MQ Guide

JEUS 9

Open Source Software Notice

Document History

Product Version	Guide Version	Date	Remarks
JEUS 9	3.1.2	2025-03-24	-
JEUS 9	3.1.1	2024-12-24	-

Contents

1. Introduction

This chapter briefly introduces Jakarta Messaging (hereafter JMS), and describes the features of JEUS MQ.

1.1. Jakarta Messaging(JMS)

Jakarta Messaging (JMS) is a Java Standard API defined to execute inter-application communication through messaging. This section defines the necessary messaging components and message model interfaces, and describes the relationships between them.

JMS has the following features.

- Loosely coupled structure

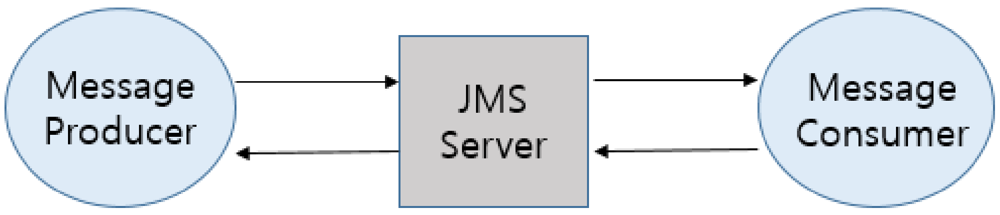
  Message producers and message consumers operate independently.

- Asynchronous communication

  Message consumers can receive messages as soon as they reach the server even if there are no requests.

- Reliable message transmission

  Messages are guaranteed to be delivered exactly once.



JMS Messaging

For more information about JMS or API usage, refer to JMS specification.

1.2. JEUS MQ Features

JEUS MQ is the implementation entity of the Jakarta Messaging specification that is included in JEUS (TmaxSoft Jakarta EE server). JEUS MQ supports non-blocking I/Os and XA transactions. It also provides highly available message services with enhanced security.

JEUS MQ supports JMS version 2.0 specification and provides the following key features.

- Supports non-blocking I/O

By using non-blocking I/O, JEUS MQ can simultaneously process more clients by using the same system resources.

- Supports XA Transaction

  Supports XA transaction that is an option defined in the JMS specification.

- Enhanced Security

  Supports the secure socket layer (SSL) communication. JEUS security can be configured to strengthen the security.

- High Availability

  Clients can automatically recover from server or network failures.

- Scalability

  Loads can be balanced by JEUS MQ clustering when the number of clients or messages increases.

2. JEUS MQ Client Programming

This chapter describes JEUS MQ client types, procedures for creating a client, and administered objects.

2.1. Overview

This section discusses JEUS MQ client types and procedures for creating a client.

Client Types

There are two types of JMS clients, message producers and message consumers, and a single client can play both roles. JEUS MQ clients can be divided into the following categories according to the methods used for Java application deployment and execution.

- Independent applications

  Independently executed in the Java Standard Edition (SE) environment.

- Jakarta EE applications

  Deployed to Jakarta EE servers, like EJBs and servlets. For information about deploying Jakarta EE applications to JEUS and executing them, refer to "JEUS Applications & Deployment Guide".

- Message Driven Bean

  A message-driven bean (MDB) is a type of EJB. It allows a single-threaded Java SE application to run by using multiple threads concurrently. For more information about MDBs, refer to EJB related books or "Jakarta EE Tutorials".

  For information about how to configure MDB in JEUS, refer to "Message Driven Bean(MDB)" in JEUS EJB Guide.

Currently, JEUS MQ does not support clients that are written in a language other than Java.

Steps for Creating a Client

The following are the steps for creating a JEUS MQ client using an independent application.

When a client program that runs in the Jakarta EE environment is specified in the XML Deployment Descriptor, the lookup process of the initial Java Naming and Directory Interface (hereafter JNDI) is not required.

1. Create a JNDI InitialContext.

```
Context context = new InitialContext();
```

The JEUS JNDI service configuration will be discussed in Defining JNDI Services.

2. Obtain a connection factory through JNDI.

```
ConnectionFactory connectionFactory =
    (ConnectionFactory) context.lookup("jms/ConnectionFactory");
```

A connection factory can be obtained by using the JEUS MQ API, without using a JNDI lookup. For more information, refer to Connection Factory.

3. Create a connection through the obtained connection factory.

```
Connection connection = connectionFactory.createConnection();
```

4. Create a session from the connection.

```
Session session = connection.createSession(false, AUTO_ACKNOWLEDGE);
```

5. Look up a destination through JNDI.

```
Destination destination = (Destination) context.lookup("ExamplesQueue");
```

Like the connection factory, a destination can be obtained using JEUS MQ API, without using a JNDI lookup. For more information, refer to Destination.

6. To send a message, create a message producer using the session object.

```
MessageProducer producer = session.createProducer(destination);
```

To receive a message, create a message consumer using the session object.

```
MessageConsumer consumer = session.createConsumer(destination);
```

To asynchronously receive the message, register the object that implements the MessageListener interface in the message consumer.

```
MessageListener listener = new MyMessageListener();
consumer.setMessageListener(listener);
```

7. Initiate the connection.

   ```
   connection.start();
   ```

8. Send or receive messages to execute the business logic.

   Send a message through the message producer.

   ```
   TextMessage message = session.createTextMessage("Hello, World");
   producer.send(msg);
   ```

   To synchronously receive messages, invoke the receive() method of the message consumer.

   ```
   Message message = consumer.receive();
   ```

   To asynchronously receive messages, the received messages are processed by the onMessage() method of the MessageListener object registered in **Step 6**.

9. After all messages have been sent and received, close the connection.

   ```
   connection.close();
   ```

2.2. JMS Administered Objects

There are two types of JMS administered objects, the connection factory and destination.

In general, they are created on the server through JMS server settings, and managed by the administrator. The JMS client obtains the objects or object references from the server in order to use the JMS Service. A JNDI lookup is a typical way for the client to obtain the JMS administered objects from the server.

This section explains how to obtain a connection factory or destination reference from the JEUS MQ server, and how to implement the JEUS JNDI service in the client programs to use JNDI lookup. In addition, it discusses how to dynamically create a destination in the JEUS MQ server by using the API, and describes the dead message destination of the JEUS MQ server.

2.2.1. Defining JNDI Services

The JNDI service should be defined first to obtain JMS administered objects with a JNDI lookup.

The JNDI service can be defined using the following three properties.

- java.naming.factory.initial

- java.naming.factory.url.pkgs
- java.naming.provider.url

The three properties can be applied in the following ways.

- **Creating a JNDI property file**

  The easiest way to define JNDI service is to create the jndi.properties file where the property values are configured.

  The following is an example of the jndi.properties file that defines the environment for using JEUS JNDI service.

  <jndi.properties>

  ```
  java.naming.factory.initial=jeus.jndi.JEUSContextFactory
  java.naming.factory.url.pkgs=jeus.jndi.jns.url
  java.naming.provider.url=127.0.0.1:9736
  ```

  To enable InitialContext objects to read the jndi.properties file settings, put the jndi.properties file in the class path, or include it in a JAR file with other files for deployment.

- **Passing the parameters to system property**

  It is difficult to modify the jndi.properties file included in the JAR file at runtime. The configuration values should be passed to the system properties when executing the client as in the following.

  ```
  java -Djava.naming.factory.initial=jeus.jndi.JEUSContextFactory \
       -Djava.naming.factory.url.pkgs=jeus.jndi.jns.url \
       -Djava.naming.provider.url=127.0.0.1:9736 \
       . . .
  ```

- **Passing the parameters as applet properties**

  If the JEUS MQ client is an applet, it is better to pass the parameters by using the <param> element inside the <applet> element in the HTML file rather than creating a jndi.properties file as in the following.

  ```
  <applet code="JeusMqApplet" width="640" height="480">
      <param name="java.naming.factory.initial"
             value="jeus.jndi.JEUSContextFactory"/>
      <param name="java.naming.factory.url.pkgs" value="jeus.jndi.jns.url"/>
      <param name="java.naming.provider.url" value="127.0.0.1:9736"/>
  </applet>
  ```

  In the applet, configure an environment that is needed to create the InitialContext using the Applet.getParameter() method.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, getParameter("java.naming.factory.initial"));
env.put(Context.URL_PKG_PREFIXES, getParameter("java.naming.factory.url.pkgs"));
env.put(Context.PROVIDER_URL, getParameter("java.naming.provider.url"));

Context context = new InitialContext(env);
```

- **Inserting the configuration value in the code**

  The environment that is needed to create the InitialContext can be directly implemented in the client code.

  Include the properties in the Hashtable object, and use them as the parameters of the InitialContext constructor.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "jeus.jndi.JEUSContextFactory");
env.put(Context.URL_PKG_PREFIXES, "jeus.jndi.jns.url");
env.put(Context.PROVIDER_URL, "127.0.0.1:9736");

Context context = new InitialContext(env);
```

However, this method may incur maintenance inconvenience because the source code needs to be modified to use other JNDI services or change the service definition.

2.2.2. Connection Factory

A connection factory has useful information on creating a connection to the JEUS MQ server.

In general, the JMS client obtains a connection factory through JNDI lookup, and uses it to create a connection with the JMS server.

```
ConnectionFactory connectionFactory =
    (ConnectionFactory) context.lookup("jms/ConnectionFactory");
QueueConnectionFactory queueConnectionFactory =
    (QueueConnectionFactory) context.lookup("jms/QueueConnectionFactory");
TopicConnectionFactory topicConnectionFactory =
    (TopicConnectionFactory) context.lookup("jms/TopicConnectionFactory");
```

To use distributed transactions in JEUS MQ, look up XAConnectionFactory, XAQueueConnectionFactory, and XATopicConnectionFactory classes as in the following.

The Jakarta EE client can obtain a connection factory by using the Resource annotation, instead of using the InitialContext.lookup() method.

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

If the JNDI Service is not available, a connection factory can be obtained by using the API provided by JEUS MQ.

```
jeus.jms.client.util.JeusConnectionFactoryCreator connectionFactoryCreator =
    new jeus.jms.client.util.JeusConnectionFactoryCreator();
connectionFactoryCreator.setFactoryName("ConnectionFactory");
connectionFactoryCreator.addServerAddress("127.0.0.1", 9741, "internal");
ConnectionFactory connectionFactory =
    (ConnectionFactory) connectionFactoryCreator.createConnectionFactory();
```

When using JNDI lookup or the Resource annotation, the JNDI name of the connection factory object (e.g., "jms/ConnectionFactory" in the example) is used to obtain a connection factory. When using the JeusConnectionFactoryCreator object, the connection factory name (e.g., "ConnectionFactory" in the previous example) which is internally used by JEUS MQ server is used. For the difference between the connection factory name and JNDI name, refer to the example in Connection Factory Configuration.

2.2.3. Destination

There are two types of destinations—queue and topic—used as storage for messages managed by the server.

Queue supports one message to one client (one to one), and Topic supports one message to all clients (one to many).

The JMS client obtains a reference to the destination object managed by the server in order to receive or send a message from/to a destination

In general, the JEUS MQ client obtains a destination by looking up the JNDI name on the JNDI server where the destination references are registered.

```
Queue queue = (Queue) context.lookup("jms/ExamplesQueue");
Topic topic = (Topic) context.lookup("jms/ExamplesTopic");
```

The Jakarta EE client can obtain a destination reference by using the @Resource annotation, instead of using the InitialContext.lookup() method as in the following.

```
@Resource(mappedName="jms/ExamplesQueue")
private static Queue queue;
```

Like connection factories, JEUS MQ API can be used to obtain a destination reference without JNDI Lookup. To refer to the destination, use the destination name that is internally used by the JEUS MQ server.

```
jeus.jms.client.util.JeusDestinationCreator destinationCreator =
    new jeus.jms.client.util.JeusDestinationCreator();
destinationCreator.setDestinationName("__ExamplesQueue__");
destinationCreator.setDestinationClass(Destination.class);
Destination destination = (Destination) destinationCreator.createDestination();
```

It is also possible to use the Session.createQueue (string) and Session.createTopic (string) methods. They are implemented differently for each JMS implementation. As shown in the previous example, the actual destination name must be passed as a parameter.

```
Queue queue = session.createQueue("ExamplesQueue");
Topic topic = session.createTopic("ExamplesTopic");
```

Dynamic Creation of Destinations

JEUS MQ enables a client to dynamically create a destination on the server by using the Session.createQueue (string) and Session.createTopic (string) methods.

When specifying a destination string, append a "?" after the destination name. To use options, append each option using the form "param=value" after the "?". To set two or more options, use "&" as a delimiter. The destination parameters and values can be specified like those set in the domain.xml configuration file.

The export-name is used to dynamically create a destination.

The following example shows how to dynamically create a queue by using the destination name "DynamicQueue" and the JNDI name "jms/DynamicQueue".

```
QueueConnectionFactory queueConnectionFactory =
    (QueueConnectionFactory) context.lookup("jms/QueueConnectionFactory");
QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();
QueueSession queueSession =
    queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = queueSession.createQueue(
    "DynamicQueue?export-name=jms/DynamicQueue");
```

The API has the following features.

- If the specified destination name already exists on the server, or another object is already registered as the specified JNDI name, a JMSException is thrown. If a JNDI name is not provided, the destination name is used as the JNDI name.

- A connection needs to be created in order to enable the JEUS MQ server security function. If the subject used for creating a connection does not have "createDestination" permission on the "jeus.jms.destination.creation" resource, a JMSException is thrown.
- Other settings follow the basic settings of the destination.
- The dynamically created destinations are deleted when the JEUS MQ server shuts down.

Destinations created in the above example operate in the same way as destinations depending on settings until JEUS MQ server shuts down.

A destination cannot be dynamically created when JEUS MQ clustering is enabled.

Dead Message Destination

Dead message destination is a system destination where failed messages are saved. A message may fail if the destination of the message could not be found, or the message has been recovered more than the allowed number of times.

The following are the cases when a message that is received by the consumer is recovered to the server-side destination.

- The message consumer calls Session.recover(), etc.
- An exception occurs while executing the onMessage() method of the message listener.

The recovered message gets re-sent to the message consumer. However, if message processing fails repeatedly due to a business logic or client program error, messages will accumulate at the destination and the client cannot receive them. To resolve this problem, JEUS MQ limits the number of times that a message can be recovered. Set a value in the "JMS_JEUS_RedeliveryLimit" message property variable, and send the message.

```
message.setIntProperty("JMS_JEUS_RedeliveryLimit", <integer value>)
```

You can change the default message property value by setting the "jeus.jms.client.default-redelivery-limit" system property when executing the message producer client.

```
-Djeus.jms.client.default-redelivery-limit=<integer value>
```

If not set, 3 is used as the default value. When booting, the JEUS MQ server creates a dead message destination with the name "JEUSMQ_DLQ". In general, the messages saved in the dead message destination are processed using the system administration tool. General clients can also access and process the dead message destination by using this name.

Too many messages in a dead message destination may cause an OutOfMemoryError. To avoid this error, you can specify a client that will receive messages, or remove messages in the dead message destination periodically by using a system management tool.

2.3. Connections and Sessions

When a JMS client exchanges messages with a server, it must connect to the server and create a session. The connections and the sessions are very important client resources in JMS that may affect the system performance and determine error response methods. JEUS MQ provides a number of options to enhance connection performance.

This section describes how to set the connection options and how the JEUS MQ client runs based on the specified options.

2.3.1. Creating Connections

Connection is the first object that a client creates to work with a JMS server, and is physically or logically linked with the JMS server. Connections are created by calling the createConnection() method of the connection factory.

```
public Connection createConnection()
                            throws JMSException;
public Connection createConnection(String userName, String password)
                            throws JMSException;
```

If the JEUS MQ server uses the security function and the subject that has the userName/password does not have "createConnection" permission on the "jeus.jms.client.connectionFactory" resource, a JMSException is thrown.

The following option can be set as a system property when executing the JEUS MQ client.

```
-Djeus.jms.client.connect.timeout=<long value>
```

This option specifies the amount of time that the ConnectionFactory.createConnection() method waits for a successful connection. The default value is 5 seconds, and the unit is in milliseconds. If not connected within this time, a JMSException occurs in the ConnectionFactory.createConnection() method. If set to "0", it waits for a successful connection indefinitely.

This setting can be modified at runtime by adding either of the following statements to the client code:

```
System.setProperty("jeus.jms.client.connect.timeout", <long value>);
```

or

```
System.setProperty(jeus.jms.common.JeusJMSProperties.KEY_CLIENT_CONNECT_TIMEOUT, <long value>);
```

2.3.2. Sharing Physical Connections

An application sends a message using the following steps.

1. Create a connection
2. Create a session
3. Create a message producer
4. Send a message
5. Close a connection

If a JMS connection can be connected to only one physical connection (socket), performance is reduced because a new physical connection has to be created whenever a message is sent. Creating a physical connection takes longer than sending a message. The more physical connections are created, the more file descriptors are used which may cause IOExceptions to occur and disrupt system stability.

In a JEUS 6 Fix#6 and later, physical connections can be shared to resolve performance and stability problems mentioned previously. However, in some environments, since it would be better to create a new physical connection each time instead of sharing connections, an option is provided to set this function.

In general, one physical connection is connected to each ConnectionFactory, but if JEUS MQ failover is configured or "jeus.jms.client.use-single-server-entry" is set to "false", one physical connection is used for each connection. For more information about JEUS MQ Failover, refer to JEUS MQ Failover.

The following are JVM options related to physical connection sharing.

-
  ```
  -Djeus.jms.client.use-single-server-entry=<boolean value>
  ```

  This option determines whether to create one physical connection for each ConnectionFactory and share it with other connections. The default value is true. If set to false, the physical connection is used for only one connection.

  This setting can be modified at runtime by adding either of the following statements to the client code:

  ```
  System.setProperty("jeus.jms.client.use-single-server-entry", <boolean value>);
  ```

  or

```
System.setProperty(jeus.jms.common.JeusJMSProperties.USE_SINGLE_SERVER_ENTRY,
                                    <boolean value>);
```

-
  ```
  -Djeus.jms.client.single-server-entry.shutdown-delay=<long value>
  ```

  This option determines when to disconnect the physical connection if it is not in use. Idle physical connections are not maintained but returned to the system. (Unit: milliseconds, default value: 600,000 (10 minutes))

  This setting can be modified at runtime by adding either of the following statements to the client code:

  ```
  System.setProperty("jeus.jms.client.single-server-entry.shutdown-delay", <long value>);
  ```

  or

  ```
  System.setProperty(jeus.jms.common.JeusJMSProperties.SINGLE_SERVER_ENTRY_SHUTDOWN_DELAY, <long
  value>);
  ```

The JMS specification has a restriction on closing connections in onMessage of MessageListener as this may cause a deadlock.

2.3.3. Creating Sessions

JMS session is the basic unit for all messaging tasks such as sending a created message to the destination or receiving a message from the destination. A session is also a unit that is used for a JMS client that participates in a local or XA transaction.

A session and all tasks processed in the session need to be processed by a single thread context, and this means that the session objects are not thread-safe.

The JEUS MQ client does not guarantee safe operation when multiple threads use a single session. It is recommended to create as many sessions as the number of threads. However, since creating multiple sessions means that multiple threads are used, the JMS specification has a restriction on more than one session created for a client that operates on Jakarta EE web or EJB container.

The following API is used to create a session from the connection object.

```
public Session createSession(boolean transacted, int acknowledgeMode)
```

```
                              throws JMSException;
```

The JMS specification defines the following four acknowledge modes.

```
Session.AUTO_ACKNOWLEDGE = 1
Session.CLIENT_ACKNOWLEDGE = 2
Session.DUPS_OK_ACKNOWLEDGE = 3
Session.SESSION_TRANSACTED = 0
```

JEUS MQ supports an additional acknowledge mode to maximize the messaging performance.

```
jeus.jms.JeusSession.NONE_ACKNOWLEDGE = -1
```

The JMS specification does not allow a client that operates on Jakarta EE web or EJB container to use local transactions. Thus, such a client cannot create transacted sessions, and CLIENT_ACKNOWLEDGE is disabled for ACKNOWLEDGE mode.

2.3.4. Client Facility Pooling

As mentioned in the previous section, the client facilities including connections, sessions, and message producers are repeatedly created for use. However, these objects exchange messages with the server each time they are created and multiple messages have to be sent and received in order to send a single user message, which can result in performance degradation.

To resolve this issue, JEUS MQ provides client facility pooling. This function can only be used when the message producer is used in a non-transaction environment or an environment where the JEUS MQ failover is not configured. If a message consumer or a transaction is used, the client facilities that are not pooled are removed immediately when they are closed.

This function is optional. To enable the function, configure the following JVM options.

-
  ```
  -Djeus.jms.client.use-pooled-connection-factory=<boolean value>
  ```

  Option to enable Client Facility Pooling function. (Default value: true)

-
  ```
  -Djeus.jms.client.pooled-connection.check-period=<long value>
  ```

  Interval for deleting unused pooled objects. (Unit: milliseconds, default value: 60,000 (1 minute))

-
  ```
  -Djeus.jms.client.pooled-connection.unused-timeout=<long value>
  ```

Option to remove the pooled objects that have been idle longer than the specified time period. (Unit: milliseconds, default value: 120,000 (2 minutes))

2.3.5. NONE_ACKNOWLEDGE Mode

Except for transacted sessions, JMS message acknowledge modes are meaningful only when receiving messages, but the NONE_ACKNOWLEDGE mode also affects the sending of messages.

The acknowledge mode improves performance, but makes reliable messaging difficult. Therefore, when deciding whether to use the NONE_ACKNOWLEDGE mode, consider the message's characteristics, performance and reliability requirements, etc.

When exchanging FileMessages or in transacted sessions, the NONE_ACKNOWLEDGE mode works like the AUTO_ACKNOWLEDGE mode.

Sending Messages

In general, after a client sends a message to the JMS server, the client thread calls the MessageProducer.send() method and suspends its operation until it receives a reply from the server.

In the NONE_ACKNOWLEDGE mode, MessageProducer.send() returns immediately after the client sends a JMS message to the server. This can improve message transmission performance reducing the client waiting time.

The following shows the difference in how a message is sent in general and in the NONE_ACKNOWLEDGE mode. The grey boxes indicate that the messages are logged when they arrive at the server.

In the following cases, a message may be lost.

- When a network failure occurs while a message is being transmitted from the client to the server.
- When an error occurs on the JEUS MQ server before a message arrives at the JEUS MQ server and is added to the destination.

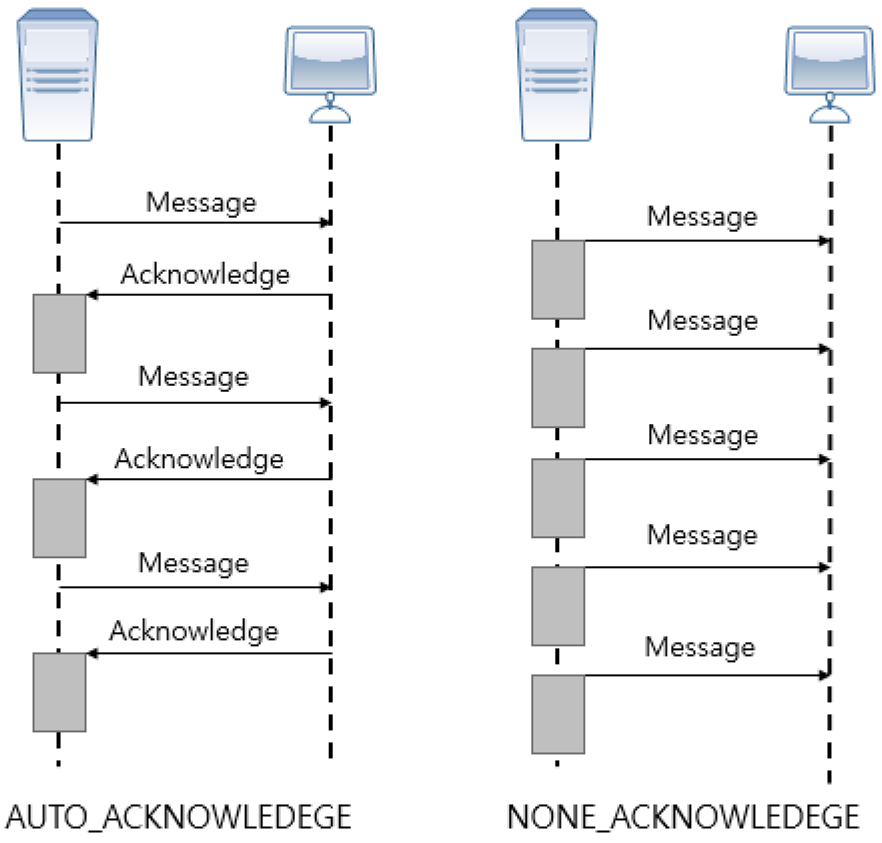
The lost message is not recovered even if the message transmission method is set to DeliveryMode.PERSISTENT.

Receiving Messages

A JMS server does not delete the message information until it receives an acknowledgement from the client that received the message. This can help improve the reliability of messaging, but is not good for performance because each acknowledgment message incurs network overhead.

In the NONE_ACKNOWLEDGE mode, a user can expect faster message transmission speed. This is because the JEUS MQ server sends a JMS message to the client and then immediately deletes the message information from the server without sending an acknowledgement message to the client.

The following figure shows how a message is received in the AUTO_ACKNOWLEDGE and NONE_ACKNOWLEDGE modes. The gray box indicates that a message is deleted from the server.



Receiving Messages in AUTO_ACKNOWLEDGE and NONE_ACKNOWLEDGE modes

In the following cases, a message may be lost.

- When a network failure occurs while a message is being transmitted from the server to the client.
- When an error occurs while the message is being processed by the JEUS MQ client library.
- When an exception occurs in the onMessage() method of the MessageListener object registered by the client.

The lost message cannot be recovered by calling the Session.recover() method.

2.3.6. JMSContext

The JMS specification provides the JMSContext interface that combines connections and sessions, and has roles in both of them. Just like in connections, JMSContext can be created by using the following APIs defined in ConnectionFactory.

```
public Connection createContext()
                        throws JMSException;
public Connection createContext(int sessionMode)
                        throws JMSException;
public Connection createContext(String userName, String password)
                        throws JMSException;
public Connection createConnection(String userName, String password, int sessionMode)
```

```
                        throws JMSException;
```

The ACKNOWLEDGE mode is set just like when creating a session. However, transactions are used according to the mode, not a parameter. In JEUS MQ, JMSContext uses client facility pooling just like connections and sessions. As a result, connections and session pools are shared. Therefore, detailed configuration depends on the settings for connections and session pools.

2.4. Messages

This section discusses the extended features of JMS messaging supported by JEUS MQ.

2.4.1. Message Header Field

JMS defines the following message header fields.

- JMSDestination
- JMSDeliveryMode
- JMSMessageID
- JMSTimestamp
- JMSCorrelationID
- JMSReplyTo
- JMSRedelivered
- JMSType
- JMSExpiration
- JMSPriority

Because JEUS MQ assigns a unique ID to each message, it does not support the following functions:

- MessageProducer.setDisableMessageID(boolean) method disables message ID assignment to each JMS message.
- MessageProducer.setDisableTimestamp(boolean) method disables timestamp assignment to each JMS message.
- Overriding of the JMSDeliveryMode, JMSExpiration and JMSPriority field values that are set in the client.

If a client calls the Message.getJMSMessageID() method after sending a message using the session whose acknowledge mode is set to NONE_ACKNOWLEDGE, the message ID is displayed as NULL. This is because MessageProducer.send() returns before receiving a response from the server. For more information, refer to NONE_ACKNOWLEDGE Mode.

2.4.2. Message Properties

JMS defines the following property names that start with "JMSX?".

- JMSXUserID
- JMSXAppID
- JMSXDeliveryCount
- JMSXGroupID
- JMSXGroupSeq
- JMSXProducerTXID
- JMSXConsumerTXID
- JMSXRcvTimestamp
- JMSXState

The "JMSX?" message properties are not required, except for JMSXDeliveryCount, and they are not supported by JEUS MQ.

The following message properties are supported by JEUS MQ.

- JMS_JEUS_Schedule

  The amount of time that the JEUS MQ server waits before sending a message to the message consumer. The property is identical to Message Delivery Delay defined by JMS. The JEUS MQ server sends the message to the message consumer after the specified amount of time (JMSTimestamp value) has passed since the arrival of the message. The timestamp is a "long" value in milliseconds.

  ```
  Message.setLongProperty("JMS_JEUS_Schedule", <long value>);
  ```

  Message Delivery Delay overrides this property.

- JMS_JEUS_Compaction

  The option to compact the message body. If set to true, the message body is compacted by using the ZLIB library when transmitting the message over the network.

  ```
  Message.setBooleanProperty("JMS_JEUS_Compaction", <boolean value>);
  ```

- JMS_JEUS_RedeliveryLimit

The maximum number of times to re-send a message to a message consumer. When the number of re-send attempts exceeds the specified limit, the message is saved in the dead message destination (Dead Message Destination).

```
Message.setIntProperty("JMS_JEUS_RedeliveryLimit", <integer value>);
```

2.4.3. Message Body

JMS specification defines the following five types of messages, according to the message body type.

- StreamMessage
- MapMessage
- TextMessage
- ObjectMessage
- BytesMessage

Each message type can be created by the session object using the following APIs.

```
public StreamMessage createStreamMessage()
                            throws JMSException;
public MapMessage createMapMessage()
                            throws JMSException;
public TextMessage createTextMessage()
                            throws JMSException;
public TextMessage createTextMessage(String text)
                            throws JMSException;
public ObjectMessage createObjectMessage()
                            throws JMSException;
public ObjectMessage createObjectMessage(Serializable object)
                            throws JMSException;
public BytesMessage createBytesMessage()
                            throws JMSException;
```

2.4.4. FileMessage

JEUS MQ supports the FileMessage type in addition to the basic JMS message types. Since JMS runs based on the message content, the memory has to retain the message content when sending and receiving messages. This may cause memory overflow on the client or server if the message size is too big. To avoid this problem, JEUS MQ supports the FileMessage type that sends message contents in block units.

Creating Messages

Create a FileMessage using the following method defined in the jeus.jms.JeusSession class.

```
public jeus.jms.FileMessage createFileMessage()
                                       throws jakarta.jms.JMSException;
public jeus.jms.FileMessage createFileMessage(java.net.URL url)
                                       throws jakarta.jms.JMSException;
```

The Session, QueueSession, and TopicSession objects created by the JEUS MQ client library can be casted to a jeus.jms.JeusSession, jeus.jms.JeusQueueSession, and jeus.jms.JeusTopicSesison objects respectively.

FileMessage Interface

The following is the definition of the jeus.jms.FileMessage interface.

```
public interface FileMessage extends jakarta.jms.Message {
    public java.net.URL getURL();
    public void setURL(java.net.URL url)
                throws jakarta.jms.MessageNotWriteableException;
    public boolean isURLOnly();
    public void setURLOnly(boolean urlOnly);
}
```

To send a message, the URL of the message file can be set using the setURL() method. The file can also be passed to the JeusSession.createFileMessage() method as a parameter when the message is created. The urlOnly property determines whether to only send the URL of the file on the server to the message consumer. This property determines the return value of getURL() that is called on the FileMessage object.

The following describes getURL() value according to the urlOnly property.

urlOnly	getURL()
true	URL of a file on the JEUS MQ server. This URL is used to receive the file using protocols such as HTTP and FTP.
false	URL of a temporary file on the local server where JEUS MQ client library saves the entire content of the file that it receives. For more information, refer to "Temporary File Path".

When sending a FileMessage, the MessageProducer.send() method always returns a value after all file contents are sent to the server. This is also applicable in the NONE_ACKNOWLEDGE mode.

A file included in a FileMessage is divided into 4 KB blocks. The block size can be modified by specifying the system property like "-Djeus.jms.file.blocksize=<*integer value*>" when executing the JEUS MQ client.

Temporary File Path

If a message consumer receives a file in a FileMessage, the file is saved in the temporary file path. The path is specified according to the following conditions.

- If the application is deployed to JEUS:

  ```
  SERVER_HOME/.workspace/client/
  ```

- If the jeus.jms.client.workdir system property is set:

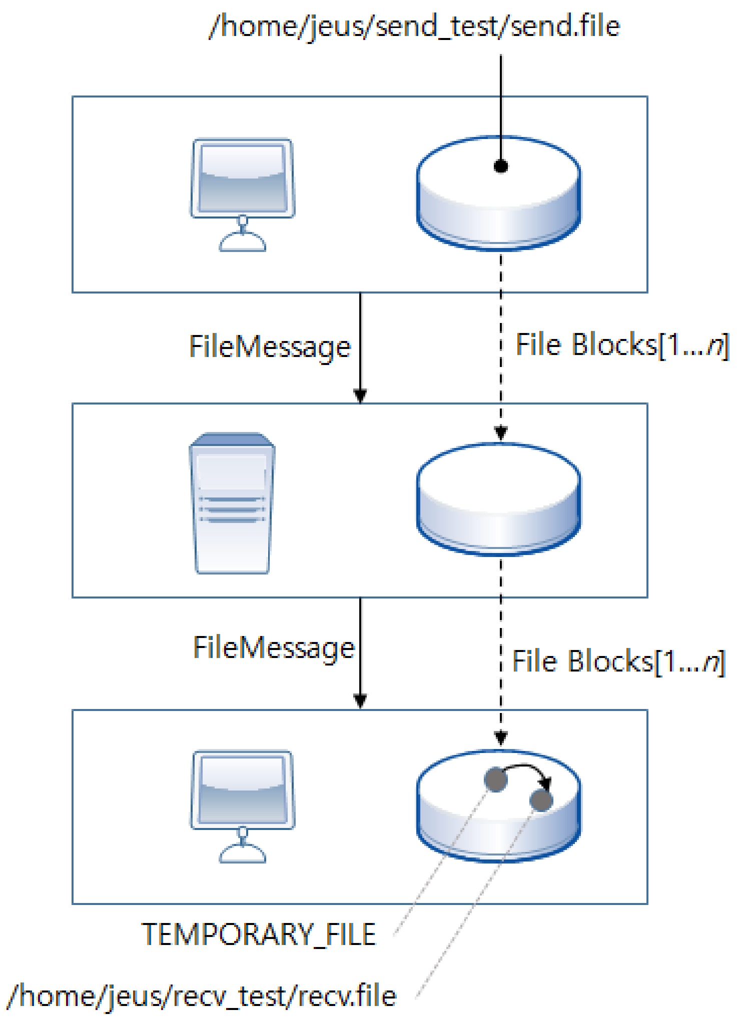
  ```
  Path specified with system property
  ```

- Otherwise:

  ```
  USER_HOME/.jeusmq_client_work/
  ```

FileMessage Transmission Example

This section shows how to use the FileMessage API to send a FileMessage.



FileMessage Transmission Example

The following Java code shows how to send the "/home/jeus/send_test/send.file" file using a FileMessage object.

Sending a FileMessage

```
. . .

jeus.jms.JeusSession session = (jeus.jms.JeusSession)
    connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(destination);

jeus.jms.FileMessage message = session.createFileMessage();
File file = new File("/home/jeus/send_test/send.file");
message.setURL(file.toURI.toURL());

producer.send(message);

. . .
```

The following example shows how to obtain an InputStream from the file's URL and write the file contents in the "/home/jeus/recv_test/recv.file" file.

Receiving a FileMessage

```
. . .
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
MessageConsumer consumer = session.createConsumer(destination);
Message message = consumer.receive();

if (message instanceof jeus.jms.FileMessage) {
    URL url = ((jeus.jms.FileMessage) message).getURL();
    if (url != null) {
       InputStream inputStream = url.openStream();
       BufferedInputStream bufInputStream = new BufferedInputStream(inputStream);

       File outFile = new File("/home/jeus/recv_test/recv.file");
       FileOutputStream fileOutputStream = new FileOutputStream(outFile);
       BufferedOutputStream bufOutputStream = new BufferedOutputStream(fileOutputStream);

       int buf;
       while ((buf = bufInputStream.read()) != -1) {
           bufOutputStream.write(buf);
       }
       bufOutputStream.close();
       bufInputStream.close();
    }
}
. . .
```

2.5. Transactions

This section discusses the features of local transactions in JEUS MQ and distributed(XA) transactions and the scope of the transactions. It also explains what a client developer needs to know to process a transaction.

JMS transaction involves the task of sending and receiving messages inside a session. JMS specification defines the local transaction that starts and ends inside a session. It also defines the distributed transaction that involves the processing of other resources such as one or more JMS sessions, EJBs, and JDBCs.

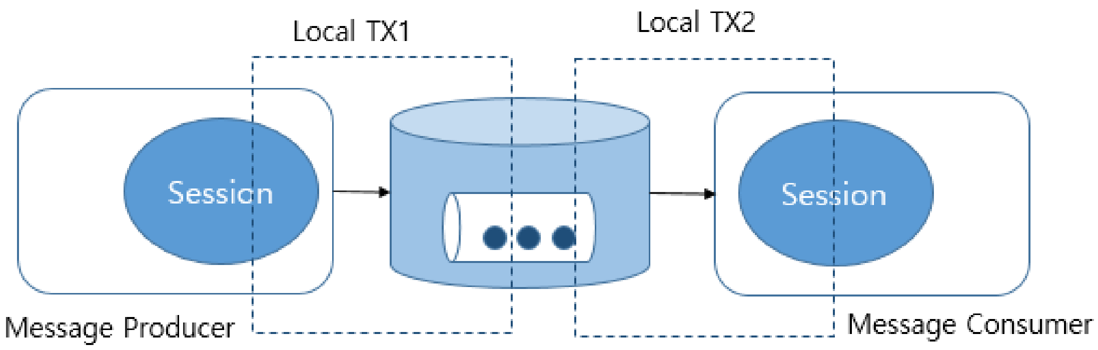
Since a message sent through a session that is involved in a transaction is not seen as having arrived at the server, the message does not get delivered to the message consumer that is created in the same session. They also do not appear in a queue browser that is created in the same session.

2.5.1. Local Transactions

A local transaction is executed by a session that is created by setting the transacted parameter value to true in the Connection.createSession(boolean transacted, int acknowledgeMode) method. It includes all messaging tasks since the last commit or rollback or the tasks that have been executed since the creation of the session. This means that all tasks belong to a particular transaction.

Multiple sessions cannot be processed in a single local transaction, but one or more message producers can be created and sent to multiple destinations. Also, a message can be received from multiple destinations.

The following figure shows the tasks participating in a local transaction and the scope.



JMS Transaction Scope

A local transaction is completed by the session’s commit() or rollback() API. Since a transacted session always participates in a transaction, there is no API that can start a transaction by its name.

Unlike in a distributed transaction, the client can commit or roll back a transaction in a local transaction. Therefore, it is possible to process messages asynchronously in the transaction.

The JMS specification does not allow a client that operates on Jakarta EE web or EJB container to use local transactions. Thus, such a client cannot create transacted sessions.

2.5.2. Distributed Transactions

In the JMS specification, XAResource provided by XASession can be registered to participate in an XA transaction. The actual implementation may vary for each vendor.

The following points must be taken into consideration to use XASession in the JEUS MQ client.

- When invoking Session.getAcknowledgeMode() on an XASession, if the XASession is participating in a global transaction it returns Session.SESSION_TRANSACTED, and otherwise it returns Session.AUTO_ACKNOWLEDGE.
- If Session.commit() or Session.rollback() is called on an XASession participating in a global transaction, the TransactionInProgressException or IllegalStateException is triggered.

Propagation of Distributed Transactions

The JEUS MQ client library registers an XAResource in a global transaction of the thread that uses a JMS API, regardless of when the XASession was created. To propagate a transaction associated with the client thread, a specific API invocation is required. JEUS MQ’s participation in a distributed transaction is made only through a **synchronous API invocation**, which includes the transmission or synchronous reception of messages. The messages asynchronously received through a

MessageListener must not be included in a distributed transaction.

To process an asynchronously received message in a distributed transaction, use message-driven beans (MDB). For more information, refer to "Message Driven Bean (MDB)" in JEUS EJB Guide.

Recovery of Distributed Transactions

The JEUS MQ server stores the ongoing session tasks of a transaction in a storage in order to recover them when the server has to restart due to an unexpected error. The transaction manager can obtain the IDs of ongoing transactions in JEUS MQ through the XAResource obtained from the XASession, and use them to commit or roll back the transactions.

To quickly recover from a failure, it is recommended to use the JEUS MQ failover function. For more information, refer to JEUS MQ Failover.

When restarting, JEUS MQ server automatically rolls back the tasks of transactions that are not in an in-doubt state.

3. JEUS MQ Server Configuration

This chapter describes how to configure the JMS engines and resources that are used to run a JEUS MQ server.

3.1. Overview

A JEUS MQ server functions as a JMS engine in JEUS, and a single JEUS MQ server can run on each JEUS server. For more information about each component of JEUS, refer to "Introduction" in JEUS Server Guide.

3.1.1. Directory Structure

The following figure shows the location of files that are needed to set and manage JEUS and JEUS MQ servers. "JEUS_HOME" is the JEUS installation directory path.

```
{JEUS_HOME}
   |--bin
   |   |--[01]jeusadmin
   |--logs

* Legend
- [01]: binary or executable file
- [X] : XML document
- [J] : JAR file
- [T] : Text file
- [C] : Class file
- [V] : java source file
- [DD] : deployment descriptor
```

jeusadmin

A console tool for managing JEUS Manager. It provides overall management functions for JEUS including JEUS Manager and server control. For more information about how to use the tool, refer to "jeusadmin" in JEUS Reference Guide.

3.2. Configuring JMS Resources

JMS Resource consists of the following two elements.

- **Destination**

  JEUS MQ server destination setting section. Like Queues or Topics, JEUS MQ servers register destinations in JEUS JNDI Service by reading the domain configuration. For more information, refer to Configuring Destinations.

- **Durable Subscription**

  Durable Subscriber setting section for JEUS MQ. For more information, refer to Configuring Durable Subscription.

3.2.1. Configuring Destinations

Destinations such as Queues and Topics also read domain settings and register with the JEUS JNDI service when the JEUS MQ server starts.

Default Settings

The following is an example of setting Destinations in domain.xml. The setting values are configured within the <destination> tag.

domain.xml

```
<domain>
  ...
    <jms-resource>
        <destination>
            <type>queue</type>
            <name>ExamplesQueue</name>
            <export-name>jms/ExamplesQueue</export-name>
        </destination>

        <destination>
            <type>topic</type>
            <name>ExamplesTopic</name>
            <export-name>jms/ExamplesTopic</export-name>
        </destination>
    </jms-resource>
</domain>
```

The following describes each configuration tag.

Tag	Description
<type>	The type of a destination. Can be set to either a queue or a topic.
<export-name>	Destinations can be looked up in JNDI with <export-name> just like Connection Factory. If <export-name> is not specified, the <name> value is used.

Configuring Message Flow Control

If the message listener is registered in a queue or topic, and when messages arrive, the messages will be immediately sent to the clients. If the client cannot process the messages fast enough, the messages are accumulated on the client and an OutOfMemory error occurs. To prevent this error, you need to specify the maximum number of messages that can accumulate on the client after which the server suspends sending messages for a given time.

The following is an example of defining message flow control settings by destination in domain.xml. The settings are configured within the <destination> tag.

domain.xml

```
<domain>
  ...
    <jms-resource>
        <destination>
            <type>queue</type>
            <name>ExamplesQueue</name>
            <export-name>jms/ExamplesQueue</export-name>
            <max-pending-limit>1000</max-pending-limit>
            <resume-dispatch-factor>0.5</resume-dispatch-factor>
        </destination>
    </jms-resource>
</domain>
```

The following describes each configuration tag.

Tag	Description
<max-pending-limit>	The maximum number of messages. (Default: 8192)
<resume-dispatch-factor>	Specifies the proportion of the <max-pending-limit> value. When the JEUS MQ server stops sending messages due to the limitation set in <max-pending-limit>, message transmission will resume once the number of accumulated messages in the client decreases to this proportion. (Default: 0.4) For example, if set as described above, the server will stop sending messages when the number of accumulated messages in the client reaches 1,000, and will resume transmission once the number of messages drops to 500.

3.2.2. Configuring Durable Subscription

Since durable subscribers are typically created by clients, any messages sent to the topic before the durable subscriber creation cannot be delivered to the client. To avoid this issue, JEUS MQ registers a durable subscriber by configuring the durable subscription before the server starts so that it can send a message when a client is connected. Similar to invoking the Session.createDurableSubscriber() method, the client ID, subscription name, and topic must be specified when configuring a durable subscription in the domain.xml file. A message selector can also be configured.

domain.xml

```
<domain>
  ...
    <jms-resource>
        <durable-subscription>
            <client-id>client_id1</client-id>
            <name>subscription1</name>
            <destination-name>ExamplesTopic</destination-name>
        </durable-subscription>
    </jms-resource>
</domain>
```

3.3. Configuring JMS Quotas

3.3.1. Configuring Quotas

If messages fail to be sent to a client and accumulate at the destination, an OutOfMemoryError may occur on the JVM and the server may be terminated. To avoid this problem, JEUS MQ provides quota configuration to apply different memory management for each JMS destination.

JEUS MQ servers hold strong references to message contents when the destination does not use a lot of memory. JEUS MQ servers operate according to the **'Quota'** property settings.

- If the amount of memory used exceeds the **'Quota'** value, messages from the clients trigger a JMSException.
- If the amount of memory used exceeds 75% of the **'Quota'** value, the memory can no longer manage the stored messages.

Even after a message at a destination is deleted from the memory, the message will not be lost. Since the message is saved in the storage, a JEUS MQ server can read and process the message in the storage when necessary. The processing speed is slower when using the storage than memory.

The memory of a JEUS MQ server always keeps the messages that are not saved in the storage. The **'Quota'** property is applicable only when a storage is configured and a message is set to DeliveryMode.PERSISTENT.

After configuring quota, you can configure the destination to use quota functionality. For more information, refer to Configuring Destinations.

3.4. Configuring JMS Engines

You can configure the JMS engine by writing the **domain.xml file**.

3.4.1. Basic Information

JEUS MQ server can be configured within the <jms-engine> tag. The JMS engine provides an environment for using the JMS server on the server. It runs when the server boots, and only one JMS engine is supported on one server.

The JMS engine configuration contains basic options including engine roll, JEUS MQ failover, and quota, as well as advanced options for the JMS engines. To enable recovery from a server or network failure, you can configure the failover settings. For more information about the failover settings, refer to JEUS MQ Failover.

- **Engine Memory Management**

  The quota information for JMS memory management can be configured through **'Max Bytes'** and **'Max Messages'** items on the basic JMS engine configuration. JEUS MQ server manages memory by using the quota settings for the JMS engine, which manages the memory of all destinations within the server, and the quota settings for each destination. For more information about memory management policies, refer to Configuring JMS Quotas.

  The following is an example of configuring the engine memory settings on the server.

  domain.xml

```
<domain>
  ...
    <jms-engine>
        <max-byte>128M</max-byte>
        <max-message>0</max-message>
    </jms-engine>
</domain>
```

  The following describes each configuration tag.

Tag	Description
<max-byte>	Specifies the maximum memory size that can be used by the JMS Engine. It can be set by adding 'K' (KiloBytes), 'M' (MegaBytes), or 'G' (GigaBytes) after the number. (Default: 128MBytes)
<max-message>	Specifies the maximum number of messages that can be used by the JMS Engine. If not set, there is no limit on the number of messages.

- **Advanced Options**

  In the Advanced Options section for the JMS engine, you can configure the thread pool information.

  There are two main types of message handling in the JMS engine. First, a generic message is processed as a single thread by the event manager in order to avoid contention and concurrency. Other special functions, message transfer between cluster, transaction control, and JMS engine control are handled by a thread pool according to each type.

  The following is an example of thread settings on the server.

  domain.xml

```
<domain>
```

```
  ...
    <jms-engine>
        <thread-pool>
             <min>10</min>
             <max>20</max>
             <keep-alive-time>300</keep-alive-time>
        </thread-pool>
    </jms-engine>
</domain>
```

The following describes each configuration tag.

Tag	Description
<min>	Specifies the minimum size of the thread pool.
<max>	Specifies the maximum size of the thread pool.
<keep-alive-time>	Specifies the duration (in seconds) for which unused threads, exceeding the <min> value, can remain idle before being destroyed. (Default: 300)

3.4.2. Configuring Service Channels

A JEUS MQ server communicates with a client through a service channel. Each JEUS MQ server must have one service channel, and it is possible to set a different network such as a service URL for each service channel.

The name of a listener used by the service channel is set with <listener-name> under <service-config>. For detailed information on listener settings, refer to "Listener Configuration" in JEUS Server Guide.

The following sets a service channel on the server.

domain.xml

```
<domain>
  ...
    <jms-engine>
        <service-config>
             <name>default</name>
             <listener-name>jms</listener-name>
             <client-limit>1000</client-limit>
             <client-keepalive-timeout>20</client-keepalive-timeout>
        </service-config>
    </jms-engine>
</domain>
```

The following describes each configuration tag. At least one must be set for the service channel to provide the messaging service.

Tag	Description
<name>	Specifies the name of a service channel. Required to specify channel information to the Connection Factory.
<listener-name>	Specifies a listener for the service channel. Must be selected from the settings that already exist on the server. If not set, the Base-listener is used.
<client-limit>	Specifies the maximum number of clients allowed by the service channel.
<client-keepalive-timeout>	Specifies the time period (in seconds) to wait for reconnection when the client connection is abnormally terminated. When the specified time elapses, all client resources are returned to the server. Since the clientID value is maintained during this period, this setting should only be used when network conditions are poor. If set to a value less than 0, resources are returned immediately without waiting.

3.4.3. Configuring Connection Factories

A connection factory is a JMS management object that contains the connection information needed to access the JMS server and basic information for the client. It is created when the JEUS MQ server boots, and registered in the JEUS JNDI service.

The following is an example of configuring a connection factory on the server.

domain.xml

```
<domain>
  ...
    <jms-engine>
        <connection-factory>
            <type>nonxa</type>
            <name>ConnectionFactory</name>
            <export-name>jms/ConnectionFactory</export-name>
        </connection-factory>
    </jms-engine>
</domain>
```

The following describes each configuration tag.

Tag	Description
<type>	Specifies the type of a connection factory.
<name>	Specifies the name of a connection factory used for management purposes within the JMS system.
<export-name>	Specifies the name with which the connection factory is bound to the Naming Server. If not set, the Name property is used as is.

3.4.4. Configuring Persistence Stores

A persistence store is needed to restore messages, subscriptions, or transactions to the previous state when the server restarts. If a persistence store is not configured, a message that the client sent in the DeliveryMode.PERSISTENT mode cannot be properly delivered if the server fails. JEUS MQ provides two types of persistence stores, journal log and database.

Persistence store can be configured in <persistence-store> under <jms-engine>.

The following is an example of configuring a persistence store on the server.

domain.xml

```
<domain>
  ...
    <jms-engine>
        <persistence-store>
          <journal>
            <base-dir>/home/example/store/jeusmq</base-dir>
            <!--<initial-log-file-count>5</initial-log-file-count>-->
            <!--<max-log-file-count>10</max-log-file-count>-->
            <!--<log-file-size>128M</log-file-size>-->
            <!--<property>-->
              <!--<key>jeus.store.journal.overflow-factor</key>-->
              <!--<value>0.3</value>-->
            <!--</property>-->
          </journal>
          <!--<jdbc>-->
            <!--<data-source>datatsource1</data-source>-->
            <!--<vendor>oracle</vendor>-->
            <!--<destination-table>TEST_DEST</destination-table>-->
            <!--<durable-subscriber-table>TEST_DSUB</durable-subscriber-table>-->
            <!--<message-table>TEST_MESG</message-table>-->
            <!--<subscription-message-table>TEST_SMSG</subscription-message-table>-->
            <!--<transaction-table>TEST_TRAN</transaction-table>-->
          <!--</jdbc>-->
        </persistence-store>
    </jms-engine>
</domain>
```

The following is a description of each child tag for the Store setting of **<persistence-store>**.

- <journal>

  Uses journal log mode.

Tag	Description
<base-dir>	Specifies the name of a directory in which the Store will be created. This directory name must be unique for each Store.
<initial-log-file-count>	Specifies the number of log files that will be initially generated when creating a Journal Store.
<max-log-file-count>	Specifies the maximum number of log files to create.

Tag	Description
<log-file-size>	Specifies the size of a log file. It can be set by appending 'K' (KiloBytes), 'M' (MegaBytes), or 'G' (GigaBytes) after an integer type value or number.

- <jdbc>

  Uses database mode. Currently compatible external databases are 'Oracle Database 9i or later (Enterprise Edition)', 'Tibero 3 SP2 or later', and 'Altibase 4.x (5 or later is not supported)'.

  The **Data Source** property specifies the JNDI name of a data source to be used as a persistence store. For more information about adding data sources to JEUS, refer to "DB Connection Pool and JDBC" in JEUS Server Guide.

  The JNDI configuration is used to change the names of tables in the database. If the aforementioned basic configurations are used, the JEUS MQ server that is in operation can be tested by changing the table names, without having to install additional persistence stores. The following tables in the database are used by default if table names are not configured. <*SERVER-NAME*>_DEST, <*SERVER-NAME*>_DSUB, <*SERVER-NAME*>_MESG, <*SERVER-NAME*>_SMSG, and <*SERVER-NAME*>. For detailed information about each column of the tables created when JDBC is configured, refer to JDBC Persistence Store Columns.

Tag	Description
<data-source>	Set the data source of the database.
<destination-table>	Changes the destination table name. Lowercase is not allowed.

3.4.5. Message Sorting

For more information about message sorting, refer to JEUS MQ Message Sort.

3.5. Management and Monitoring

JEUS MQ servers should be managed with the following functions in order to guarantee seamless services without message loss.

- Manages and monitors the resources on running JEUS MQ servers.

  The following resources are managed and monitored.

  - JMS management objects including connection factories and destinations
  - Messages in destinations and durable subscriptions
  - Connections, sessions, and message producers and consumers
  - Memory space occupied by JEUS MQ servers
  - Persistent storage used by JEUS MQ servers

- Handles failures and recovers JEUS MQ servers from them.

This section introduces how to manage and monitor JEUS MQ servers with jeusadmin, a JEUS console tool.

3.5.1. Server Management

The following describes how to manage servers using jeusadmin.

Using the console tool

You can manage the server resources by executing various commands in the console tool. The console tool is found in the following directory.

```
JEUS_HOME/bin/
```

The following are the jeusadmin commands.

- **Connection Factory**

Command	Description
add-jms-connection-factory	Adds connection factories.
remove-jms-connection-factory	Deletes connection factories.

- **Destination**

Command	Description
add-jms-destination	Adds destinations.
remove-jms-destination	Deletes destinations.

- **Durable Subscriber**

Command	Description
add-jms-durable-subscription	Adds durable subscriptions.
remove-jms-durable-subscription	Deletes durable subscriptions.

For detailed descriptions of these commands, usage, and usage examples, refer to "JMS Engine Commands" in JEUS Reference Guide.

3.5.2. Server Monitoring

This section explains how to monitor servers using jeusadmin.

Using the console tool

You can manage the server resources by executing various commands in the console tool. The console tool is found in the following directory.

```
JEUS_HOME/bin/
```

The following commands are provided for server monitoring.

- **Connection Factory**

Command	Description
list-jms-connection-factories	Checks the connection factory list and displays connection factory information.

- **Destination**

Command	Description
list-jms-destinations	Checks the destination list and displays destination information.
control-jms-destination	Controls the state of a specified destination.

- **Messages**

Command	Description
list-jms-messages	Searches for messages in a specified destination.
view-jms-message	Searches for the detailed information of a specified message.
move-jms-messages	Moves the specified messages to another destination in the cluster or server.
delete-jms-messages	Deletes the specified messages from the destination.
export-jms-messages	Sends the specified messages in the XML format.
import-jms-messages	Retrieves the exported XML messages to the specified destination.

- **Durable Subscription**

Command	Description
list-jms-durable-subscriptions	Searches the list of Durable Subscriptions or the information about a specified Durable Subscription.

- **Clients**

Command	Description
list-jms-clients	Searches the list of clients and displays the client information.
ban-jms-client	Forcibly closes the connection with the client.

- **Transactions**

Command	Description
list-jms-pending-transactions	Displays the list of pending transactions.
commit-jms-pending-transaction	Forcibly commits the specified pending transaction.

For detailed descriptions of these commands, usage, and usage examples, refer to "JMS Engine Commands" in JEUS Reference Guide.

4. JEUS MQ Clustering

This chapter describes the clustering technology that groups multiple servers together to reduce JEUS MQ server loads and provide seamless services.

4.1. Overview

If you are using a single JEUS MQ server, it has to process too many stored messages or client connections, which increases the network load or server memory usage. This may cause performance degradation or server shutdowns. To prevent these problems, you can add and group JEUS MQ servers to work together like a single server. This process is called JEUS MQ clustering. A clustered server can be accessed by a client as it were a stand-alone server.

Furthermore, JEUS MQ Clustering incorporates the failover feature as an integral part. For more information, refer to JEUS MQ Failover.

4.2. Clustering Type

There are two types of clustering, connection factory clustering and destination clustering.

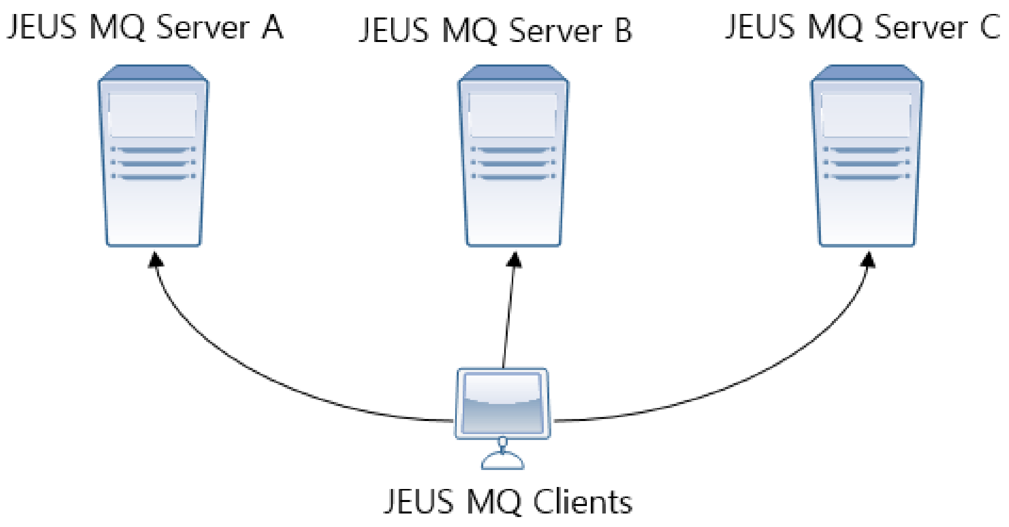
4.2.1. Connection Factory Clustering

When two or more JEUS MQ servers are clustered, client connections to the servers must be distributed. Because connection factories are used to make the connections between servers and clients, this distribution function must be provided by connection factories. This process is called **connection factory clustering**.

When a connection is created from a connection factory, each connection is assigned to a different server, and users can set a policy for selecting a different server each time.

A JEUS MQ client is connected to one server in the cluster, and the client does not need to be concerned about which server it is connected to.

The following figure shows connection factory clustering.

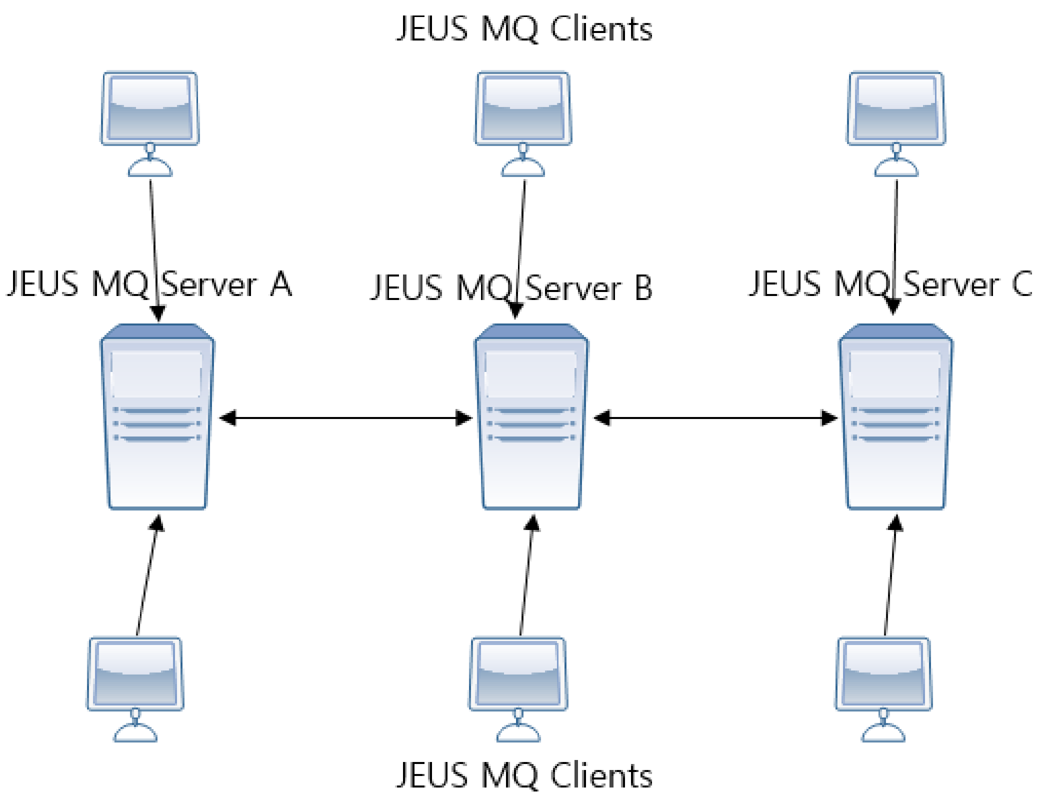


Connection Factory Clustering

4.2.2. Destination Clustering

Even if all client connections are distributed by connection factory clustering, messages that a server receives may exceed its capacity and the server may not be able to receive more messages, depending on the processing speed and network conditions. To prevent this problem, excess messages are moved to the JEUS MQ servers that are holding fewer messages. This process is called **destination clustering**.

The following figure shows destination clustering.



Destination Clustering

4.3. How to Use Clustering

This section describes the configurations required for JEUS MQ clustering.

4.3.1. Server Configuration

JEUS MQ clustering is configured based on JEUS server clustering configurations. For more information about JEUS server clustering, refer to "JEUS Clustering" in JEUS Domain Guide.

JEUS MQ Clustering Configuration Requirements

You can configure Destinations and Durable Subscriptions of JEUS MQ servers that are clustered under the <cluster> setting in domain.xml. The configurations are applied to the servers in the cluster. Since each server has a different address, its connection factory is configured under the <server> tag in domain.xml. Note that all servers must have the same connection factory name in this case as well.

After configuring the <cluster> in domain.xml, you can set a Destination or a Durable Subscription in the cluster under <jms-resource> which is a child setting. For information about configuring individual Destination or Durable Subscription, refer to JMS Resource Configuration.

1. For information on how to add a server to a cluster, refer to "Adding a Server to the Cluster" in JEUS Domain Guide.
2. If a destination is set, the destination or durable subscription settings will be ignored. However, duplicate names must be avoided so that the system can accurately operate.

4.3.2. Client Settings for Clustering

This section describes the client settings required for JEUS MQ clustering. To enable JEUS MQ clustering, the first thing to configure is a connection factory.

There are two types of connection factory clustering in JEUS MQ based on how a connection factory is obtained.

- **Using Clustered Connection Factory**

  The clustered connection factory can be used in the same way as described in Connection Factory, regardless of whether the JEUS MQ server is clustered or not. However, the name of the connection factory must be the same with the name of clustered server as described in Server Configuration section. A client creates only the connection by reusing an already obtained connection factory. The policy for selecting a JEUS MQ server can be set in Connection Factory Configuration, and the current round-robin and random methods are supported.

- **Using JEUS MQ API**

When you are creating a connection factory by using JEUS MQ API, if the JEUS MQ server is clustered, you need to perform an additional task besides those described in Connection Factory.

That is, you need to add information about all the clustered servers by using the JeusConnectionFactoryCreator.addBrokerAddress() method as in the following.

```
jeus.jms.client.util.JeusConnectionFactoryCreator connectionFactoryCreator =
new jeus.jms.client.util.JeusConnectionFactoryCreator();

connectionFactoryCreator.setFactoryName("ConnectionFactory");
connectionFactoryCreator.addServerAddress("192.168.1.2", 9741, <service-name>);
connectionFactoryCreator.addServerAddress("192.168.1.3", 9741, <service-name>);
connectionFactoryCreator.addServerAddress("192.168.1.4", 9741, <service-name>);

ConnectionFactory connectionFactory = connectionFactoryCreator.createConnectionFactory();
```

4.4. Example

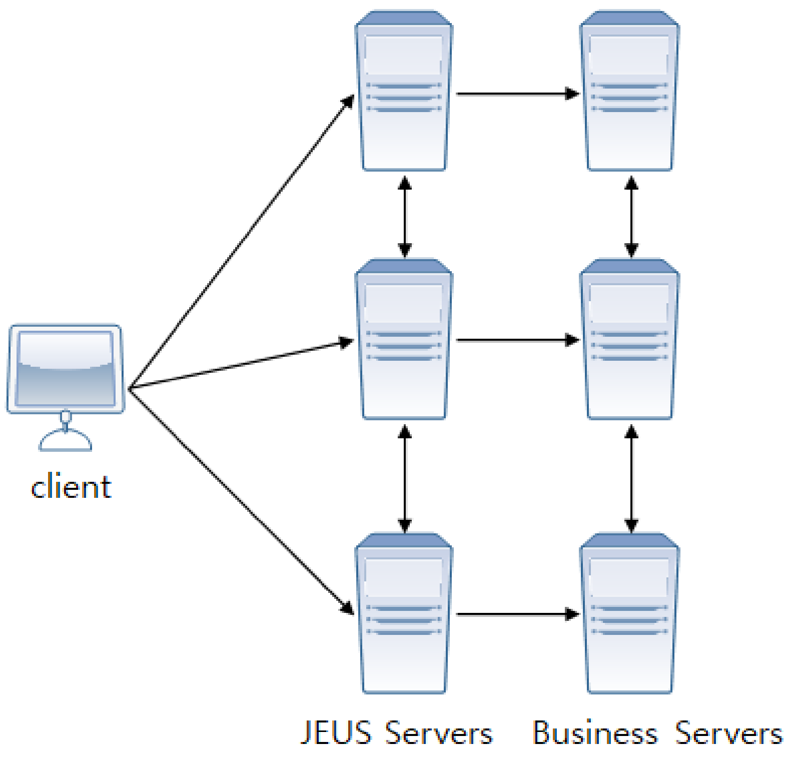
This section describes examples of best and poor JEUS MQ clustering practices.

4.4.1. Example of Best JEUS MQ Clustering Practice

Suppose that there is an online shop that uses the JEUS MQ for processing product orders.

The orders are implemented as messages. The messages are queued in the JEUS MQ and processed on the business servers, and the JEUS MQ servers are clustered.

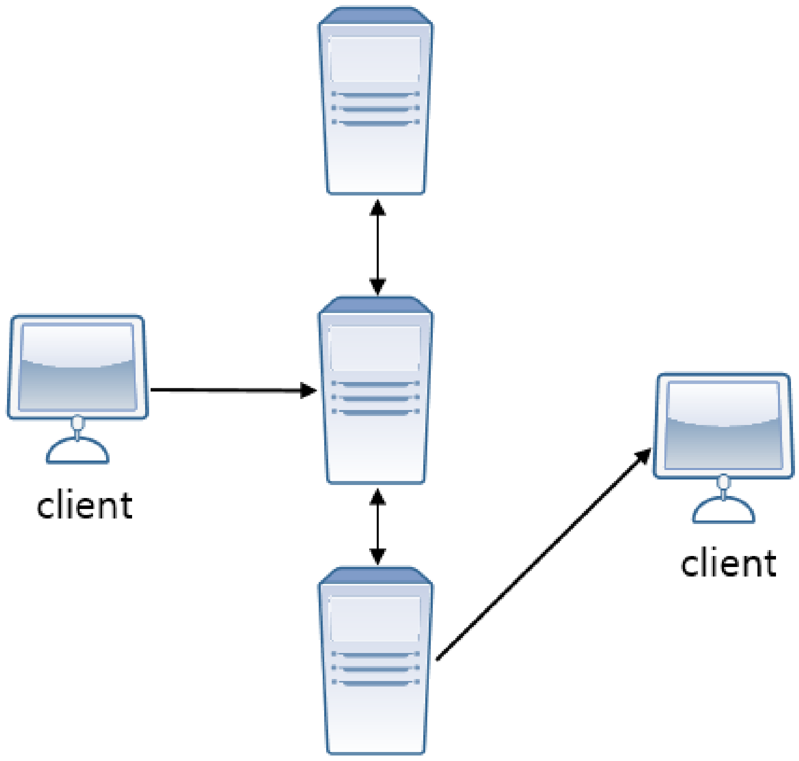
The following is an ideal example of a JEUS MQ server cluster. Each business server receives messages from its corresponding JEUS MQ server, and the request messages are evenly distributed to the JEUS MQ servers. If a particular business server processes its requests fast, its JEUS MQ server can clear the queue faster than others. In this case, the JEUS MQ server can receive messages from other JEUS MQ servers through destination clustering and pass them to its business server for processing.



Example of Best JEUS MQ Clustering Practice

4.4.2. Example of Poor JEUS MQ Clustering Practice

The following JEUS MQ server clustering configuration may reduce processing efficiency.



Example of Poor JEUS MQ Clustering Practice

If a JMS client continually reuses a connection with a particular JEUS MQ server, loads can be unevenly distributed across multiple connections causing unnecessary transmission of messages, which can significantly reduce the processing efficiency.

5. JEUS MQ Failover

This chapter describes how a JMS client can recover from a JEUS MQ server or network failure and re-establish the connection. It also explains the server configuration and server failure recovery that are required for JMS client recovery.

5.1. Overview

In JEUS MQ, when failure occurs a client automatically reconnects to the client application and restores the connection to a point before the failure by using the failover functionality.

Reasons for failure can be classified into the following two categories.

- Network Failure

  If a network failure occurs, a JEUS MQ server can no longer communicate with a client. It maybe that the network is down temporarily or completely unavailable, or the server is down.

  If a network failure occurs, a JEUS MQ client attempts to reconnect to the failed server or to its backup server. If the attempt succeeds, the client state is recovered, and services become available again.

- Server Failure

  Server failure includes all types of failures except network failures. In general, server failures occur due to disk or database failures or a lack of memory. When a server fails, the standby backup server automatically restores data, and continues to provide the service.

To handle such failures, the network between JEUS MQ servers and required JEUS MQ client settings must be configured. Failover properties can be configured for each client by calling the client API provided by JEUS MQ.

5.2. Server Failover

This section describes the network configuration and other required JEUS MQ failover settings.

5.2.1. Network Configuration

To use JEUS MQ failover, one or more active servers must be clustered. A standby server provides a backup support and sufficient capacity during a failure and it is optional. For more information about JEUS clustering configuration, refer to "JEUS Clustering" in JEUS Domain Guide.

- Active Server

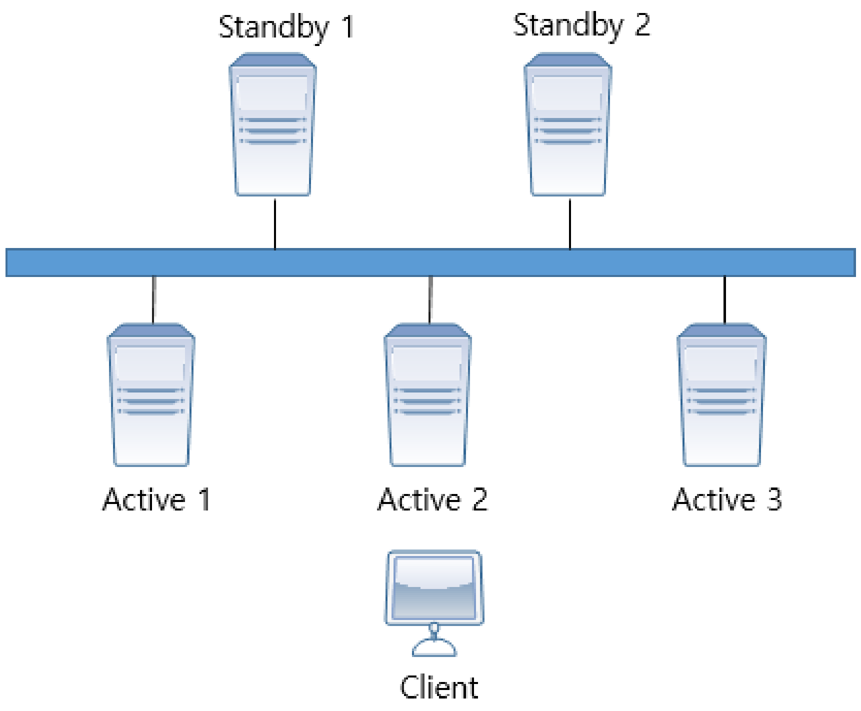
  The main server that processes client requests during normal operation.

- Standby Server

  The backup server that provides the services of the active server when it fails.

JEUS MQ clustering and JEUS MQ failover functions are integrated. Therefore, configuring a JEUS MQ cluster also enables JEUS MQ failover. Unlike the previous versions, active and standby servers are not configured as a pair. This allows servers to be configured more flexibly. A general configuration usually consists of many active servers and some standby servers, or active servers only.

To enable failover, the network between MQ servers is configured as in the following figure.



JEUS MQ Clustering with 3 Active Servers and 2 Standby Servers

When an active server fails, one of the standby servers that are available takes over the operations of the failed active server. If another failure occurs on any of the active or standby servers, another standby server available takes over the failed server's operations. If no standby server is available, one of the active servers that are available takes over the operation, thereby providing services that two servers normally provide. This operation continues until only one server is available. When the last available server fails, the JEUS MQ failover service no longer works.

Active and Standby Server Configuration

The following is an example of setting up failover between the Active server and Standby server.

- **Active Server Configuration**

  Active Server Failover

  domain.xml

```
<domain>
  ...
    <jms-engine>
```

```
        <engine-roll>Active</engine-roll>
        <failover-check-timeout>5</failover-check-timeout>
        <failover-check-count>0</failover-check-count>
    </jms-engine>
</domain>
```

The following describes each configuration tag.

Tag	Description
<engine-roll>	Specifies the role of the JMS Engine. (Default: Active) ◦ Active: Handles service during normal operation. ◦ Standby: Takes over service if the Active engine fails.
<failover-check-timeout>	Specifies the duration (in seconds) to wait before rechecking the availability of the target JMS Engine after a failure is detected, prior to initiating failover. This value represents the time taken for a single attempt. (Default: 5)
<failover-check-count>	Specifies the maximum number of times to recheck the availability of the target JMS Engine after a failure is detected, prior to initiating failover. (Default: 0) If the specified number of attempts to check the engine's availability fails, the engine is considered unavailable, and failover is initiated. If the value is set to 0, failover occurs immediately after a failure is detected.

- **Standby Server Configuration**

  To configure the Standby server, specify 'Standby' within the **<engine-roll>** tag.

5.2.2. Configuring Connection Factories

Failover enables connection factories to redirect connection requests when a JEUS MQ server is unavailable.

The following is a sample connection factory configuration, defined within the <connection-factory> tag in the domain.xml file.

Connection Factory Settings: <domain.xml>

```
<domain>
  ...
    <jms-engine>
        <connection-factory>
            <type>queue</type>
            <name>qcf</name>
            <service>default</service>
            <reconnect-enabled>true</reconnect-enabled>
            <reconnect-period>0</reconnect-period>
            <reconnect-interval>5000</reconnect-interval>
        </connection-factory>
    </jms-engine>
</domain>
```

The following describes each configuration tag.

Tag	Description
<reconnect-enabled>	Specifies whether to reconnect when a failure occurs. (Default: false) To enable client recovery through reconnection in the event of a failure, set the value to true. When enabled, the JEUS MQ client will continuously attempt to reconnect to both the Active and Standby servers.
<reconnect-period>	Specifies the time period for attempting reconnections. If set to the default value, reconnection attempts continue indefinitely. (Default: 0)
<reconnect-interval>	Specifies the wait time between reconnection attempts. (Default: 5)

5.2.3. Configuring Persistence Stores

When the DeliveryMode is set to PERSISTENT, messages are saved in a persistence store.

When a server fails, another active or standby server can retrieve the messages of the failed server from the persistence store to provide seamless services. The persistence store is a key resource of JEUS MQ failover function.

Before configuring a persistence store for JEUS MQ failover, the persistence store must be in a path that can be accessed by the active and standby servers.

- Journal Log Persistence Store

  To use the journal log as the persistence store, the base journal log directory (**Base Dir** of the journal log configuration) has to be under a directory that can be accessed by the active and standby servers. This requires a setup of a disk sharing hardware like SAN and the creation of a journal log base directory.

- JDBC Persistence Store

  To use JDBC as the persistence store, configure a data source under the <jms-engine><persistence-store><jdbc> tags in domain.xml. However, to ensure service continuity in the event of a database failure, you must also set up failover mechanisms using clustering technologies such as Tibero TAC or Oracle RAC.

If a server cannot access the persistence store, failover will be attempted with another server that can access the persistence store.

5.2.4. Automatic Failback

When an active server fails over to another active or standby server, the server administrator must quickly identify possible reasons for failure and restore the failed server.

When the active server restarts, the data is migrated from the backup server to the active server and the connected clients are also reconnected to the restarted server. Such process is called failback. Failback is always performed automatically.

5.3. Client Failover

If a JEUS MQ client is disconnected from the server due to a server or network failure, the client attempts to reconnect to an active server and a standby server, alternating the attempts between the two servers. If successfully reconnected, the client attempts to restore the server to the state where it was before being disconnected. Such client failover process is automatically performed through JEUS MQ configurations without having to change client application source codes.

This section describes the details and restrictions of client failover process and explains how to handle a failure without message loss.

5.3.1. Reconnection

The **"Reconnect Enabled"** option determines whether to try to reconnect if the connection between a client and server is lost. This applies to all connections that are established through the connection factory. For more information, refer to Connection Factory Configuration.

To modify the reconnection configuration of a particular connection, use the "jeus.jms.client.facility.connection.JeusConnection" class, which is the JEUS MQ client API.

```
. . .
import jeus.jms.client.facility.connection.JeusConnection;
. . .
Context ctx = new InitialContext();
ConnectionFactory factory = ctx.lookup("connection-factory");
JeusConnection connection = (JeusConnection)factory.createConnection("jeus", "jeus");
connection.setReconnectEnabled(true);
connection.setReconnectInterval(1000); // 1 second
connection.setReconnectPeriod(3600000); // 1 hour
. . .
```

When **Reconnect Enabled** is set to true, the entire reconnection process is automatically performed on the client application without modifying the client source code.

5.3.2. Reusing the Connection Factory

In JEUS MQ, active and standby servers use the same connection factory. Once a connection factory is obtained through a JNDI lookup, it can be reused without having to look it up again when a server or network failure occurs.

5.3.3. Reusing Destinations

Like connection factories, active and standby servers share the same destination name. Once a destination is obtained through JNDI lookup, it can be reused without having to look it up again when a server or network failure occurs.

When a server fails, all the messages stored at the destination are restored, and the client can continue to process the messages by using the destination.

5.3.4. Request Blocking Time

All requests sent from JEUS MQ clients wait for a response from the server for a specific amount of time. (Default Value: 200000, Unit: ms). This wait time is configured in the **<request-blocking-time>** tag under the **Connection Factory** section in domain.xml.

To configure settings for each connection, you can use the JEUS MQ Client API "jeus.jms.client.facility.connection.JeusConnection" class.

```
. . .
import jeus.jms.client.facility.connection.JeusConnection;
. . .
Context ctx = new InitialContext();
ConnectionFactory factory = ctx.lookup("connection-factory");
JeusConnection connection = (JeusConnection)factory.createConnection("jeus", "jeus");
connection.setRequestBlockingTime(300000); // 5 minutes
. . .
```

RequestBlockingTime is also used as the default transaction timeout value for session or CA transaction.

5.3.5. Connection Recovery

When connection recovery is not configured, JEUS MQ connection share a physical connection (a socket) by default. But if the <reconnect-enabled> element of the connection factory configuration in domain.xml is set to true, each client gets a one-to-one connection with the socket for fail over.

When physical and logical connections establish a one-to-one relationship, a new physical connection has to be created whenever a new connection is created. Since this may result in performance degradation, the client application must be implemented to reuse connections without having to create a new one each time.

On a connection recovery, the connection state is also recovered.

- Start state

  If Connection.start() has been called to receive messages, then it continues to receive messages after the connection recovers.

- Stop state

  If Connection.start() has been called to stop receiving messages, then it does not receive messages after the connection recovers.

Other objects created by using the connection object including sessions and connection consumers are all restored.

- Session Recovery

  Sessions that were created through a connection are restored when the connection recovers, unless Session.close() was called before the failure. For more information, refer to Session Recovery.

- Connection Consumer Recovery

  Connection consumers that were created through a connection are restored when the connection recovers, unless Session.close() was called before the failure. If the connection was in the start state before the connection recovers, then the consumer will start receiving messages again after the recovery. Since the messages that were received before the failure are all returned to the server and retrieved again, the Message.getJMSRedelivered() method call for these messages may return "true".

After recovering from the failure, the methods that create sessions or connection message receivers will re-send its requests and wait for a response. If Connection.close is invoked, recovery is not performed regardless of whether or not a response is issued.

5.3.6. Session Recovery

Sessions are automatically restored during the connection recovery process unless Session.close() is called. In addition, other objects derived from the connection object including MessageConsumers or MessageProducers are all restored.

A session implements methods for creating various objects. The following shows how each method is used after recovery.

- **Message Creation Method**

  Message creation methods are immediately called regardless of the failure.

  ```
  createBytesMessage()
  createMapMessage()
  createMessage()
  ```

```
createObjectMessage()
createObjectMessage(Serializable object)
createStreamMessage()
createTextMessage()
createTextMessage(String text)
```

- **Queue Browser Creation Method**

  Queue browser creation methods complete the request after the recovery. If a failure is not handled during the RequestBlockingTime, a JMSException is raised.

  ```
  createBrowser(Queue queue)
  createBrowser(Queue queue,String messageSelector)
  ```

- **Destination Creation Method**

  Destination creation methods complete the request after the recovery. If a failure is not handled during the RequestBlockingTime, a JMSException is raised.

  ```
  createQueue(String queueName)
  createTopic(String topicName)
  ```

- **Temporary Destination Creation Method**

  Temporary destination creation methods complete the request regardless of failure.

  ```
  createTemporaryQueue()
  createTemporaryTopic()
  ```

- **Message Consumer Creation Method**

  Message consumer creation methods complete a request after the recovery. If a failure is not handled during RequestBlockingTime, a JMSException is raised.

  ```
  createConsumer(Destination destination)
  createConsumer(Destination destination, java.lang.String messageSelector)
  createConsumer(Destination destination, java.lang.String messageSelector,boolean NoLocal)
  ```

- **Durable Message Subscriber Creation Method**

  Durable message subscriber creation methods complete the request after the recovery. If a failure is not handled during RequestBlockingTime, a JMSException is raised.

  ```
  createDurableSubscriber(Topic topic, String name)
  createDurableSubscriber(Topic topic,String name, String messageSelector,boolean noLocal)
  ```

- **Message Producer Creation Method**

  Message producer creation methods complete the request after the recovery. If a failure is not handled during RequestBlockingTime, a JMSException is raised.

  ```
  createProducer(Destination destination)
  ```

If an error occurs in the session, the session transaction is affected in the following cases.

Operation	Description
commit()	If an error occurs while sending and receiving a message through the message producer and consumer that are created by the transaction session, then the "jakarta.jms.TransactionRolledBackException" is generated at the first commit point and the transaction is rolled back. If there are no messages to commit at the commit point, the exception is not thrown. Even after the failure of a commit operation, the subsequent commit operations for the transaction are executed normally. If a failure that occurred during the commit operation is not recovered during the RequestBlockingTime, a JMSException is raised. In this case, the commit operation must be checked by using the administration tool.
rollback()	Rollback() completes a rollback request after the failure is recovered. If a failure that occurred during the rollback operation is not recovered during the RequestBlockingTime, a JMSException is raised. Even after a JMSException, the rollback operation is performed normally.

The Session recover() method completes the recovery request even after the failure has been recovered. When an error occurs after a Session recover() is issued and the failure is not recovered when the RequestBlockingTime expires, a JMSException is raised.

When the acknowledge mode of the session is configured to Session.CLIENT_ACKNOWLEDGE, Message.acknowledge() will be issued for the unacknowledged messages that exist in the session. If an error occurs during the acknowledgement, the ExceptionListener issues a jeus.jms.common.message.MessageAcknowledgeException for each message. The exception notifies that an error has occurred during the message acknowledgement, and the message may be re-delivered.

The MessageID of the failed message can be obtained by calling MessageAcknowledgeException.getErrorCode().

5.3.7. Transmission Error Message Recovery

This section describes how to handle the errors that occur while sending messages through message producers.

The send() method of the message producer is blocked until the message is sent to the server and a response is returned. The following describes possible error scenarios for this process.

- **The send() method is called, but the message has not yet been sent.**

  After recovery, the message is sent to the server and processed successfully. If the failure is not recovered after the RequestBlockingTime expires, a JMSException is raised.

- **The send() method is called, and the message was processed on the server. However, a network error occurs.**

  If the server is reconnected after recovery, a response message is successfully issued. If the failure is not recovered after the RequestBlockingTime expires, a JMSException is raised.

- **The send method is called, and the message was processed on the server. However, a server error occurs, and then the server recovers.**

  Even if the server is reconnected after recovery, it is hard to know whether the message has been successfully transmitted. Thus, a "jeus.jms.common.message.MessageSendException" is issued through the ExceptionListener after the RequestBlockingTime expires.

- **The send method is called, and the message has not yet been processed on the server. However, a network or server error occurs.**

  Even if the server is reconnected after recovery, it is hard to know whether the message has been successfully transmitted. Thus, a "jeus.jms.common.message.MessageSendException" is issued through the ExceptionListener after the RequestBlockingTime expires.

The MessageID of the failed message can be obtained by calling MessageSendException.getErrorCode().

5.3.8. Reception Error Message Recovery

JEUS supports synchronous and asynchronous message reception methods, which perform recovery in different ways. Synchronous message reception methods are described first, followed by asynchronous message reception methods.

Recovery of Synchronously Received Messages

A message consumer can be invoked with three methods for synchronously receiving messages, MessageConsumer.receive(), MessageConsumer.receive(long timeout), and MessageConsumer.receiveNoWait().

The following describes what happens when an error occurs during each method call.

Operation	Description
receive()	This method blocks until a message arrives. But when a failure occurs, it may take a long time for a message to arrive. To avoid an indefinite wait time, change the wait time to the RequestBlockingTime. If the failure is recovered before the wait time expires, send the request message again. Otherwise, a JMSException is raised. If the Session.AUTO_ACKNOWLEDGE option is set, an acknowledgement is sent to the server before the message is passed to the client. If an error occurs, the ExceptionListener issues a jeus.jms.common.message.MessageAcknowledgeException for the messages that have not been acknowledged. The exception indicates that an error has occurred during the message acknowledgement, and the message may be redelivered.
receive(long timeout)	This method blocks until a message arrives. But when a failure occurs, it may take a long time for a message to arrive. To avoid an indefinite time out, change the timeout value that is greater than the RequestBlockingTime to the RequestBlockingTime value. If the failure is recovered before the timeout expires, send the request message again. Otherwise, a JMSException is raised. If the Session.AUTO_ACKNOWLEDGE option is set, an acknowledgement is sent to the server before the message is passed to the client. If an error occurs, the ExceptionListener issues a jeus.jms.common.message.MessageAcknowledgeException for the messages that have not been acknowledged. The exception indicates that an error has occurred during the message acknowledgement, and the message may be redelivered.
receiveNoWait()	This method does not block even if a message does not arrive. It immediately receives the next message that has arrived.

Recovery of Asynchronously Received Messages

Asynchronously received messages are categorized into those being processed by MessageListener.onMessage, those being acknowledged after being processed by MessageListener.onMessage, or those prefetched and waiting in the client queue.

Each category of messages goes through a different fail over process.

- If a failure occurs while the on message method is processed, the failure is recovered first and then an acknowledgement is sent. After this, the message is normally processed.
- If a failure occurs while an acknowledgement is being delivered, the ExceptionListener will issue a jeus.jms.common.message.MessageAcknowledgeException for the message that has not been acknowledged. The exception indicates that a failure occurred during message acknowledgement, and the message may be redelivered.
- If a failure occurs while prefetched messages are waiting in the client queue, the failure is

recovered and then the messages are sent to the server and then later to the client. The Message.getJMSRedelivered() method call for these messages may return "true".

The MessageID of the failed message can be obtained by calling MessageAcknowledgeException.getErrorCode().

5.3.9. Message Loss Prevention and Transactions

A JEUS MW fail over is automatically and transparently processed in a client application. But when messages become lost during the message transmission process, they must be processed separately by the ExceptionListener.

Message loss in an enterprise messaging application can be critical. The only way to perfectly recover from a failure while preventing message loss is to use transactions.

It is strongly recommended to use the following method to create an application.

- In the Jakarta EE environment, messages have to be sent and received within a transaction.
- For servlet, the UserTransaction must be looked up in the JNDI object, and messages must be sent and received within the UserTransaction.
- For EJBs, the TransactionAttribute of the EJB method must be set to "Required" or "RequireNew" so that the messages can be sent and received within a transaction.

General Java clients call the Connection.createSession(true, Session.SESSION_TRANSACTED) method to create a session. Such sessions can send and receive messages within a transaction by calling commit() or rollback().

6. JEUS MQ Special Functions

This chapter describes the special functions of JEUS MQ, including message bridges, message sort, Global Order, message groups, message management, and topic multicast.

6.1. JEUS MQ Message Bridge

Message bridge is a function that connects two different MQs. The two different MQs can be classified into the following cases.

- Different versions of the same MQ service (two different versions of JEUS MQs that are not compatible with each other)
- Different MQ services (JEUS MQ and another vendor's MQ such as WebLogic)

It is not recommended to configure a message bridge between the same versions of JEUS MQ services.

If you are using the same versions, it is safer and more efficient to connect a client to a remote destination. If a bridge between the same JEUS MQ service versions must be configured, to distinguish the bridge from general message services between the MQs, add the following option to the server's execution script:

```
-Djeus.jms.client.use-single-server-entry=false
```

6.1.1. Server Configuration

The following two configurations are required to use a message bridge.

- Configure two bridge connections for each end of a message bridge.
- Configure the actual bridge entry that connects the two bridge connections.

Since Message Bridge operates separately from JEUS MQ, it is set under <resources> in domain.xml. The following is an example of setting up a bridge from JEUS 6's MQ to Weblogic 10.3's MQ.

domain.xml

```
<domain>
  ...
    <resource>
        <message-bridge>
            <bridge-connections>
                <connection>
                    <name>jeus6</name>
                    <classpath>file:///home/seunghoon/workspace/jeus6/lib/client/
    clientcontainer.jar</classpath>
                    <jndi-provider-url>127.0.0.1:19736</jndi-provider-url>
                    <jndi-initial-context-factory>jeus.jndi.JNSContextFactory
                    </jndi-initial-context-factory>
                    <connection-factory>XAQueueConnectionFactory</connection-factory>
                    <xa-support>false</xa-support>
                </connection>
                <connection>
                    <name>weblogic103</name>
                    <classpath>
                   file:///home/seunghoon/bea/wlserver_10.3/server/lib/wljmsclient.jar
                    </classpath>
                    <jndi-provider-url>t3://localhost:7001 </jndi-provider-url>
                    <jndi-initial-context-factory>
                   weblogic.jndi.WLInitialContextFactory</jndi-initial-context-factory>
                    <connection-factory>QueueConnectionFactory</connection-factory>
                    <xa-support>false</xa-support>
                </connection>
            </bridge-connections>
            <bridges>
                <bridge>
                    <name>bridge1</name>
                    <source>
                        <connection-name>jeus6</connection-name>
                        <destination>ExamplesQueue</destination>
                        <type>queue</type>
                    </source>
                    <target>
                        <connection-name>weblogic103</connection-name>
                        <destination>ExamplesQueue</destination>
                        <type>queue</type>
                    </target>
                </bridge>
            </bridges>
        </message-bridge>
    </resource>
</domain>
```

The following describes each configuration tag. All settings can be made under <message-bridge>.

- <bridge-connection>

  Specifies a specific MQ to connect to.

Tag	Description
<name>	Specifies the name to represent the MQ.
<classpath>	Specifies the path to the JMS client library provided by the MQ.
<jndi-provider-url>	Specifies the Provider URL required when accessing the Naming Service where the Destination or ConnectionFactory of the corresponding MQ is registered.

Tag	Description
<jndi-initial-context-factory>	Specifies the fully qualified name of the Initial Context Factory required when accessing the Naming Service where the Destination or Connection Factory of the corresponding MQ is registered.
<connection-factory>	Specifies the name of the Connection Factory to use when accessing the MQ.
<xa-support>	Specifies whether the MQ supports XA. Even if set to true, XA cannot be used if the Connection Factory itself does not support it.

- <bridges><bridge>

  Specifies the connection (Bridge) between MQs set in <bridge-connection>. The child <bridge> tag indicates an individual bridge.

Tag	Description
<name>	Specifies the name of the bridge.
<source>	Indicates the MQ from which the Bridge will receive messages and the specific Destination of that MQ. ◦ <connection-name>: Choose one of the names specified in <bridge-connection>. ◦ <destination>: Set a specific destination for the MQ. ◦ <type>: Set the type of the destination. Choose either queue or topic.
<target>	Specifies the MQ to which the Bridge will forward the received message and the specific Destination of that MQ. It is configured in the same way as the <source> setting.

6.2. JEUS MQ Message Sort

If there is a message producer but no consumer in a queue, or if there is a message consumer but the message reception is relatively slow, messages accumulate in the queue. When this happens, you can use a message sort function sort the queued messages according to key values that you define.

Messages can also be accumulated in a durable subscriber that is similar to a queue, and when this happens, you can use the message sort function. The key values include the basic JMS property values or property values that you define (User Property).

Message sorting is configured differently for servers and clients. To enable message sorting, servers are configured with domain.xml, while clients are configured with the message properties.

6.2.1. Server Configuration

To enable message sorting, a server has to be configured first. This message sorting configuration can be applied for a destination or durable subscription. For more information, refer to Destination Configuration and Durable Subscription Configuration.

The following is an example of setting the message sorting feature on the server.

domain.xml

```
<domain>
  ...
    <jms-engine>
        <message-sort>
            <name>example</name>
            <key>TEST</key>
            <type>Integer</type>
            <direction>ascending</direction>
        </message-sort>
    </jms-engine>

    <jms-resource>
        <destination>
            <type>queue</type>
            <name>ExamplesQueue</name>
            <message-sort>example</message-sort>
        </destination>
    </jms-resource>
</domain>
```

The **<message-sort>** tag specifies the property that will be the key value for sorting, the type of the property, and the sorting direction. And if you add the name of this <message-sort> to the destination or durable subscriber, the settings will be applied.

The following describes each child tag of <message-sort>.

Tag	Description
<name>	Defines a name to represent the <message-sort> setting. Add this name to the <message-sort> setting of a queue or durable subscriber will to apply the setting.
<key>	Defines the name of the key that will be the basis for sorting.
<type>	Specifies the type of the key value. Choose one of Boolean, Byte, Float, Integer, Double, or String. (Default: String)
<direction>	Specifies the direction of sorting. Choose either ascending or descending.

The messages that do not correspond to the configured key value are not sorted, and the original order is maintained. This means that if there is a mixture of messages that correspond to and do not correspond to a key value in the queue, the former will be sorted in the configured order, and the latter will be kept in their original order.

6.2.2. Client Configuration

You can configure a key value that defines the Message Sorting property.

In the following example, the message sort key property is set to "TEST_KEY" that is used to sort messages.

```
Message msg = session.createTextMessage("Test");
msg.setIntProperty("TEST_KEY", 1);
```

6.3. JEUS MQ Global Order

Global Order ensures that the messages queued at the destination are delivered to a client exactly one at a time. In general, messages are processed simultaneously by multiple clients and there is no way to control the processing order of the messages. If multiple messages are sent in parallel, they are processed regardless of the order in which they are sent. Although this does not impact system performance, the global order function can be used to ensure the message processing order.

6.3.1. Client Settings

The client can use the Global Order API without additional configurations.

For a single message consumer, Global Order does not change the existing order. Hence, to use Global Order, multiple message consumers have to be configured for the destination.

In the following example, a message sender creates a Producer and casts the message to JeusMessageProducer, and then calls the Global Order API.

```
JeusMessageProducer producer = (JeusMessageProducer) session.createProducer(queue);
// Set global order and name
producer.startGlobalOrder("GLOBAL-ORDER-NAME");
// If name is not set
//producer.startGlobalOrder();
```

The GLOBAL-ORDER-NAME identifies the Global Order. If it is not configured, a random name is assigned. The Global Order name can be shared by multiple clients. **If this function is used with clustering, the processing order is applied to the entire cluster.**

6.4. JEUS MQ Message Group

Message group is a function that sends a group of messages that have the same purpose and are all queued at a destination to a single message consumer. For example, if 10 messages are set in a group, the messages are not delivered to the consumer until all 10 messages are in the queue.

This is similar to the transaction mechanism and can be used for similar purposes, and it can also be used with clustering.

6.4.1. Server Configuration

The following is an example of setting the message group feature on the server.

domain.xml

```
<domain>
  ...
    <jms-resource>
        <destination>
            <type>queue</type>
            <name>ExamplesQueue</name>
            <message-group>
                <message-handling>Pass</message-handling>
                <expiration-time>-1</expiration-time>
            </message-group>
        </destination>
    </jms-resource>
</domain>
```

The **<message-group>** tag defines the message group settings for a destination.

The following describes each child tag of <message-group>.

Tag	Description
<message-handling>	Defines how the destination processes message groups. Choose either 'Pass' or 'Gather'. 'Pass' processes message groups the same way as individual messages. 'Gather' combines multiple messages in a group and delivers them as a single message.
<expiration-time>	Specifies the duration (in seconds) for which an incomplete message group can exist in the destination. The default value is -1, meaning it will not be destroyed until it is completed.

Make sure that you specify a value for **'Expiration Time'**. Otherwise, an incomplete message group is created that is kept permanently in the server. This can cause unnecessary memory usage.

6.4.2. Client Configuration

In the client configuration, you must configure the message producer and message consumer settings.

- Configuring the Message Producer

  To configure the message producer, use the message's User Property.

```
Message msg = session.createMessage();
// Message group name that represents the group.
msg.setStringProperty("JMS_JEUS_MSG_GROUP_NAME", "MESSAGE-GROUP-NAME");

// The order within the group, as well as the the priority number that determines the sequence
for delivery to the client.
msg.setIntProperty("JMS_JEUS_MSG_GROUP_NUMBERING", 1);
producer.send(msg);

msg = session.createMessage();
msg.setStringProperty("JMS_JEUS_MSG_GROUP_NAME", "MESSAGE-GROUP-NAME");
msg.setIntProperty("JMS_JEUS_MSG_GROUP_NUMBERING", 2);
producer.send(msg);
. . .
msg = session.createMessage();
msg.setStringProperty("JMS_JEUS_MSG_GROUP_NAME", "MESSAGE-GROUP-NAME");
msg.setIntProperty("JMS_JEUS_MSG_GROUP_NUMBERING", 10);

// Last message of the group.
msg.setBooleanProperty("JMS_JEUS_MSG_GROUP_END", true);
producer.send(msg);
```

- Configuring the Message Consumer

  The message consumer receives the message group as an ObjectMessage that contains the messages.

```
// Messages within a message group is a single ObjectMessage, which is a list of the messages.
ObjectMessage result = (ObjectMessage) receiver.receive(TIME_OUT);
List list = (List) result.getObject();

int cnt = 1;
// Get one by one from the list as follows.
for(Object obj : list) {
    // Process messages by the order set in the message transmitter.
    TextMessage msg = (TextMessage) obj;
    . . .
}
```

6.5. JEUS MQ Message Management Functions

Message management functions are used to check, move, and delete messages that arrive at a destination while JEUS MQ messaging services are enabled.

The message management functions can be grouped into the following three categories.

- Message Monitoring Function

  Monitors messages in a destination.

- Message Control Function

  Controls messages by moving, deleting, exporting, and importing them.

- Destination Monitoring

  Monitors the detailed information of a destination.

- Destination Control Functions

  Controls destinations for flexible message monitoring and control.

Message monitoring and control are supported for the messages in a queue or durable subscription. Destination control is provided for the messages in the queues and topics.

6.5.1. Message Monitoring

With message monitoring functions, you can monitor messages in a queue or durable subscription and retrieve detailed message information.

Searching the Message List

You can monitor messages in a queue or durable subscription by selecting the ID, type, creation time, and other information. To search, delete, move, export, and perform other tasks, select a message from the list.

JMS messages are created and consumed quickly. If a message is searched for without enabling the Consumption Suspended setting in the Destination page, the search result may contain a message that has already been consumed.

Following are the steps for searching the message list.

1. Different jeusadmin commands can be used depending on the location of the message to be searched.
   - Messages in a Queue

     For information about commands for querying the destination list, refer to "list-jms-destinations" in JEUS Reference Guide.

   - Messages in a Durable Subscription

     For information about commands for querying the durable subscription list, refer to "list-jms-durable-subscriptions" in JEUS Reference Guide.

2. The message list in a Queue or Durable Subscription can be queried as well.

   You can look for a specific message in the message list by specifying the message ID, type, or creation time, or by using the message selector. To retrieve all messages in a Queue or Durable Subscription, enter the command with no parameter.

3. From the list of messages, you can delete, move, or export a message. You can also see detailed information about the message.

Retrieving Detailed Message Information

The detailed information about messages in the Queues or Durable Subscriptions can be retrieved. It is possible to check the header information defined in JMS specification, message information, and configured properties.

JMS messages are created and consumed quickly. If a message is queried without enabling the Consumption Suspended setting for the Destination, the search result may contain a message that has already been consumed and thus no longer exists.

For information about commands for viewing message details, refer to "view-jms-message" in JEUS Reference Guide.

6.5.2. Message Control

jeusadmin provides control functions for managing incoming messages on the server, including moving, deleting, exporting, and importing messages.

Moving Messages

You can move messages to another destination on a server or cluster. This function is useful for handling errors such as sending messages to a wrong destination.

1. JMS messages are created and consumed quickly. If the messages are moved without enabling the Consumption Suspended setting in the Destination page, the search result may contain a message that has already been consumed.
2. You can move messages to another destination. If a message at the target destination has the same ID as the one you moved, the new message overwrites the existing one. But if the existing message is already sent to a client while being overwritten, the message may not be synchronized between the client and the server. To avoid this problem, enable the Consumption Suspended setting before moving a message.

For information about commands for moving messages, refer to "move-jms-messages" in JEUS Reference Guide.

'All' option moves all messages in the current Queue or Durable subscriber to the specified Destination.

Deleting Messages

You can delete messages that are no longer needed.

JMS messages are created and consumed quickly. If the messages are deleted without enabling the Consumption Suspended setting in the Destination page, the search result may contain a message that has already been consumed. It is recommended to enable the Consumption Suspended setting before deleting a message.

For information about commands for deleting messages, refer to "delete-jms-messages" in JEUS Reference Guide.

To delete all messages in the current destination, use the **All** option.

Exporting Messages

The messages in a Queue or Durable Subscription can be exported to another server or cluster.

JMS messages are created and consumed quickly. If the messages are exported without enabling the Consumption Suspended setting in the Destination page, the search result may contain a message that has already been consumed. It is recommended to enable the Consumption Suspended setting before exporting a message.

For information about commands for exporting messages, refer to "export-jms-messages" in JEUS Reference Guide.

To export all messages in a destination or a durable subscription, use the **'All'** option. As the result, messages are exported and then an xml file named 'messages.xml' will be downloaded through the browser.

Importing Messages

Messages can be imported to a destination as XML formatted data. An imported message is treated as a new message. Therefore, a new message ID is assigned to the message when it is imported to a destination.

If the 'Overwrite' checkbox is checked when importing messages, some messages that are overwritten may already have been delivered to the clients. In this case, messages are imported successfully but incorrectly overwritten. Hence, it is recommended to enable the Consumption Suspended setting before importing a message.

For information about commands for importing messages, refer to "import-jms-messages" in JEUS Reference Guide.

To overwrite an existing message at the destination that has the same message ID as the imported one, use the **'Overwrite'** option. The message with the same message ID as the imported message will be overwritten.

6.5.3. Destination Monitoring

Using jeusadmin, you can monitor detailed information and statistics by destination as follows:

Item	Description
Processed Messages	Messages processed by the destination.
Remaining Messages (current)	Messages currently remaining in the destination.
Remaining Messages (high mark)	The maximum number of messages remaining in the destination.
Pending Messages	Messages that have arrived at the destination successfully but have not yet been delivered to the receiver. You can search only when the Destination is a Queue.
Dispatched Messages	Messages that have been delivered from the destination to the receiver but have not received a response yet. You can search only when the Destination is a Queue.
Delivered Messages	Messages that have entered the destination and have been successfully delivered to the receiver.
Expired Messages	Messages that have entered the destination and have expired.
Moved Messages	Messages that have entered the destination and have been transferred to another destination. You can search only when the Destination is a Queue.
Removed Messages	Messages that have entered the destination and have been removed. You can search only when the Destination is a Queue.
Poisoned Messages	Messages that have entered the destination but were processed incorrectly.
Memory Usage (current)	The amount of memory currently used by the destination.

Item	Description
Memory Usage (high mark)	The maximum amount of memory that has been used by the destination.

6.5.4. Destination Control

Using jeusadmin, you can suspend or resume production and consumption services by destination.

Suspending or Resuming Message Production at a Destination

You can suspend or resume message production and consumption at a destination. When message production is suspended at a destination, you cannot create a message.

For information about commands for suspending and resuming production of the destination, refer to "Destination Status Control" in JEUS Reference Guide.

When a message is sent to a destination where message production has been suspended, the server waits for a specified time and throw a jeus.jms.common.destination.InvalidDestinationStateException through the ExceptionListener.

Suspending or Resuming Message Consumption at a Destination

You can suspend or resume message consumption at a destination. If you suspend message consumption is suspended at a destination, messages cannot be consumed at the destination.

For information about commands for suspending and resuming consumption of destinations, refer to "Destination Status Control" in JEUS Reference Guide.

When an attempt is made to consume a suspended message, the server operates as if there were no messages at the destination.

6.6. JEUS MQ Topic Multicast

Topic Multicast is a feature of transmitting messages from a topic to subscriber based on an IP multicast using User Datagram Protocol (UDP).

This provides significant performance advantages when using a pub/sub topic, but there may be a decrease in the reliability due to UDP for transmission.

This feature is available only in environments where IP multicast can be used. For testing whether the environment supports IP Multicast, refer to "Creating a Domain that Uses Virtual Multicast" in

JEUS Domain Guide.

6.6.1. Server Configuration

Exchanging topic messages via IP Multicast requires two configurations. Firstly, the server and client must be configured to connect to the same multicase address. Secondly, the clients connected to the server must be configured to receive messages through Topic Multicast.

The multicast address is configured in the destination settings. By including the multicast address in the destination settings for topics, the client using that destination can receive messages through the same multicast address as the server without separate client configuration.

You can specify the way clients receive messages as Topic Multicast in the ConnectionFactory. To use the Topic Multicast feature, use the ConnectionFactory in which the Topic Multicast has been configured. If the Topic Multicast is not configured in the destination, messages can be received through the Transmission Control Protocol (TCP), which is the traditional method. Even if the Topic Multicast is set in a destination, messages are received through the TCP method if the Topic Multicast is not configured in the ConnectionFactory. When using durable subscriptions, a higher level of reliability is required compared to regular topic messages. Since one subscription delivers messages to only one client at a time, it does not support the Topic Multicast.

6.7. Reliable Message Transmission

In JEUS MQ, reliable message transmission is ensured by using a persistent store even in the case of a failure. A persistent store guarantees reliable message delivery between a JEUS MQ server and the message recipient. But it does not guarantee reliable message delivery when a message is sent to the server and when the server or the client fails.

To compensate for this limit, JEUS MQ provides a function to increase the reliability between the sender and the JEUS MQ server using a local persistent queue (hereafter LPQ) that is similar to the persistence store. LPQ saves data in the local storage and ensures that the stored data is processed normally. JEUS MQ saves messages using LPQ before they are sent and makes continuous attempts to send them until they are successfully sent to the server. Thus, LPQ increases reliability of the message transmission between the sender and the server.

Reliable message transmission using LPQ has the following characteristics.

- Retransmitting messages until it succeeds regardless of the server status
- Recovery of messages that failed to be sent due to abnormal client status
- Asynchronous messaging

When the JMS client is sending a message, if the client is a Jakarta EE application that is deployed on the JMS server, which is the message recipient, the message is not sent over the network to improve performance. In this case, the LPQ configuration is ignored.

6.7.1. LPQ Activation

To use reliable messaging through LPQ, you must activate LPQ first. You need to create a storage to store messages, a queue for processing, and an object for managing the messages.

To execute a standalone client that uses LPQ, LPQ and storage libraries are additionally needed. The libraries are jeus-lpq-spi.jar, jeus-lpq.jar, jms-extentsion.jar, and jeus-store.jar, which are in the JEUS_HOME/lib/system directory.

To activate LPQ, choose one of the following ways:

- Using the JVM option

  When a JEUS MQ client is executed, the JVM option is applied to activate LPQ as in the following. This option activates LPQ when JMS creates a connection in the client.

  ```
  -Djeus.jms.client.send-by-lpq-only=true
  ```

- Using the LPQ configuration file

  When a JEUS MQ client is executed, LPQ is activated as it is configured by reading the file if the LPQ configuration file can be found in the designated path. For detailed information about the LPQ configuration file path and how to configure it, refer to LPQ Configuration.

- Using the API for JEUS

  LPQ can be activated in the client source code by using the following API of JeusSession, a JEUS session object.

  ```
  public void startLPQ();
  ```

1. The LPQ operation can be configured through the configuration file or system property when the JVM option or JEUS API is used. The default value is applied for unconfigured items. For detailed information about LPQ configuration, refer to LPQ Configuration.
2. If JEUS API is not used, when LPQ will be activated is not specified. In this case, LPQ is activated when the JVM creates the first connection. LPQ terminates when the last connection is closed. However, if LPQ still has messages, it does not terminate until they have been processed.

6.7.2. Enabling LPQ

Once LPQ is activated, a message can be sent more reliably by using LPQ. There are four configuration units for sending messages through LPQ.

- **JVM Unit Configuration**

  Used when LPQ is activated by using the JVM option. All messages from the JVM are sent through LPQ.

- **ConnectionFactory Unit Configuration**

  When the Connection Factory names to use LPQ are added to the JVM option, all the messages that are sent from the connections created by the Connection Factory are sent through LPQ. Connection factory names are separately by a comma (,).

```
jeus.jms.client.connection-factory-for-lpq=<ConnectionFactoryName1,ConnectionFactoryName2,..>
```

JMS specifications do not have naming limits for connection factories. However, if a commas is used, the above option cannot be normally applied and the LPQ function cannot be used.

- **Session Unit Configuration**

  Used to specify whether to send messages to each session by using LPQ. The following API of the JEUS message sender object can be used to send all messages that will be sent to the session by using LPQ.

```
public void setLPQOnly(boolean lpqOnly);
```

- **Message Unit Configuration**

  Used to specify whether to send each message through LPQ. Either use the following APIs of the JEUS message sender object, or configure the User Property of the message.

```
// JEUS-dedicated API
public void sendWithLPQ(Message message);
public void sendWithLPQ(Message message, int deliveryMode, int priority, timeToLive);
public void sendWithLPQ(Destination destination, Message message);
public void sendWithLPQ(Destination destination, Message message, int deliveryMode, int priority,
timeToLive);

// JMS message user property
Message msg = session.createMessage();
msg.setBooleanProperty("JMS_JEUS_USE_LPQ", true);
```

LPQ activation by using JEUS API only enables LPQ to be used in the corresponding session. LPQ configuration for messages that are sent using other sessions is ignored.

6.7.3. LPQ Listener Configuration

When LPQ is used to send messages, the messages are processed asynchronously. Therefore, the client does not know whether a message has been sent or not. LPQ provides a listener to be able to get the message transmission time and result.

The following jeus.jms.LPQMessageListener interface is provided to get the message transmission result through the listener.

```
package jeus.jms;

public interface LPQForwardListener {
    /**
     * Event initiated when the message transfer is complete
     * @param message Transferred message
     */
    public void onComplete(Message message);

    /**
     * Event initiated when an exception occurs during message transfer
     * @param message Message being transferred
     * @param e Exception that occurred
     */
    public void onException(Message message, Exception e);

    /**
     * Event initiated when message transfer fails
     *
     * @param message Message that failed to be transferred
     * @param cause Cause of the failure
     */
    public void onFailure(Message message, Throwable cause);
}
```

You can retrieve the message transmission result by setting the implemented LPQMessageListener to the JeusSession object by using the following API.

```
public void setLPQMessageListener(jeus.jms.LPQMessageListener lpqMessageListener);
```

When using a JEUS object to activate LPQ, you can use the following API to both activate LPQ and set the listener.

```
public void startLPQ(jeus.jms.LPQMessageListener lpqMessageListener);
```

6.7.4. LPQ Configuration

LPQ Configurations include items for processing failed transmissions, operation during disconnection, and the storage size or location.

LPQ Configuration Items

The following describes LPQ configuration items.

- Common Items

Item	Type	Description
jeus.lpq.name	String	LPQ name. (Default: JEUS_LPQ)
jeus.lpq.max-message-count	int	Maximum number of messages that LPQ can process at a time. (Default: 819200)
jeus.lpq.time-to-live	long	How long a message can remain in LPQ. (Default: 43200000ms (12 hours))

- Transmission-Related Items

Item	Type	Description
jeus.lpq.retry-limit	int	Number of attempts to resend a failed message. If set to 0 or less, retries are attempted indefinitely. (Default: -1)
jeus.lpq.retry-interval	long	Transmission interval for resending a message. (Default: 1000 ms)
jeus.lpq.retry-interval-increment	long	Increment to increase the retry interval every time a message is resent. (Default: 0 ms)

- Reconnection-related Items

Item	Type	Description
jeus.lpq.reconnect-retry-interval	long	Interval for reconnecting when the connection is lost. (Default: 5000 ms)

- Storage-related Items

Item	Type	Description
jeus.lpq.store.store-mode	int	Storage type. Set to 1 to use the journal store and 2 to use the memory storage. (Default: 1)

Item	Type	Description
jeus.lpq.store.journal. store-base-dir	String	Name of the directory to create the Store in. The name must be unique per LPQ configuration and two or more concurrent connections are not allowed. If set to a relative path, it is set to a path under the directory where the configuration file exists or JVM is running. (Default: JEUS_LPQ_STORE)
jeus.lpq.store.journal. initial-log-file-count	int	Number of log files to create initially when creating the Journal Store. (Default: 2)
jeus.lpq.store.journal. max-log-file-count	int	Maximum number of log files to create. (Default: 10)
jeus.lpq.store.journal. log-file-size	String	Log file size. (Default: 64 M) Append one of the following text after an integer value or number. • 'K'(KiloBytes) • 'M'(MegaBytes) • 'G'(GigaBytes)

How to Configure LPQ

The following describes how to configure LPQ. The order of configuration is runtime property configuration, the configuration file, and then the JVM option.

- **Configuration File**

  When LPQ configuration file exists in the specified location, the configurations in the file are applied when LPQ is activated. Find the configuration file in order of DEPLOYED_HOME/myApp/WEB-INF/, DEPLOYED_HOME/myApp/META-INF/, and DEPLOYED_HOME/myApp/. The default value of the detailed configuration file path is 'jeus-lpq.properties' and can be changed using the following option.

  ```
  -Djeus.jms.client.lpq-configuration-path=jeus-lpq.properties
  ```

  The following is an example of the LPQ configuration file. The name of this example file is 'jeus-lpq.properties', and the file exists in 'JEUS_HOME/templates/lpq/'.

  ```
  #JEUS Local-Persistent-Queue Configuration
  #[commons]
  jeus.lpq.name=JEUS_LPQ
  ```

```
jeus.lpq.max-message-count=819200
jeus.lpq.time-to-live=43200000

#[forward]
jeus.lpq.retry-limit=-1
jeus.lpq.retry-interval=1000
jeus.lpq.retry-interval-increment=0

#[reconnect]
jeus.lpq.reconnect-retry-interval=5000

#[store]
jeus.lpq.store.store-mode=1

#[journal-store]
jeus.lpq.store.journal.store-base-dir=JEUS_LPQ
jeus.lpq.store.journal.max-log-file-count=10
jeus.lpq.store.journal.initial-log-file-count=2
jeus.lpq.store.journal.log-file-size=64M
```

- **System Property**

  The previous configuration items can be set as system properties by using each item name as the key. For instance, if the LPQ name is configured, the JVM option can be configured when a JEUS MQ client is executed as in the following.

  ```
  -Djeus.lpq.name=<jeus lpq name>
  ```

  Runtime Configurations can be modified by changing the client source code.

  ```
  System.setProperty("jeus.lpq.name", <jeus lpq name>);
  ```

Appendix A: Additional Journal Store Properties

This appendix describes additional journal store properties. These advanced properties can be used to resolve performance issues or functional differences between platforms.

- jeus.store.journal.control-file-name

Description	Save a control file under a different name. This attribute is only used under special circumstances.
Default	control.dat

- jeus.store.journal.log-file-mode

Description	File mode used to open a log file. Set one of "rw", "rws", and "rwd". If set to "rw", file force is forcibly executed.
Default	rwd

- jeus.store.journal.max-move-count

Description	If a log file becomes full, an overflow processing executes to secure the required space. During the overflow processing, the records that have not been used for a long time will be moved to a secondary storage to prevent performance degradation. This property is used to set the allowed maximum number of times the surplus records can be moved.
Default	1

- jeus.store.journal.overflow-factor

Description	If the current log file’s unused space ratio is the same or less than the specified rate, overflow is checked.
Default	0.5

- jeus.store.journal.min-buffer-size

Description	Minimum buffer size for writing records in batch.
Default	4 KB

- jeus.store.journal.max-buffer-size

Description	Largest buffer size for writing records in batch.
Default	4 MB

- jeus.store.journal.use-direct-buffer

Description	Option to use DirectByteBuffer when writing through the FileChannel.
Default	false

- jeus.store.journal.max-waiting-thread-count

Description	Maximum number of threads that are waiting to use the buffer.
Default	32

Appendix B: JDBC Persistence Store Columns

This appendix describes the columns of the tables created when the Persistence store is set to JDBC in JEUS MQ.

B.1. Destination Table

The following table contains information about destinations.

Item	Type	Description
DT_ID	BIGINT	Destination ID
DT_NAME	VARCHAR(255)	Destination name
DT_QUEUE	BIT	Indicates whether the destination is a Queue or Topic. ◦ Queue : true ◦ Topic : false
DT_VALID	BIT	Indicates whether the destination is valid.
DT_LVID	BIGINT	Current version of the destination
DT_DYNAMIC	BIT	Indicates whether the destination is created dynamically or not
DT_OBJECT	BLOB	Binary data of the destination

B.2. Durable Subscription Table

The following table contains information about durable subscription.

Item	Type	Description
DS_ID	BIGINT	Durable subscription ID
DS_CLIENT_ID	VARCHAR(255)	Client ID assigned to the durable subscription
DS_NAME	VARCHAR(255)	Durable subscription name
DS_SELECTOR	VARCHAR(255)	Message selector assigned to the durable subscription
DS_VALID	BIT	Indicates whether the durable subscription is valid
DS_LVID	BIGINT	Current version of the durable subscription
DT_ID	BIGINT	ID of the topic linked to the durable subscription
DT_LVID	BIGINT	Version of the topic linked to the durable subscription

B.3. Message Table

The following table contains information about messages.

Item	Type	Description
MG_ID	BIGINT	Message ID
MG_TYPE	TINYINT	Message type
MG_LENGTH	INTEGER	Message length
MG_OBJECT	TINYINT[]	Binary data of the message
MG_STATUS	SMALLINT	Message status
MG_GLOBAL_ORDER_CLOCK	SMALLINT	Time when the message’s Global Order was set
MG_PERSISTENT	TINYINT[]	Indicates whether a message is set as persistent or not.
DT_ID	BIGINT	ID of the message’s destination
DT_LVID	BIT	Version of the message’s destination
MG_HEADER_LENGTH	INTEGER	Message header length
MG_HEADER_OBJECT	TINYINT[]	Binary data of the message header

B.4. MetaInfo Table

The following table contains information about JEUS MQ’s persistence store.

Item	Type	Description
SERVER_NAME	VARCHAR(255)	JEUS MQ server name
VERSION	BIGINT	JEUS MQ version

B.5. Subscription Message Table

The following table contains information about durable subscription messages.

Item	Type	Description
DM_ID	BIGINT	Durable subscription message ID
DM_STATUS	SMALLINT	Status of the durable subscription message
DM_LVID	BIGINT	Current version of the durable subscription message
MG_ID	BIGINT	ID of the actual message
DS_ID	BIGINT	ID of a Durable subscription containing durable subscription messages

B.6. Transaction Table

The following table contains information about transactions.

Item	Type	Description
TR_ID	BIGINT	Transaction ID
TR_STATUS	TINYINT	Transaction status
TR_OBJECT	TINYINT[]	Binary data of the transaction