requested.

   `javax.security.auth.callback.TextOutputCallback`
   The broker uses this callback to provide logging services to the authentication service by logging the text output to the broker's log file. The callback's MessageType `ERROR, INFORMATION, WARNING` are mapped to the broker log levels `ERROR, INFO,` and `WARNING` respectively.
2. Create a JAAS configuration file with entries that reference the login module classes and specify the location of this file to the Message Queue administrator. (The file can be located remotely, and its location can be specified with a url.)
3. Note the name of the entry (that references the login implementation classes) in the JAAS configuration file.
4. Archive the classes that implement the login modules to a jar file, and place the jar file in the Message Queue `lib/ext` directory.
5. Configure the broker properties that relate to JAAS support. These are described in Table 1–2.
6. Set the following system property to specify the location of the JAAS configuration file.

   `java.security.auth.login.config=`*JAAS_Config_File_Location*

For example, you can specify the configuration file when you start the broker.

```
imqbrokerd -Djava.security.auth.login.config=JAAS_Config_File_Location
```

There are other ways to specify the location of the JAAS configuration file. For additional information, please see

http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/tutorials/LoginConfigFile.html

The following table lists the broker properties needed to set up JAAS support.

TABLE 1–2 Broker Properties for JAAS Support

Property	Description
`imq.authentication.type`	Set to `basic` to indicate Base-64 password encoding. This is the only permissible value for JAAS authentication.
`imq.authentication.basic.user_repository`	Set to `jaas` to specify JAAS authentication.
`imq.accesscontrol.type`	Set to `file`.
`imq.user_repository.jaas.name`	Set to the name of the desired entry (in the JAAS configuration file) that references the login modules you want to use as the authentication mechanism. This is the name you noted in Step 3.
`imq.user_repository.jaas.userPrincipalClass`	This property, used by Message Queue access control, specifies the `java.security.Principal` implementation class in the login module(s) that the broker uses to extract the Principal name to represent the user entity in the Message Queue access control file. If, it is not specified, the user name passed from the Message Queue client when a connection was requested is used instead.
`imq.user_repository.jaas.groupPrincipalClass`	This property, used by Message Queue access control, specifies the `java.security.Principal` implementation class in the login module(s) that the broker uses to extract the Principal name to represent the group entity in the Message Queue access control file. If, it is not specified, the group rules, if any, in the Message Queue access control file are ignored.

Persistent Store Format Change

Version 4.1 of Message Queue changes the JDBC store to support high availability. For this reason the JDBC store version is increased to 410. JDBC store versions 350, 370, and 400 are automatically migrated to the 410 version format.

Please note that the file-based persistent store version remains 370 because no changes were made to it.

Broker Configuration

The property `IMQ_DEFAULT_EXT_JARS` has been added to the `imqenv.conf` file. You can set this property to specify the path names of external .jar files to be included in `CLASSPATH` when the broker starts up. If you use this property to specify the location of external `.jar` files, you will no longer need to copy these files to the `lib/ext` directory. External jars can refer to JDBC drivers or to JAAS login modules. The following sample command, specifies the location of jdbc drivers.

```
IMQ_DEFAULT_EXT_JARS=/opt/SUNWhadb4/lib/hadbjdbc4.jar:/opt/SUNWjavadb/derby.jar
```

JES Monitoring Framework Support

Message Queue supports the Sun Java Enterprise System (JES) Monitoring Framework, which allows Java Enterprise System components to be monitored using a common graphical interface. This interface is implemented by a web-based console called the Sun Java System Monitoring Console. If you are running Message Queue along with other JES components, you might find it more convenient to use a single interface to manage all these components.

The JES monitoring framework defines a common data model (CMM) to be used by all JES component products. This model enables a centralized and uniform view of all JES components. Message Queue exposes the following objects to the JES monitoring framework:

- the installed product
- the broker instance name
- the broker port mapper
- each connection service
- each physical destination
- the persistent store
- the user repository

Each one of these objects is mapped to a CMM object whose attributes can be monitored using the JES monitoring console. At runtime, administrators can use the console to view performance statistics, create rules to monitor automatically, and acknowledge alarms. For detailed information about the mapping of Message Queue objects to CMM objects, see the *Sun Java Enterprise System Monitoring Guide*.

To enable JES monitoring, you must do the following

1. Install and configure all the components in your deployment (Message Queue and other components) according to instructions given in the *Sun Java Enterprise System Installation Guide*.
2. Enable and configure the Monitoring Framework for all of your monitored components, as described in the *Sun Java Enterprise System Monitoring Guide*.

3. Install the Monitoring Console on a separate host, start the master agent, and then start the web server, as described in the *Sun Java Enterprise System Monitoring Guide*.

Using the JES Monitoring Framework will not impact broker performance because all the work of gathering metrics is done by the monitoring framework, which pulls data from the broker's existing monitoring data infrastructure.

Transaction Management

Previously, only transactions in a `PREPARED` state were allowed to be rolled back administratively. That is, if a session that was part of a distributed transaction did not terminate gracefully, the transaction remained in a state that could not be cleaned up by the broker administrator. In Message Queue 4.1, you can use the imqcmd utility to clean up (roll back) transactions that are in the following states: `STARTED`, `FAILED`, `INCOMPLETE`, `COMPLETE`, `PREPARED`.

To help you determine whether a particular transaction can be rolled back (especially when it is not in a `PREPARED` state), the imqcmd utility provides additional data as part of the `imqcmd query txn` output: it provides the connection id for the connection that started the transaction and specifies the time when the transaction was created. Using this information, the administrator can decide whether the transaction needs to be rolled back. In general, the administrator should avoid rolling back a transaction prematurely.

Fixed Ports for C Client Connections

C clients can use the `MQ_SERVICE_PORT_PROPERTY` connection property to specify a fixed port to connect to. This can be useful if you're trying to get through a firewall or if you need to bypass the broker's port mapper service (which assigns ports dynamically).

Remember that you need to configure the JMS service port on the broker side as well. For example, if you want to connect your client via `ssljms` to port 1756, you would do the following.

- On the client side: Set the `MQ_SERVICE_PORT_PROPERTY` to 1756 and set the `MQ_CONNECTION_TYPE_PROPERTY` to `SSL`.
- On the broker side: Set the `imq.`*serviceNameType.protocol*`.port` property to 1756 as follows.

```
imq.ssljms.ssl.port=1756
```

Note – The `MQ_SERVICE_PORT_PROPERTY` connection property was introduced with version 3.7 Update 2 of Message Queue.

Hardware and Software Requirements

For hardware and software requirements for Version 4.1, please consult the *Sun Java System Message Queue 4.1 Installation Guide*.

About Message Queue 4.0

Message Queue 4.0 is a release limited to supporting Application Server 9 PE. It is a minor release that includes a few new features, minor enhancements, and bug fixes. This section includes the following information.

What's New in the 4.0 Release

Message Queue 4.0 includes the following new features:

These are described in the following subsections.

Caution – One of the more minor but potentially disruptive changes introduced with version 4.0 is the deprecation of the command-line option to specify a password. Henceforth, you must store all passwords in a file as described in "Deprecated Password Option" on page 40.

Interface Changes to the C API and C Client Runtime

Version 4.0 of Message Queue adds two new properties which will be set on all messages that have been placed on the dead message queue.

- `JMS_SUN_DMQ_PRODUCING_BROKER` indicates the broker where the message was produced.
- `JMS_SUN_DMQ_DEAD_BROKER` indicates the broker who marked the message dead.

Interface Changes to the Java API and the Java Client Runtime

Version 4.0 of Message Queue adds two new properties which will be set on all messages that have been placed on the dead message queue.

- `JMS_SUN_DMQ_PRODUCING_BROKER` indicates the broker where the message was produced.
- `JMS_SUN_DMQ_DEAD_BROKER` indicates the broker who marked the message dead.

Displaying Information About the Persistent Store

The `query` subcommand was added to the `imqdbmgr` command. Use this subcommand to display information about the persistent store, including the store version, the database user, and whether the database tables have been created.

The following is an example of the information displayed by the command.

```
imqdbmgr query

[04/Oct/2005:15:30:20 PDT] Using plugged-in persistent store:
        version=400
        brokerid=Mozart1756
        database connection url=jdbc:oracle:thin:@Xhome:1521:mqdb
        database user=scott
Running in standalone mode.
Database tables have already been created.
```

Persistent Store Format Changes

Version 3.7 UR1 of Message Queue introduced two changes to the persistent store format to improve performance. One change is to the file store, the other is to the JDBC store.

- Format of Transaction Data Persisted in File Store

  The format of transaction state information stored in the Message Queue file-based persistent store was changed to reduce disk I/O and to improve the performance of JMS transactions.

- Oracle JDBC Store

  In previous versions of Message Queue, the store schema for Oracle used the `LONG RAW` data type to store message data. In Oracle 8, Oracle introduced the `BLOB` data types and deprecated the `LONG RAW` type. Message Queue 3.7 UR1 switched to the `BLOB` data type to improve performance and supportability.

Because these changes impact store compatibility, the store version for both the file store and the JDBC store was changed from 350 to 370 in version 3.7 UR1 of Message Queue.

Version 4.0 of Message Queue introduced changes to the JDBC store for optimization and to support future enhancements. For this reason the JDBC store version was increased to 400. Note that in Version 4.0, the file-based persistent store version remains 370 because no changes were made to it.

Message Queue 4.0 supports automatic conversion of the persistent store to the newest versions of the file-based and JDBC persistent stores. The first time `imqbrokerd` starts, if the utility detects an older store it will migrate the store to the new format, leaving the old store behind.

- File-based store versions 200 and 350 will be migrated to the 370 version format.
- JDBC store versions 350 and 370 will be migrated to the 400 version format. (If you need to upgrade a 200 store, you will need to step through an intermediate 3.5 or 3. 6 release.)

If you should need to roll back this upgrade, you can uninstall Message Queue 4.0 and then reinstall the version you were previously running. Since the older copy of the store is left intact, the broker can run with the older copy of the store.

Broker Administration

The Command utility (`imqcmd`) has added a subcommand and several options that allow administrators to quiesce the broker, to shutdown the broker after a specified interval, to destroy a connection, or to set java system properties (for example, connection related properties.)

- Quiescing the broker moves it into a quiet state, which allows messages to be drained before the broker is shut down or restarted. No new connections can be created to a broker that is being quiesced. To quiesce the broker, enter a command like the following.

  ```
  imqcmd quiesce bkr -b Wolfgang:1756
  ```

- To shut down the broker after a specified interval, enter a command like the following. (The time interval specifies the number of seconds to wait before the broker is shut down.)

  ```
  imqcmd shutdown bkr -b Hastings:1066 -time 90
  ```

  If you specify a time interval, the broker will log a message indicating when shutdown will occur. For example,

  ```
  Shutting down the broker in 29 seconds (29996 milliseconds)
  ```

  While the broker is waiting to shut down, its behavior is affected in the following ways.

  - Administrative jms connections will continue to be accepted.
  - No new jms connections will be accepted.
  - Existing jms connections will continue to work.
  - The broker will not be able to take over for any other broker in a high availability cluster.
  - The imqcmd utility will not block, it will send the request to shut down to the broker and return right away.

- To destroy a connection, enter a command like the following.

```
imqcmd destroy cxn -n 2691475382197166336
```

Use the command `imqcmd list cxn` or `imqcmd query cxn` to obtain the connection ID.

- To set a system property using `imqcmd`, use the new –D option. This is useful for setting or overriding JMS connection factory properties or connection-related java system properties. For example:

```
imqcmd list svc -secure -DimqSSLIsHostTrusted=true
imqcmd list svc -secure -Djavax.net.ssl.trustStore=/tmp/mytruststore
            -Djavax.net.ssl.trustStorePassword=mytrustword
```

For complete information about the syntax of the `imqcmd` command, see Chapter 13, "Command Line Reference," in *Sun Java System Message Queue 4.1 Administration Guide*.

JDBC Persistence Support

Apache Derby Version 10.1.1 is now supported as a JDBC-compliant persistent store provider.

SSL Support

Starting with release 4.0, the default value for the client connection factory property `imqSSLIsHostTrusted` is `false`. If your application depends on the prior default value of `true`, you need to reconfigure and to set the property explicitly to `true`.

You might choose to trust the host when the broker is configured to use self-signed certificates. In this case, in addition to specifying that the connection should use an SSL-based connection service (using the imqConnectionType property), you should set the `imqSSLIsHostTrusted` property to true.

For example, to run client applications securely when the broker uses self-signed certificates, use a command like the following.

```
java -DimqConnectionType=TLS
      -DimqSSLIsHostTrusted=true <ClientAppName>
```

To run the administration tool `imqcmd` securely when the broker uses self-signed certificates, use a command like the following.

```
imqcmd list svc -secure -DimqSSLIsHostTrusted=true
```

JMX Support

A new API has been added for configuring and monitoring Message Queue brokers in conformance with the Java Management Extensions (JMX) specification. Using this API, you can configure and monitor broker functions programmatically from within a Message Queue client application. In earlier versions of Message Queue, these functions were accessible only from the command line or the Administration Console.

The API consists of a set of JMX *Managed Beans (MBeans)* for managing the following Message Queue–related resources:

- Message brokers
- Connection services
- Connections
- Destinations
- Message producers
- Message consumers
- Transactions
- Broker clusters
- Logging
- The Java Virtual Machine (JVM)

These MBeans provide *attributes* and *operations* for synchronously polling and manipulating the state of the underlying resources, as well as *notifications* that allow a client application to listen for and respond asynchronously to state changes as they occur. Using the JMX API, client applications can perform configuration and monitoring tasks like the following:

- Set a broker's port number
- Set a broker's maximum message size
- Pause a connection service
- Set the maximum number of threads for a connection service
- Get the current number of connections on a service
- Destroy a connection
- Create a destination
- Destroy a destination
- Enable or disable auto-creation of destinations
- Purge all messages from a destination
- Get the cumulative number of messages received by a destination since the broker was started
- Get the current state (running or paused) of a queue
- Get the current number of message producers for a topic
- Purge all messages from a durable subscriber
- Get the current JVM heap size

For an introduction to the JMX API and for complete reference information, see the *Sun Java System Message Queue 4.1 Developer's Guide for JMX Clients*.

Broker Support: JMX-Related Properties

Several new broker properties have been added to support the JMX API (see Table 1–3). None of these properties can be set from the command line with the Message Queue Command utility (`imqcmd`). Instead, they can either be set with the `-D` option of the Broker utility (`imqbrokerd`) or edited by hand in the broker's instance configuration file (`config.properties`). In addition, some of these properties (`imq.jmx.rmiregistry.start`, `imq.jmx.rmiregistry.use`, `imq.jmx.rmiregistry.port`) can be set with the new Broker utility options described in Table 1–4. The table lists each option, specifies its type, and describes its use.

TABLE 1–3 New Broker Properties for JMX Support

Property	Type	Description
`imq.jmx.rmiregistry.start`	`Boolean`	Specifies whether to start RMI registry at broker startup. If `true`, the broker will start an RMI registry at the port specified by `imq.jmx.rmiregistry.port` and use it to store the RMI stub for JMX connectors. Note that the value of `imq.jmx.rmiregistry.use` is ignored in this case. **Default value:** `false`
`imq.jmx.rmiregistry.use`	`Boolean`	Specifies whether to use an external RMI registry. Applies only if `imq.jmx.rmiregistry.start` is `false`. If `true`, the broker will use an external RMI registry at the port specified by `imq.jmx.rmiregistry.port` to store the RMI stub for JMX connectors. The external RMI registry must already be running at broker startup. **Default value:** `false`
`imq.jmx.rmiregistry.port`	`Integer`	Port number of RMI registry Applies only if `imq.jmx.rmiregistry.start` or `imq.jmx.rmiregistry.use` is `true`. JMX connectors can then be configured to use the RMI registry by including this port number in the URL path of their JMX service URLs. **Default value:** `1099`
`imq.jmx.connector.list`	`String`	Names of preconfigured JMX connectors, separated by commas **Default value:** `jmxrmi,ssljmxrmi`
`imq.jmx.connector.activelist`	`String`	Names of JMX connectors to be activated at broker startup, separated by commas **Default value:** `jmxrmi`

TABLE 1–3 New Broker Properties for JMX Support *(Continued)*

Property	Type	Description
`imq.jmx.connector.`*connectorName*`.urlpath`	`String`	*urlPath* component of JMX service URL for connector *connectorName* Useful in cases where the JMX service URL path must be set explicitly (such as when a shared external RMI registry is used). **Default value:** If an RMI registry is used to store the RMI stub for JMX connectors (that is, if `imq.jmx.registry.start` or `imq.jmx.registry.use` is `true`) `/jndi/rmi://`*brokerHost*`:`*rmiPort* */brokerHost/brokerPort/connectorName* If an RMI registry is not used (the default case, `imq.jmx.registry.start` and `imq.jmx.registry.use` both `false`): `/stub/`*rmiStub* where *rmiStub* is an encoded and serialized representation of the RMI stub itself
`imq.jmx.connector.`*connectorName*`.useSSL`	`Boolean`	Specifies whether to use a Secure Socket Layer (SSL) for connector *connectorName*. **Default value:** `false`
`imq.jmx.connector.`*connectorName*`.brokerHostTrusted`	`Boolean`	Specifies whether to trust any certificate presented by broker for connector *connectorName*. Applies only when `imq.jmx.connector.`*connectorName*`.useSSL` is `true`. If `false`, the Message Queue client runtime will validate all certificates presented to it. Validation will fail if the signer of the certificate is not in the client's trust store. If `true`, validation of certificates is skipped. This can be useful, for instance, during software testing when a self-signed certificate is used. **Default value:** `false`

The `imq.jmx.connector.list` property defines a set of named JMX connectors to be created at broker startup; `imq.jmx.connector.activelist` specifies which of these are to be activated. Each named connector then has its own set of properties:

```
imq.jmx.connector.connectorName.urlpath
imq.jmx.connector.connectorName.useSSL
imq.jmx.connector.connectorName.brokerHostTrusted
```

By default, two JMX connectors are created, named `jmxrmi` and `ssljmxrmi`; the first is configured not to use SSL encryption (`imq.jmx.connector.jmxrmi.useSSL` = `false`, the second to use it (`imq.jmx.connector.ssljmxrmi.useSSL` = `true`). By default, only the `jmxrmi` connector is activated at broker startup; see "SSL Support for JMX Clients" on page 23 for information on how to activate the `ssljmxrmi` connector for secure communications.

For convenience, new options (Table 1–4) are also added to the command-line Broker utility (`imqbrokerd`) to control the usage, startup, and port for the RMI registry. The use and effects of these options are the same as those of the equivalent broker properties, as described in Table 1–3. The table lists each option, specifies its equivalent broker property, and describes its use.

TABLE 1–4 New Broker Utility Options for JMX Support

Option	Equivalent Broker Property	Description
`-startRmiRegistry`	`imq.jmx.rmiregistry.start`	Specifies whether to start RMI registry at broker startup.
`-useRmiRegistry`	`imq.jmx.rmiregistry.use`	Specifies whether to use external RMI registry.
`-rmiRegistryPort`	`imq.jmx.rmiregistry.port`	The port number of RMI registry

A new subcommand (Table 1–5) is added to the command-line Command utility (`imqcmd`) for listing the JMX service URLs of JMX connectors created and started at broker startup. This information is needed by JMX clients that do not use the Message Queue convenience class `AdminConnectionFactory` to obtain their JMX connectors, and can also be used for managing or monitoring Message Queue via a generic JMX browser such as the Java Monitoring and Management Console (`jconsole`).

TABLE 1–5 New Command Utility Subcommand

Subcommand	Description
`list jmx`	List JMX service URLs of JMX connectors

SSL Support for JMX Clients

As mentioned above, a Message Queue message broker is configured by default for insecure communication using the preconfigured JMX connector `jmxrmi`. Applications wishing to use the Secure Socket Layer (SSL) for secure communication must activate the alternate, secure JMX connector, `ssljmxrmi`. This requires the following steps:

1. Obtain and install a signed certificate in the same way as for the `ssljms`, `ssladmin`, or `cluster` connection service, as described in the *Message Queue Administration Guide*.
2. Install the root certification authority certificate in the trust store if necessary.
3. Add the `ssljmxrmi` connector to the list of JMX connectors to be activated at broker startup:

   `imq.jmx.connector.activelist=jmxrmi,ssljmxrmi`
4. Start the broker with the Message Queue Broker utility (`imqbrokerd`), either passing it the key-store password in a password file or typing it from the command line when prompted.
5. By default, the `ssljmxrmi` connector (or any other SSL-based connector) is configured to validate all broker SSL certificates presented to it. To avoid this validation (for instance, when using self-signed certificates during software testing), set the broker property `imq.jmx.connector.ssljmxrmi.brokerHostTrusted` to `true`.

On the client side, the administrator connection factory (`AdminConnectionFactory`) must be configured with a URL specifying `ssljmxrmi` as the preferred connector:

```
AdminConnectionFactory  acf = new AdminConnectionFactory();
acf.setProperty(AdminConnectionConfiguration.imqAddress, "mq://myhost:7676/ssljmxrmi");
```

If needed, use the system properties `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` to point the JMX client to the trust store.

Client Runtime Logging

This section describes Message Queue 4.0 support for client runtime logging of connection and session-related events.

JDK 1.4 (and above) includes the `java.util.logging` library. This library implements a standard logger interface that can be used for application-specific logging.

The Message Queue client runtime uses the Java Logging API to implement its logging functions. You can use all the J2SE 1.4 logging facilities to configure logging activities. For example, an application can use the following Java logging facilities to configure how the Message Queue client runtime outputs its logging information:

- Logging Handlers
- Logging Filters
- Logging Formatters
- Logging Level

For more information about the Java Logging API, please see the Java Logging Overview at `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/overview.html`

Logging Name Spaces, Levels, and Activities

The Message Queue provider defines a set of logging name spaces associated with logging levels and logging activities that allow Message Queue clients to log connection and session events when a logging configuration is appropriately set.

The root logging name space for the Message Queue client runtime is defined as `javax.jms`. All loggers in the Message Queue client runtime use this name as the parent name space.

The logging levels used for the Message Queue client runtime are the same as those defined in the `java.util.logging.Level` class. This class defines seven standard log levels and two additional settings that you can use to turn logging on and off.

`OFF`	Turns off logging.
`SEVERE`	Highest priority, highest value. Application-defined.
`WARNING`	Application-defined.
`INFO`	Application-defined.
`CONFIG`	Application-defined
`FINE`	Application-defined.
`FINER`	Application-defined.
`FINEST`	Lowest priority, lowest value. Application-defined.
`ALL`	Enables logging of all messages.

In general, exceptions and errors that occur in the Message Queue client runtime are logged by the logger with the `javax.jms` name space.

- Exceptions thrown from the JVM and caught by the client runtime, such as `IOException`, are logged by the logger with the logging name space `javax.jms` at level `WARNING`.
- JMS exceptions thrown from the client runtime, such as `IllegalStateException`, are logged by the logger with the logging name space `javax.jms` at level `FINER`.
- Errors thrown from the JVM and caught by the client runtime, such as `OutOfMemoryError`, are logged by the logger with the logging name space `javax.jms` at level `SEVERE`.

The following tables list the events that can be logged and the log level that must be set to log events for JMS connections and for sessions.

The following table describes log levels and events for connections.

TABLE 1–6 Log Levels and Events for `javax.jms.connection` Name Space

Log Level	Events
`FINE`	Connection created
`FINE`	Connection started
`FINE`	Connection closed
`FINE`	Connection broken
`FINE`	Connection reconnected
`FINER`	Miscellaneous connection activities such as `setClientID`
`FINEST`	Messages, acknowledgments, Message Queue action and control messages (like committing a transaction)

For sessions, the following information is recorded in the log record.

- Each log record for a message delivered to a consumer includes ConnectionID, SessionID, and ConsumerID.
- Each log record for a message sent by a producer includes ConnectionID, SessionID, ProducerID, and destination name.

The table below describes log levels and events for sessions.

TABLE 1–7 Log Levels and Events for `javax.jms.session` Name Space

Log Level	Event
`FINE`	Session created
`FINE`	Session closed
`FINE`	Producer created
`FINE`	Consumer created
`FINE`	Destination created
`FINER`	Miscellaneous session activities such as committing a session.
`FINEST`	Messages produced and consumed. (Message properties and bodies are not logged in the log records.)

By default, the output log level is inherited from the JRE in which the application is running. Check the `JRE_DIRECTORY/lib/logging.properties` file to determine what that level is.

You can configure logging programmatically or by using configuration files, and you can control the scope within which logging takes place. The following sections describe these possibilities.

Using the JRE Logging Configuration File

The following example shows how you set logging name spaces and levels in the `JRE_DIRECTORY/lib/logging.properties` file, which is used to set the log level for the Java runtime environment. All applications using this JRE will have the same logging configuration. The sample configuration below sets the logging level to `INFO` for the `javax.jms.connection` name space and specifies that output be written to `java.util.logging.ConsoleHandler`.

```
#logging.properties file.
# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.

    handlers= java.util.logging.ConsoleHandler

# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overriden by a facility-specific level.
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.

    .level= INFO

# Limit the messages that are printed on the console to INFO and above.

    java.util.logging.ConsoleHandler.level = INFO
    java.util.logging.ConsoleHandler.formatter =
                                    java.util.logging.SimpleFormatter

# The logger with javax.jms.connection name space will write
# Level.INFO messages to its output handler(s). In this configuration
# the ouput handler is set to java.util.logging.ConsoleHandler.

    javax.jms.connection.level = INFO
```

Using a Logging Configuration File for a Specific Application

You can also define a logging configuration file from the java command line that you use to run an application. The application will use the configuration defined in the specified logging file. In

the following example, `configFile` uses the same format as defined in the `JRE_DIRECTORY/lib/logging.properties` file.

```
java -Djava.util.logging.config.file=configFile MQApplication
```

Setting the Logging Configuration Programmatically

The following code uses the `java.util.logging` API to log connection events by changing the `javax.jms.connection` name space log level to `FINE`. You can include such code in your application to set logging configuration programmatically.

```
import java.util.logging.*;
//construct a file handler and output to the mq.log file
//in the system's temp directory.

    Handler fh = new FileHandler("%t/mq.log");
    fh.setLevel (Level.FINE);

//Get Logger for "javax.jms.connection" domain.

    Logger logger = Logger.getLogger("javax.jms.connection");
    logger.addHandler (fh);

//javax.jms.connection logger would log activities
//with level FINE and above.

    logger.setLevel (Level.FINE);
```

Connection Event Notification

Connection event notifications allow a Message Queue client to listen for closure and reconnection events and to take appropriate action based on the notification type and the connection state. For example, when a failover occurs and the client is reconnected to another broker, an application might want to clean up its transaction state and proceed with a new transaction.

If the Message Queue provider detects a serious problem with a connection, it calls the connection object's registered exception listener. It calls the listener's `onException` method, and passes it a `JMSException` argument describing the problem. The Message Queue provider also offers an event notification API that allows the client runtime to inform the application about connection state changes. The notification API is defined by the following elements:

- The `com.sun.messaging.jms.notification` package, which defines the event listener and the notification event objects .
- The `com.sun.messaging.jms.Connection` interface, which defines extensions to the `javax.jms.Connection` interface.

The following sections describe the events that can trigger notification and explain how you can create an event listener.

Connection Events

The following table lists and describes the events that can be returned by the event listener.

Note that the JMS exception listener is not called when a connection event occurs. The exception listener is only called if the client runtime has exhausted its reconnection attempts. The client runtime always calls the event listener before the exception listener.

TABLE 1–8 Notification Events

Event Type	Meaning
`ConnectionClosingEvent`	The Message Queue client runtime generates this event when it receives a notification from the broker that a connection is about to be closed due to a shutdown requested by the administrator.
`ConnectionClosedEvent`	The Message Queue client runtime generates this event when a connection is closed due to a broker error or when it is closed due to a shutdown or restart requested by the administrator. When an event listener receives a `ConnectionClosedEvent`, the application can use the `getEventCode()` method of the received event to get an event code that specifies the cause for closure.
`ConnectionReconnectedEvent`	The Message Queue client runtime has reconnected to a broker. This could be the same broker to which the client was previously connected or a different broker. An application can use the `getBrokerAddress` method of the received event to get the address of the broker to which it has been reconnected.
`ConnectionReconnectFailedEvent`	The Message Queue client runtime has failed to reconnect to a broker. Each time a reconnect attempt fails, the runtime generates a new event and delivers it to the event listener. The JMS exception listener is not called when a connection event occurs. It is only called if the client runtime has exhausted its reconnection attempts. The client runtime always calls the event listener before the exception listener.

Creating an Event Listener

The following code example illustrates how you set a connection event listener. Whenever a connection event occurs, the event listener's `onEvent` method will be invoked by the client runtime.

```
//create an MQ connection factory.

com.sun.messaging.ConnectionFactory factory =
        new com.sun.messaging.ConnectionFactory();

//create an MQ connection.

com.sun.messaging.jms.Connection connection =
       (com.sun.messaging.jms.Connection )factory.createConnection();

//construct an MQ event listener.  The listener implements
//com.sun.messaging.jms.notification.EventListener interface.

com.sun.messaging.jms.notification.EventListener eListener =
       new ApplicationEventListener();

//set event listener to the MQ connection.

connection.setEventListener ( eListener );
```

Event Listener Examples

In this example, an application chooses to have its event listener log the connection event to the application's logging system:

```
public class ApplicationEventListener implements
                com.sun.messaging.jms.notification.EventListener {

public void onEvent ( com.sun.messaging.jms.notification.Event connEvent ) {
          log (connEvent);
}
private void log ( com.sun.messaging.jms.notification.Event connEvent ) {
          String eventCode = connEvent.getEventCode();
          String eventMessage = connEvent.getEventMessage();
          //write event information to the output stream.
         }
}
```

Hardware and Software Requirements

For hardware and software requirements for Version 4.0, please consult the Release Notes for the Sun Java System Application Server Platform Edition 9.

Bugs Fixed in This Release

The following table shows bugs that were fixed in the 4.1 version of Message Queue.

TABLE 1–9 Bugs Fixed in Message Queue 4.1

Bug	Description
6381703	Transacted remote messages can be committed twice if the broker originating the message restarts.
6388049	Cannot clean up an uncompleted distributed transaction.
6401169	The commit and rollback options for imqcmd do not prompt for confirmation.
6473052	Default for autocreated queues should be round robin. (`MaxNumberConsumers` = -1).
6474990	Broker log shows `ConcurrentModificationException` for `imqcmd list dst` command.
6487413	Memory leak when limit behavior is `REMOVE_OLDEST` or `REMOVE_LOWER_PRIORITY`.
6488340	Broker spins, and client waits for reply to acknowledge.
6502744	Broker does not honor the dead message queue's default limit of 1000 messages.
6517341	Client runtime needs to improve reconnect logic when the client is connected to a high availability cluster by allowing the client to reconnect no matter what the value of the *imqReconnectEnabled* property is.
6528736	Windows automatic startup service (*imqbrokersvc*) crashes during startup.
6561494	Messages are delivered to the wrong consumer when both share a session.
6567439	Produced messages in a `PREPARED` transaction are delivered out of order if they are committed after broker restarts.

The following table describes the bugs fixed in Message Queue 4.0.

TABLE 1–10 Bugs Fixed in Message Queue 4.0

Bug Number	Description
4986481	In Message Queue 3.5, calling `Session.recover` could hang in auto-reconnect mode.

TABLE 1–10 Bugs Fixed in Message Queue 4.0 *(Continued)*

Bug Number	Description
4987325	Redelivered flag was set to `false` for redelivered messages after calling `Session.recover`.
6157073	Change new connection message to include the number of connections on the service in addition to the total number of connections.
6193884	Message Queue outputs garbage message to syslog in locales that use non-ASCII characters for messages.
6196233	Message selection using `JMSMessageID` doesn't work.
6251450	`ConcurrentModificationException` on `connectList` during cluster shutdown.
6252763	`java.nio.BufferOverflowException` in `java.nio.HeapByteBuffer.putLong/Int`.
6260076	First message published after startup is slow with Oracle storage.
6260814	Selector processing on `JMSXUserID` always evaluates to `false`.
6264003	The queue browser shows messages that are part of transactions that have not been committed.
6271876	Connection Flow Control does not work properly when closing a consumer with unconsumed messages.
6279833	Message Queue should not allow two brokers to use the same jdbc tables.
6293053	Master broker does not start up correctly if the system's IP address is changed, unless the store is cleared (using `–reset store`.)
6294767	Message Queue broker needs to set `SO_REUSEADDR` on the network sockets it opens.
6304949	Unable to set `ClientID` property for `TopicConnectionFactory`.
6307056	The txn log is a performance bottleneck.
6320138	Message Queue C API lacks ability to determine the name of a queue from a reply-to header.
6320325	The broker sometimes picks up JDK 1.4 before JDK 1.5 on Solaris even if both versions are installed.
6321117	Multibroker cluster initialization throws `java.lang.NullPointerException`.
6330053	The jms client throws `java.lang.NoClassDefFoundError` when committing a transaction from the subscriber.
6340250	Support `MESSAGE` type in C-API.
6351293	Add Support for Apache Derby database.

Important Information

This section contains the latest information that is not contained in the core product documentation. This section covers the following topics:

Installation Notes

Refer to the *Sun Java System Message Queue 4.1 Installation Guide*for information about pre-installation instructions, upgrade procedures, and all other information relevant to installing Message Queue, Platform Edition on the Solaris, Linux, and Windows platforms.

Refer to the *Sun Java Enterprise System Installation Guide* for information about pre-installation instructions and all other information relevant to installing Message Queue, Enterprise Edition on the Solaris, Linux, and HPUX platforms.

Refer to the *Sun Java Enterprise System Upgrade and Migration Guide* for information about upgrade and migration instructions relevant to upgrading to Message Queue Enterprise Edition on the Solaris, Linux, HPUX, and Windows platforms.

Compatibility Issues

This section covers compatibility issues in Message Queue 4.1.

Interface Stability

Sun Java System Message Queue uses many interfaces that may change over time. Appendix B, "Stability of Message Queue Interfaces," in *Sun Java System Message Queue 4.1 Administration Guide* classifies the interfaces according to their stability. The more stable an interface, the less likely it is to change in subsequent versions of the product.

Issues Related to the Next Major Release of Message Queue

The next major release of Message Queue may introduce changes that make your clients incompatible with that release. This information is provided now to allow you to prepare for these changes.

- The locations of individual files installed as part of Sun Java System Message Queue might change. This could break existing applications that depend on the current location of certain Message Queue files.
- 3.5 and earlier brokers may no longer be able to operate in a cluster with newer brokers.

- In future releases, Message Queue clients may not be able to use JDK versions that are earlier than 1.5.

Documentation Updates for Message Queue 4.1

Other than this *Release Notes* document, Message Queue 4.1 includes only one new document: *Sun Java System Message Queue 4.1 Developer's Guide for JMX Clients*. This document was introduced with the 4.0 release of Message Queue. In the 4.1 version, conceptual information has been added that introduces the JMX model.

The Message Queue documentation that was published for Message Queue 3.6 SP3, 2005Q4, is up to date with respect to the needs of Application Server 9 PE clients. This documentation set is available at the following location.

`http://docs.sun.com/app/docs/coll/1307.1`

Installation and Upgrade Information

The *Sun Java System Message Queue 4.1 Installation Guide* was updated to reflect platform-specific information. This document now contains installation and upgrade information relevant to Message Queue 4.1.

Administration Guide

The *Administration Guide* was updated to provide information about high availability clusters, JAAS support, and JMX support.

Developer's Guide for Java Clients

The*Developer's Guide for Java Clients* was updated to reflect the addition of client runtime logging support and of connection event notifications.

Developer's Guide for C Clients

The*Developer's Guide for C Clients* was updated to reflect the addition of the `MQGetDestinationName` function, of the `MQ_Message` message type, and of fixed ports.

Known Issues and Limitations

This section contains a list of the known issues with Message Queue 4.1. The following product areas are covered:

For a list of current bugs, their status, and workarounds, Java Developer Connection™ members should see the Bug Parade page on the Java Developer Connection web site. Please check that page before you report a new bug. Although all Message Queue bugs are not listed, the page is a good starting place if you want to know whether a problem has been reported.

`http://bugs.sun.com/bugdatabase/index.jsp`

Note – Java Developer Connection membership is free but requires registration. Details on how to become a Java Developer Connection member are provided on Sun's "For Developers" web page.

To report a new bug or submit a feature request, send mail to `imq-feedback@sun.com`.

Installation Issues

This section describes issues related to the installation of Message Queue version 4.1.

Product Registry and JES

Version 4.1 of Message Queue is installed by a new installer, which also installs and upgrades the shared components that Message Queue needs; for example, JDK, NSS libraries, JavaHelp, and so on. This installer and the Java Enterprise System (JES) installer do not share the same product registry. If a version of Message Queue that was installed with JES is removed and upgraded to Message Queue 4.1 by the Message Queue installer, the JES product registry may be in an inconsistent state. As a result, when the JES uninstaller is run, it may inadvertently remove Message Queue 4.1 and the shared components upon which it depends, which it did not install.

The best way to upgrade software that was installed by the JES installer is as follows.

1. Use the JES uninstaller to remove Message Queue and its shared components.

2. Use the Message Queue installer to install Message Queue 4.1.

Selecting the Appropriate JRE

The Message Queue 4. 1 Installer JDK Selection Screen allows you to select existing JDK/JRE's on the system for use by Message Queue. Unfortunately, the list shown also includes the JRE used to run the installer application. This JRE is part of the installer bundle and is not really installed on the system. *(Bug 6585911)*

The JRE used by the installer is recognizable by its path, which should be within the unzipped installer directory and should include the subdirectory `mq4_1–installer`. For example:

some_directory`/mq4_1–installer/usr/jdk/instances/jdk1.5.0/jre`

Do not select this JRE for use by Message Queue. Instead, select another JDK on the system. If one does not exist, take the action appropriate for your platform.

- Solaris or Linux: Select "Install and use the default JDK".
- Windows: Download and install a JDK before running the Message Queue 4.1 installer.

Installing on Windows

When installing Message Queue on Windows, please note the following limitations.

- The installer does not add entries for Message Queue to the Start>Programs menu (*Bug 6567258*). To start the administration console use the command line as shown in "Starting the Administration Console" in *Sun Java System Message Queue 4.1 Administration Guide*.
- The installer does not add the `IMQ_HOME\mq\bin` directory to the `PATH` environment variable.*(Bug 6567197)*. Users need either to add this entry to their `PATH` environment variable or provide a full path name when invoking Message Queue utilities (`IMQ_HOME\mq\bin\`*command*).
- The installer does not add entries to the Windows registry to indicate that Message Queue is installed.
- When run in silent mode, the installer returns right away. The installation does happen; but the user has no way of knowing when the silent installation is actually done. *(Bug 6586560)*
- Text mode (`installer –t`) is not supported on Windows. Running the installer in text mode on Windows causes an error message to be displayed. This message is displayed in English even when the installer is run in non-English locales. *(Bug 6594142)*
- The string "Install Home" displayed on the Installer Install Home screen is shown in English even when the installer is run in non-English locales. *(Bug 6592491)*

Installing on Solaris

The error message and "incomplete" summary status misleads user trying to install using the `installer-n` command. The command actually succeeds. *(Bug 6594351)*

Installing on Linux

The following issues affect installation on the Linux Platform

- On the JDK Selection panel, the scroll list displays only one item. This makes it difficult to select other JDK's in the list. *(Bug 6584735)*
- If the JDK is current and the user selects "Install default JDK" on the JDK Selection Screen, the installer still tries to install it and reports that it cannot install the package. Installation completes successfully despite this issue. *(Bug 6581310)*
- When the installer is run in dry run mode (`installer –n`), the Summary Screen shows some error messages and also displays an install status of "Incomplete". This is incorrect and misleading; a dry run does not install anything on the system; it only creates the answer file that can be subsequently used to install.*(Bug 6594351)*
- If older versions of Message Queue localization RPM's exist on your system, installation of Message Queue 4.1 localization RPM's (which happens when you select the "Install Message Queue multilingual packages" checkbox on the Multilingual Packages screen) will fail. Installation fails because of conflicts with Il8 packages from a previous 3.7 UR1 installation. *(Bug 6594381)*

  Workaround Manually remove the localization RPM's using the `rpm –e` command before running the 4.1 Installer. To determine which RPM's are relevant here, see "Message Queue Packages (RPMs)" in *Sun Java System Message Queue 4.1 Installation Guide*.

Installing on All Platforms

These issues affect installation on all platforms.

- When the Installer is in the process of installing Message Queue 4.1 and the Progress screen is displayed, the Cancel button is active. Selecting the Cancel button at this time results in incomplete or broken installs. *(Bug 6595578)*
- The Installer Summary Screen contains a number of links that when clicked will launch a log or summary page viewer. If you dismiss this viewer window using the window close button "X" instead of the button labelled "close', you will not be able to bring this viewer window back up. *(Bug 6587138)*

  Workaround Use the button labeled Close to close the window.
- When a system has older versions of Message Queue and NSS/NSPR, the Installer's Upgrade only lists Message Queue needing upgrade; it does not mention that NSS/NSPR need to be upgraded. This only an issue with the Update screen as all the relevant software will be upgraded as part of the installation process (as indicated by The ReadyToInstall screen which shows the correct information). *(Bug 6580696)*

  Workaround None needed as the NSS/NSPR files are installed if they are not current, and the older versions are unistalled.

- When the Installer or Uninstaller is run in text mode (`installer –t`), the Summary screen shows the directory containing the log/summary files but does not list the names of these files. *(Bug 6581592)*
- Specifying the name of a file that does not exist, produces inconsistent and unclear error messages. *(Bug 6587127)*

Version Information

The installer displays Message Queue version information in an opaque form. *(Bug 6586507)*

On the Solaris platform, refer to the table below to determine the version being installed.

TABLE 1–11 Version Formats

Version as Displayed by the Installer	Message Queue Release
4.1.0.0	4.1
3.7.0.1	3.7 UR1
3.7.0.2	3.7 UR2
3.7.0.3	3.7 UR3
3.6.0.0	3.6
3.6.0.1	3.6 SP1
3.6.0.2	3.6 SP2
3.6.0.3	3.6 SP3
3.6.0.4	3.6 SP4

Note – For Patch releases to 3.6 SP4 (for example, 3.6 SP4 Patch 1), the releases string displayed by the installer stays the same. You need to run the command `imqbrokerd –version` to determine the exact version.

On the Linux platform, it is not possible to provide a simple format translation. The version number displayed by the installer on Linux is in the following form.

```
<majorReleaseNumber>.<minorReleaseNumber>-<someNumber>
```

For example, 3.7–22. This tells us that it is one of the 3.7 releases, but not which specific one. To determine that, run the command `imqbrokerd –version`.

Localization Issues

The following issues relate to localization problems.

- When the installer is run in text mode (`installer –t`), in a non-English locale, multi-byte characters show up as garbage.*(Bug 6586923)*
- The Installer Summary screen allows the user to view a Summary report. Unfortunately, this report (an HTML page) shows garbage when the installer is run in multibyte locales. *(Bug 6587112)*

  Workaround Edit the HTML file to correct the charset specified in it. The HTML file should contain something like the following.

  ```
  meta http-equiv="Content-Type" content="text/html; charset=UTF-8
  ```

  Replace "UTF-8" with *locale_name*`.UTF-8`. For example, `ja_JA.UTF-8` or `ko.UTF-8` on Solaris; `ja_JA.utf8` or `ko_KO.utf8` on linux.
- On the Installer Progress screen, the progress bar shows strange characters. The tooltip is hard coded in non-English locales. *(Bug 6591632)*
- Text mode (`installer –t`) is not supported on Windows. Running the installer in text mode on Windows will cause an error message to be displayed. This message is not localized when the installer is run in non-English locales. *(Bug 6594142)*
- The License screen of the installer displays English license text no matter which locale the Installer is run in.*(Bug 6592399)*

  Workaround To access localized license files, look for at the `LICENSE_MULTILANGUAGE.pdf` file.
- Installer usage help text is not localized. *(Bug 6592493)*
- The string "None" that is seen on the Installer summary HTML page is hard coded in English. *(Bug 6593089)*
- The copyright page is not localized for locales other than France. *(Bug 6590992)*
- When the installer is run in a German locale, the Welcome screen does not show the complete text that is seen in other locales. *(Bug 6592666)*
- The string "Install Home" seen on the Installer Install Home screen is not localized. It appears in English even when the installer is run in non-English locales. *(Bug 6592491)*
- When the installer is run in text mode (`installer –t`), the English response choices "Yes" and "No" are used no matter what locale the installer is run in. *(Bug 6593230)*
- The tooltip for the browse button on the Installer JDK Selection screen is hard coded in English. *(Bug 6593085)*

Deprecated Password Option

In previous versions of Message Queue, you could use the –p or –password option to specify a password interactively for the following commands: imqcmd, imqbrokerd, and imdbmgr. Beginning with version 4.0, these options have been deprecated. You must furnish passwords as follows.

1. Set the password property to the desired value in a file used to store only passwords.

   Use the following syntax to specify passwords in the password file.

   PasswordPropertyName=MyPassword

2. Pass the name of the password file using the –passfile option.

A password file can contain one or more of the passwords listed below.

- A keystore password used to open the SSL keystore. Use the imq.keystore.password property to specify this password.
- An LDAP repository password used to connect securely with an LDAP directory if the connection is not anonymous. Use the imq.user_repository.ldap.password property to specify this password.
- A JDBC database password used to connect to a JDBC-compliant database. Use the imq.persist.jdbc.vendorName.password property to specify this password. The *vendorName* component of the property name is a variable that specifies the database vendor. Choices include hadb, derby, pointbase, oracle, or mysql.
- A password to the imqcmd command (to perform broker administration tasks). Use the imq.imqcmd.password property to specify this password.

In the following example, the password to the JDBC database is set to abracadabra.

```
imq.persist.jdbc.mysql.password=abracadabra
```

You can configure the broker to use the password file you create in one of the following ways.

- Set the following properties in the broker's config.properties file.

  ```
  imq.passfile.enabled=true
  imq.passfile.dirpath=MyFileDirectory
  imq.passfile.name=MyPassfileName
  ```

- Use the –passfile option of the imqbrokerd command.

  imqbrokerd –passfile *MyPassfileName*

General Issues

This section covers general issues in Message Queue 4.1. Some of these were introduced with previous Message Queue versions.

- When a JMS client using the HTTP transport terminates abruptly (for example, using `Ctrl-C`) the broker takes approximately one minute before releasing the client connection and all the associated resources.

  If another instance of the client is started within the one minute period and if it tries to use the same ClientID, durable subscription, or queue, it might receive a "Client ID is already in use" exception. This is not a real problem; it is just the side effect of the termination process described above. If the client is started after a delay of approximately one minute, everything should work fine.
- SOAP clients. Previously the SAAJ 1.2 implementation jar used to refer to `mail.jar` and `mail.jar` did not need to be in `CLASSPATH`. In SAAJ 1.3 this reference was removed; thus, Message Queue clients must put `mail.jar` explicitly in `CLASSPATH`.

Administration/Configuration Issues

The following issues pertain to administration and configuration of Message Queue

- The `imqadmin` and `imqobjmgr` utilities throw an error when the `CLASSPATH` contains double quotes on Windows machines (*Bug ID 5060769*).

  Workaround You can ignore this error message; the broker correctly handles notifying consumers of any error. This error does not affect the reliability of the system.
- The `-javahome` option in all Solaris and Windows scripts does not work if the value provided contains a space (*Bug ID 4683029*).

  The `javahome` option is used by Message Queue commands and utilities to specify an alternate Java 2 compatible runtime to use. However, the path name to the alternate Java runtime must not contain spaces. The following are examples of paths that include spaces.

  Windows: `C:/jdk 1.4`

  Solaris: `/work/java 1.4`

  Workaround Install the Java runtime at a location or path that does not contain spaces.
- The `imqQueueBrowserMaxMessagesPerRetrieve` attribute specifies the maximum number of messages that the client runtime retrieves at one time when browsing the contents of a queue. Note that the client application will always get all the messages on the queue. Thus, the `imqQueueBrowserMaxMessagesPerRetrieve` attribute affects how the queued messages are chunked, to be delivered to the client runtime (fewer large chunks, or more smaller chunks), but it does not affect the total messages browsed. Changing the value of this attribute might impact performance, but it will not result in the client application getting more or less data (*Bug ID 6387631*).

Broker Issues

The following issues affect the Message Queue broker.

- There has been some confusion about how to configure the broker for round-robin delivery. The solution is simple and configurable.
  1. Set the destination attribute `maxNumActiveConsumers` to -1. This turns on round-robin delivery.
  2. Set the destination attribute `consumerFlowLimit` to 1. This specifies the number of messages delivered to a single consumer before delivery progresses to the next consumer. For different chunking, set this attribute to the desired value. By default, one hundred messages are delivered to each consumer.
- Broker becomes inaccessible when persistent store opens too many destinations. (*Bug ID 4953354*).

  Workaround This condition is caused by the broker reaching the system open-file descriptor limit. On Solaris and Linux use the `ulimit` command to increase the file descriptor limit.
- Consumers are orphaned when a destination is destroyed (*Bug ID 5060787*).

  Active consumers are orphaned when a destination is destroyed. Once the consumers have been orphaned, they will no longer receive messages (even if the destination is recreated).

  Workaround There is no workaround for this problem.

Broker Clusters

The following issues affect clustered brokers.

- Only fully-connected broker clusters are supported in this release. This means that every broker in a cluster must communicate directly with every other broker in the cluster. If you are connecting brokers using the `imqbrokerd -cluster` command line argument, be careful to ensure that all brokers in the cluster are included.
- A broker using HADB cannot handle messages larger than 10 MB. *(Bug 6531734)*
- If a client is connected to a high availability broker, the client runtime will attempt to reconnect until it succeeds (no matter what the `imqAddressListIterations` value is set to be.)
- A client connected to a broker that is part of a cluster cannot currently use `QueueBrowser` to browse queues that are located on remote brokers in that cluster. The client can only browse the contents of queues that are located on the broker to which it is directly connected. The client may still send messages to any queue or consume messages from any queue on any broker in the cluster; the limitation only affects browsing.

- In a conventional cluster, if you want to cluster a 4.1 broker with a 3.x broker, you must set the property `imq.autocreate.queue.maxNumActiveConsumers=1` for the 4.1 broker. Otherwise, the brokers will not be able to establish a cluster connection.
- When converting to a high-availability cluster, you can use the Message Queue Manager utility (imqdbmgr) to convert an existing standalone HADB persistent data store to a shared HADB store. The command is as follows.

  ```
  imqdbmgr upgrade hastore
  ```

  You can use this utility in the following cases.

  - Moving from a 4.0 standalone HADB store to a 4.1 shared HADB store. In this case, the broker will automatically upgrade the store. You can then run the `imqdbmgr` command to convert the upgraded data store for shared use.
  - Moving from a standalone 4.1 HADB store to a shared HADB store. In this case, you just need to run the `imqdbmgr` command shown above to convert the data store for shared use.

  Because this command only supports conversion of HADB stores, it is not possible to use it to convert file-based stores or other JDBC-stores to a shared HADB store. If you were previously running a 3.x version of Message Queue, you must create an HADB store and then manually migrate your data to that store in order to use the high availability feature.
- The conversion to an HADB store using the command `imqdbmgr upgrade hastore` can fail with the message "too many locks are set" if the store holds more than 10,000 message. (*Bug ID 6588856)*).

  (*Workaround*) Use the following command to increase the number of locks.

  ```
  hadbm set NumberOfLocks=<desiredNumber>
  ```

  For additional information see "HADB Problems" in *Sun Java System Application Server 9.1 Enterprise Edition Troubleshooting Guide*.
- If more than 500 remote messages are committed in one transaction, the broker might return the error "HADB-E-12815: Table memory space exhausted." *(Bug ID 6550483)*

  For additional information, see "HADB Problems" in *Sun Java System Application Server 9.1 Enterprise Edition Troubleshooting Guide*.
- In a broker cluster, a broker will queue messages to a remote connection that has not been started (*Bug ID 4951010*).

  Workaround The messages will be received by the consumer once the connection is started. The messages will be redelivered to another consumer if the consumer's connection is closed.
- When consuming more than one message from a remote broker in one transaction, it is possible that the following error message will be logged to the broker. The message is benign and can be ignored:

```
[26/Jul/2007:13:18:27 PDT] WARNING [B2117]:
Message acknowledgement failed from
mq://129.145.130.95:7677/?instName=a&brokerSessionUID=3209681167602264320:
    ackStatus = NOT_FOUND(404)\
    Reason = Update remote transaction state to COMMITED(6):
transaction 3534784765719091968 not found, the transaction
may have already been committed.
    AckType = MSG_CONSUMED
    MessageBrokerSession = 3209681167602264320
    TransactionID = 3534784765719091968
        SysMessageID = 8-129.145.130.95(95:fd:93:91:ec:a0)-33220-1185481094690
        ConsumerUID = 3534784765719133952\par

[26/Jul/2007:13:18:27 PDT] WARNING Notify commit transaction
[8-129.145.130.95(95:fd:93:91:ec:a0)-33220-1185481094690,
[consumer:3534784765719133952, type=NONE]]
TUID=3534784765719091968 got response:
com.sun.messaging.jmq.jmsserver.util.BrokerException:
 Update remote transaction state to COMMITED(6):
  transaction 3534784765719091968 not found, the transaction may have already
  been committed.:
com.sun.messaging.jmq.jmsserver.util.BrokerException: Update remote transaction
state to COMMITED(6): transaction 3534784765719091968 not found, the transaction
 may have already been committed.r
```

This message gets logged when notifying the commit to the message home broker for later messages in the transaction when the `imq.txn.reapLimit` property is low compared to the number of remote messages in one transaction. *(Bug 6585449)*

Workaround To avoid this message increase the value of the `imq.txn.reapLimit` property.

JMX Issues

On the Windows platform, the `getTransactionInfo` method of the Transaction Manager Monitor MBean returns transaction information that has incorrect transaction creation time (*Bug ID 6393359*).

Workaround Use the `getTransactionInfoByID` method of the Transaction Manager Monitor MBean instead.

Support for SOAP

You need to be aware of two issues related to SOAP support

- Beginning with the release of version 4.0 of Message Queue, support for SOAP administered objects is discontinued.

- SOAP development depends upon several files: `SUNWjaf`, `SUNWjmail`, `SUNWxsrt`, and `SUNWjaxp`. In version 4.1 of Message Queue, these files are available to you only if you are running Message Queue with JDK version 1.6.0 or later.

Redistributable Files

Sun Java System Message Queue 4.1 contains the following set of files which you may use and freely distribute in binary form:

`fscontext.jar`
`imq.jar`
`imqjmx.jar`
`imqxm.jar`
`jaas.jar`
`jms.jar`
`libmqcrt.so` (HPUX)
`libmqcrt.so` (UNIX)
`mqcrt1.dll` (Windows)

In addition, you can also redistribute the `LICENSE` and `COPYRIGHT` files.

Accessibility Features for People With Disabilities

To obtain accessibility features that have been released since the publishing of this media, consult Section 508 product assessments (available from Sun upon request) to determine which versions are best suited for deploying accessible solutions. Updated versions of applications can be found at `http://sun.com/software/javaenterprisesystem/get.html`.

For information on Sun's commitment to accessibility, visit `http://sun.com/access`.

How to Report Problems and Provide Feedback

If you have problems with Sun Java System Message Queue, contact Sun customer support using one of the following mechanisms:

- Sun Software Support services online at `http://www.sun.com/service/sunone/software`.

  This site has links to the Knowledge Base, Online Support Center, and ProductTracker, as well as to maintenance programs and support contact numbers.
- The telephone dispatch number associated with your maintenance contract.

So that we can best assist you in resolving problems, please have the following information available when you contact support:

- Description of the problem, including the situation where the problem occurs and its impact on your operation.
- Machine type, operating system version, and product version, including any patches and other software that might be affecting the problem.

- Detailed steps on the methods you have used to reproduce the problem.
- Any error logs or core dumps.

Sun Java System Software Forum

There is a Sun Java System Message Queue forum available at the following location:

`http://swforum.sun.com/jive/forum.jspa?forumID=24`

We welcome your participation.

Java Technology Forum

There is a JMS forum in the Java Technology Forums that might be of interest.

`http://forum.java.sun.com`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions.

To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the document title and part number. The part number is a seven-digit or nine-digit number that can be found on the title page of the book or at the top of the document. For example, the title of this book is Sun Java System Message Queue 4.1 Release Notes, and the part number is 819-7753.

Additional Sun Resources

Useful Sun Java System information can be found at the following Internet locations:

- Documentation

  `http://docs.sun.com/prod/java.sys`
- Professional Services

  `http://www.sun.com/service/sunps/sunone`
- Software Products and Service

  `http://www.sun.com/software`
- Software Support Services

`http://www.sun.com/service/sunone/software`

- Support and Knowledge Base

  `http://www.sun.com/service/support/software`
- Sun Support and Training Services

  `http://training.sun.com`
- Consulting and Professional Services

  `http://www.sun.com/service/sunps/sunone`
- Developer Information

  `http://developers.sun.com`
- Sun Developer Support Services

  `http://www.sun.com/developers/support`
- Software Training

  `http://www.sun.com/software/training`