Ultra Messaging JMS Guide

Table of Contents

1. Introduction

The **Ultra Messaging®** JMS API lets you develop or port Java messaging applications written per the JMS (Java Message Servicea) specification and still utilize much of the flexibility and performance benefits of **Ultra Messaging**. Ultra Messaging JMS is included with the **Ultra Messaging UMQ** edition.

This document describes how Ultra Messaging JMS integrates JMS applications with **Ultra Messaging**, and requires that you have a background in JMS and **Ultra Messaging** concepts. Please refer to the following:

- Ultra Messaging Concepts (../Design/index.html)
- Java Message Service (JMS) API Specification, version 1.0.2b (http://download.oracle.com/otn-pub/jcp/7543-jms-1.0.2b-spec-oth-JSpec/jms-1_0_2b-spec.pdf)
- Java Message Service (JMS) API Specification, version 1.1 (http://download.oracle.com/otn-pub/jcp/7195-jms-1.1-fr-spec-oth-JSpec/jms-1_1-fr-spec.pdf)
- Java Message Service Tutorial (http://download.oracle.com/javaee/1.3/jms/tutorial/)

The following specific issues regarding JMS compliance are not fully supported in the 5.3.6 (or earlier) release of Ultra Messaging JMS.

- The `recover()` method (for session recovery) is not implemented.
- Though Ultra Messaging JMS with **UMQ** uses a push model, not a pull model, for delivering messages from a queue to a receiver, Ultra Messaging JMS is designed to emulate the pull model. This behavior is transparent in most, but not all, applications.
- Current **UM** message property names do not follow the requirement for provider-supplied property naming.
- The `TopicSubscriber NoLocal` attribute is not implemented.
- `JMSMessageID` is not passed over the wire from sender to receiver. This is replaced with a **UM** `JMSMessageID` and it does not occur as a message property in the message itself. This impacts applications that rely upon this message property to select or process a message.

The following **UM** features are currently not fully supported in the 5.3.6 (or earlier) release of Ultra Messaging JMS. For information on **UM** feature support, see *Interoperability*.

- Using wildcard and non-wildcard receivers simultaneously
- Ultra Messaging JMS across the **UM** Gateway
- **UMQ** Ultra Load Balancing

Informatica is aware of the following issues:

- You cannot have duplicate (same topic) subscribers on the same connection (or queue session ID).
- You cannot unsubscribe durable subscribers (`Unsubscribe()` API) during receiver creation.
- Advisory messages such as beginning or end of transport are not implemented.
- Fault tolerance for process-level transaction handling
- Zero length messages between Ultra Messaging JMS and **Ultra Messaging Desktop Services**
- Ultra Messaging JMS JAR installation not compatible with Microsoft® Windows® due to a new dependency on `libeay32.dll`. As a workaround, set `use_native_loader=false` and load the dependencies from the PATH.
- **UM** native clients interoperating with JMS clients require significant effort to decode MapMessages.
- Transactions are not resilient to application or messaging component failures that occur during a transaction.
- Durable topic subscribers may receive duplicate messages when publishers go down, due to UMP Proxy Source behavior.
- Publishers cannot send to a queue and a topic with the same name using the same JMS connection.
- Messages are not ordered across publishers on the same topic (not required per spec, but often expected).

2. Ultra Messaging JMS Overview

This section discusses the following topics.

- *The JMS Specification*
- *Publish/Subscribe Model*
- *Point-To-Point Model*
- *Quality of Service*
- *JMS Messages*
- *JMS Implementation*
- *Ultra Messaging JMS Programming Architecture*
- *Interoperability*
- *Unsupported JMS Specifications*

2.1. The JMS Specification

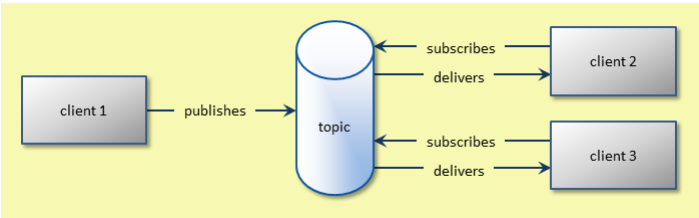
The Oracle JMS Specification 1.0.2b (http://download.oracle.com/otn-pub/jcp/7543-jms-1.0.2b-spec-oth-JSpec/jms-1_0_2b-spec.pdf) provides requirements and guidelines for developing JMS-based Java messaging applications. The specification provides for two models: Publish/Subscribe, and Point-To-Point. Both models are supported by Ultra Messaging JMS.

Ultra Messaging JMS also supports Oracle JMS Specification 1.1 (http://download.oracle.com/otn-pub/jcp/7195-jms-1.1-fr-spec-oth-JSpec/jms-1_1-fr-spec.pdf), which unifies the class hierarchies of the Point-To-Point and Pub/Sub domains, but is fully backward compatible.

2.2. Publish/Subscribe Model

With the JMS publish/subscribe model, a JMS user (or *client*) publishes messages to a topic. Other clients then subscribe to the topic and are thus able to receive the published messages. The JMS model supports the concept of topic being an unadministered object (defined simply by its name), which directly correlates to the **Ultra Messaging** model. You can configure Ultra Messaging JMS to use either the streaming or persistence features.

Figure 1. JMS Publish/Subscribe

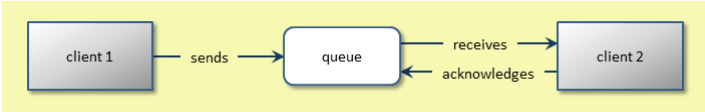


Note: The maximum topic length is 246 bytes.

2.3. Point-To-Point Model

The Point-To-Point model differs from the Publish/Subscribe model mainly in that it employs a message queue and that the sending and receiving clients are aware of each other. Receiving clients extract messages from the queue and notifies it that the messages have been consumed. The queue retains messages until they are consumed or time out. Ultra Messaging JMS employs **UM**'s queue provided by **UMQ**.

Figure 2. JMS Point To Point



Note: Ultra Messaging JMS supports a pull-based (or polling-based) implementation of this model, where the receiving application requests messages from the queue (as opposed to the queue automatically pushing messages). However, the underlying queue, as provided by UMQ, uses a push model.

2.4. JMS Messages

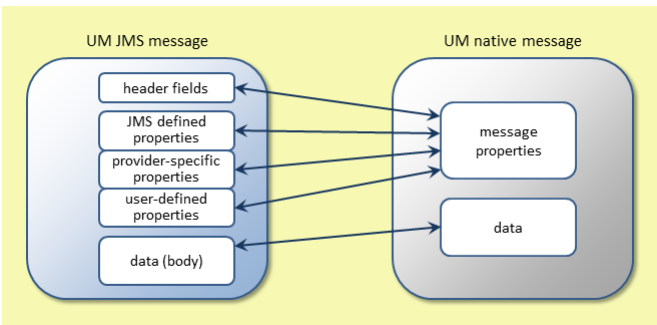
The JMS message generally consists of a header and body (data payload). You can set a JMS body type programatically and optionally identify the body type in a header field. JMS Messages can be of the types shown below. (Numeric values are used in the **UM** message property `LBMMessageType`.)

- TextMessage (0)
- BytesMessage (1)
- MapMessage (2)
- Message (3) (this message type has no body)
- ObjectMessage (4)
- StreamMessage (5)

2.4.1. Message Components

When a JMS message passes through **UM** layers, its message properties are preserved as **UM** message properties (../Design/lbm-objects.html#MESSAGEPROPERTIESOBJECT). Also, header fields are translated into additional message properties.

Figure 3. Message Structure



A JMS message header consists of the following fields. In the **UM** layers, their information becomes **UM** message properties. Note that there is not always a one-to-one correlation between **UM** message properties and JMS properties/header fields.

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

2.4.2. Message Properties

Message properties are defined by unique names and can be assigned values. **UM** supports all JMS message properties, which come in three categories.

JMS defined properties - The JMS Specification (http://download.oracle.com/otn-pub/jcp/7195-jms-1.1-fr-spec-oth-JSpec/jms-1_1-fr-spec.pdf) defines these properties (with prefix "JMSX") and reserves the use of their names.

Provider-specific properties - These are properties defined and reserved for **UM**, and include:

- LBMMessageType (JMS message body types)
- JMSTopicType (string, UMS/UMP/UMQ)
- JMSReplyToName (string, topic name)
- JMSReplyToWildcard (boolean)
- JMSReplyToType (string, UMS/UMP/UMQ)

User properties - These are properties that you defined for your applications. A typical use for these is as *Message Selectors*.

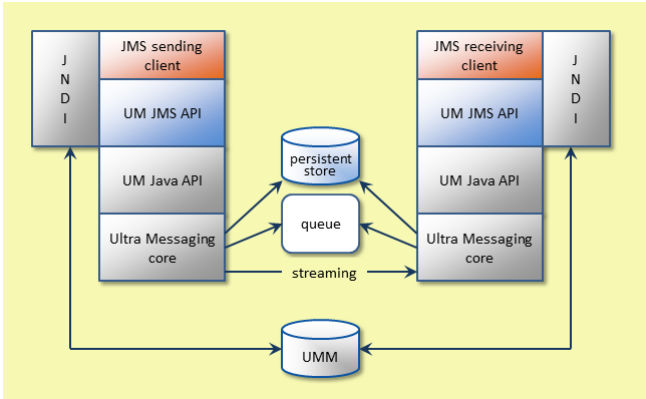
2.5. JMS Implementation

The Ultra Messaging JMS API serves as a wrapper, allowing JMS clients access to **UM** functionality. This section describes in more detail the relationship between the Ultra Messaging JMS and **UM** layers.

2.5.1. JMS Architecture

The following diagram shows how Ultra Messaging JMS relates to the **UM** core middleware. When creating connections and sessions, you typically use JNDI (Java Naming and Directory Interface) to look up administered objects in **Ultra Messaging Manager** (**UMM**).

Figure 4. Ultra Messaging JMS Architecture

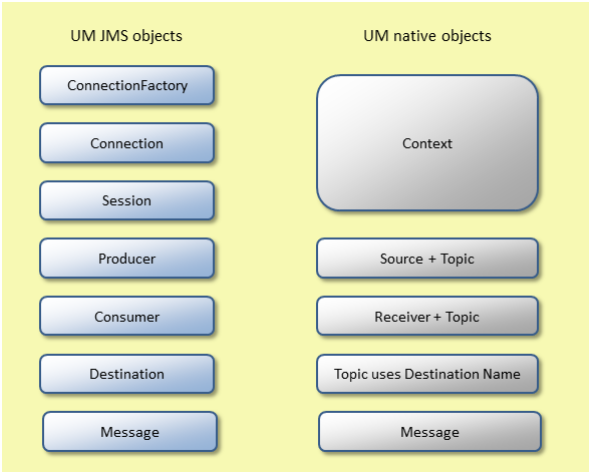


The store (**UMP**) can be used with Publish/Subscribe applications, and the **UMQ** queue is needed for use with Point-To-Point applications.

2.5.2. JMS-To-UM Object Mapping

From the JMS API layer to the Java API layer, there is a functional mapping of objects, as shown in the following diagram.

Figure 5. JMS-To-UM Mapping



ConnectionFactory - One of the two objects that are administered (the other being Destination). A client uses the ConnectionFactory to create a connection with a provider.

Connection - A connection to the provider can be either a queue connection or a topic connection, and creates session objects.

Session - The session is the factory for producing producers and consumers. The ConnectionFactory, Connection, and Session combine to functionally map to a **UM** context, though you can reuse a context for multiple ConnectionFactories/connections.

Producer - The producer maps directly to a **UM** source.

Consumer - The consumer maps directly to a **UM** receiver. The JMS concept of a durable consumer and persistent delivery employ **UMP** receivers and **UMP** persistent stores.

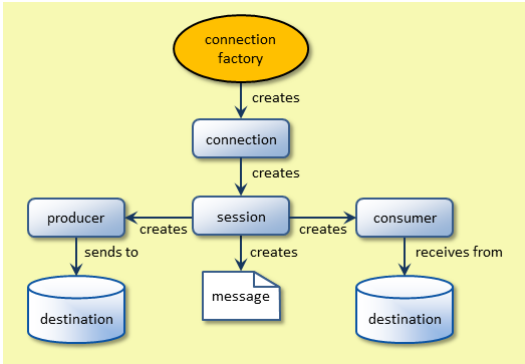
Destination - A client uses a destination to specify the target of messages it produces and/or the source of messages it consumes.

Message - The JMS and **UM** message are variations on each other, with the primary difference being that JMS message header fields are message properties in the **UM** message.

2.6. Ultra Messaging JMS Programming Architecture

The general JMS programming model is shown in the following figure. *Life Cycle of an Ultra Messaging JMS Application* offers details for developing producer and consumer applications.

Figure 6. JMS Programming Model



2.7. Quality of Service

Ultra Messaging JMS uses **UMP** and **UMQ** to provide the desired QoS based on the type of destination configured in JMSConfig.xml.

`MessageProducer`

- Persistent - The `send` method, if using a `Destination` configured as a **UMP**-type destination, blocks until the producer receives a Stability Acknowledgement.
- Non-Persistent - The `send` method does not block if using a `Destination` configured as a **UMS** or **UMQ** type.

`MessageConsumer`

- Consumer - Only receives messages while the application is active.
- Durable Consumer - Receives all messages for the topic if the MessageConsumer was created with a **UMP** type destination, including messages sent while the durable consumer application was not running.

2.8. Interoperability

In general, it is possible for JMS producers to send messages to **UM** receivers, or **UM** sources to send to JMS consumers. This is typically successful with default settings and message body types of TextMessage or BytesMessage. Such interoperability scenarios are also possible with non-default JMS or **UM** settings if you carefully select and test compatible configurations.

A JMS application can communicate with a **UM** application on a limited basis, with the proper attention paid to mapping with JMS headers and message properties. A **UM** application can receive and process the data of a JMS message the same way as when it receives standard **UM** messages.

2.8.1. Native Source

For a **UM** application sending to a JMS application, we recommend you send message data in byte or text format to avoid the risk of unrecognized data formatting. At the JMS application, set the `DEFAULT_MESSAGE_TYPE` attribute in *FactoryAttributes* to match this format. If you need to use more than one message type, at the **UM** source use the `LBMMessageType` property to manage this on a per-message basis.

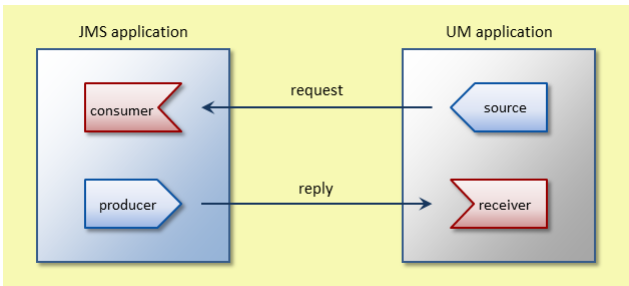
2.8.2. Native Receiver

Native receiver applications can receive messages from a JMS producer. These messages contain all JMS message properties and JMS header information within their own **UM** message properties objects.

2.8.3. Request/Reply Example

The JMS Request/Reply feature employs the `JMSReplyTo` and `JMSCorrelationID` header fields to ensure that the correct receiver/consumer receives a reply and knows which request it pertains to. Consider the Request/Reply case where a native application issues a request to a JMS client (see the figure below).

Figure 7. Interoperation Example



In this scenario, the **UM** application sends a request that the JMS client receives, and the JMS application responds by sending a reply. The following sequence of events provides more detail about this scenario. Note that code excerpts are from a C-sharp **UM** application and a Java JMS application, and example classes may not be part of any UM API.

1. The **UM** application creates a source, which sends a message on topic AAA. The application has set several message properties to identify the topic type, message type, who to reply to, and a unique message identifier.

```
LBMSourceSendExInfo exinfo = new LBMSourceSendExInfo();

/* We want JMS to reply via the JMSReplyTo */
LBMMessageProperties props = new LBMMessageProperties();
props.set("JMSReplyToName", replyTopicString);
props.set("JMSReplyToWildcard", false);
props.set("JMSReplyToType", "LBM");
props.set("JMSTopicType", "topic");
props.set("LBMMessageType", 0); // Indicates to JMS that the message payload will
exinfo.setMessageProperties(props);
exinfo.setFlags(LBM.SRC_SEND_EX_FLAG_PROPERTIES);

/* Added sequence number info flag to print out the expected JMSCorrelationID */
exinfo.setFlags(exinfo.flags() | LBM.SRC_SEND_EX_FLAG_SEQUENCE_NUMBER_INFO | LBM.

src.send(message, msglen, block ? 0 : LBM.SRC_NONBLOCK, exinfo);
```

2. The **UM** application also creates a receiver for the anticipated reply.

```
private static string replyTopicString = "REPLY_" + Guid.NewGuid().ToString();
```

   The key here is that the source sends `replyTopicString` as a message property to provide a "return address". The application then creates a receiver that listens on this topic for the reply.

```
SampleJMSRequestReceiver rcv = new SampleJMSRequestReceiver(verbose);
LBMReceiverAttributes rattr = new LBMReceiverAttributes();
LBMTopic replyTopic = ctx.lookupTopic(replyTopicString, rattr);

LBMReceiver lbmrcv = new LBMReceiver(ctx, replyTopic, rcv.onReceive, null);
```

3. The network passes the message to the JMS application, which converts message properties to JMS header field values and JMS properties, and confirms message type.
4. The JMS application has a consumer that is listening on AAA. It receives the message.
5. Using the JMS header fields, the JMS application's producer sends its reply message to the receiver, getting its reply destination extracted from the `JMSReplyTo` field. The native application's **UM** receiver already listening on this topic then receives the reply.

```
public void onMessage(Message message) {
    try {
        TextMessage requestMessage = (TextMessage) message;
        String contents = requestMessage.getText();
        System.out.println("Got Message: " + contents);
        // get the reply to destination
```

```
            Destination replyDestination = requestMessage.getJMSReplyTo();

            TextMessage replyMessage = session.createTextMessage();
            contents = "Re:" + contents;
            replyMessage.setText(contents);

            replyMessage.setJMSCorrelationID(requestMessage.getJMSMessageID());
            System.out.println("Sending reply of: " + contents);

            replyProducer.send(replyDestination, replyMessage);
        } catch (Exception e) {
            System.err.println("Exception occurred: " + e.getMessage());
            System.exit(-1);
        }
    }
```

6. The network again passes the reply message to the **UM** application, which converts JMS header field values and JMS properties to **UM** message properties usable by the receiver. The **UM** application uses the `JMSCorrelationID` to match the reply to its original request.

This example conveniently illustrates both native source and native receiver scenarios. Note that the key to successful interoperation is the correct exchange of information between **UM** message properties and JMS headers/properties.

2.8.4. Native Application Notes

When a native source or receiver accesses message property `JMSDeliveryMode`, the property must be an integer type, with a value of 1 (NON_PERSISTENT) or 2 (PERSISTENT). For example (in C language):

```
int delivery_mode = 1;
lbm_msg_properties_set(properties, "JMSDeliveryMode", &delivery_mode, LBM_MSG_PROPERTY_INT,
sizeof(int));
```

2.8.5. Compatibility With Other UM Features

Please note that while most **UM** features are compatible with Ultra Messaging JMS, some are not. Following is a table of features and their compatibilities with Ultra Messaging JMS.

UM Feature	UM JMS	Notes
Acceleration - DBL	No	
Acceleration - UD	No	
Hot Failover (HF)	No	
Hot Failover Across Contexts (HFX)	No	
Late Join	Yes	
Message Batching	Yes	
Monitoring/Statistics	Yes	
Multicast Immediate Messaging (MIM)	No	

UM Feature	UM JMS	Notes
Multi-Transport Threads	No	
Off-Transport Recovery (OTR)	Yes	
Ordered Delivery	Yes	
Pre-Defined Messaging (PDM)	Yes	Must use BytesMessage
Request/Response	No	
Self-Describing Messaging (SDM)	Yes	Must use BytesMessage
Source Side Filtering	No	
Transport LBT-IPC	Yes	
Transport LBT-RDMA	Yes	
Transport LBT-RM	Yes	
Transport LBT-RU	Yes	
Transport TCP	Yes	
Transport TCP-LB	Yes	
UM Gateway	No	
UM Spectrum	No	
Wildcard Receivers	Yes	Cannot use wildcard and non-wildcard receivers simultaneously
Zero Object Delivery (ZOD)	No	
UMP Implicit/Explicit Acknowledgements	Yes	
UMP Persistent Store	Yes	
UMP Proxy Sources	Yes	
UMP Quorum Consensus	Yes	
UMP Receiver-Paced Persistence (RPP)	Yes	
UMP Registration ID/Session Management	Yes	
UMP Round Robin	Yes	Not recommended
UMP Store Failover	Yes	Via umestored configuration
UMQ Application Sets	Yes	
UMQ Parallel Queue Dissemination (PQD)	Yes	
UMQ Queue Browser	Yes	
UMQ Queue Failover	Yes	Via umestored configuration
UMQ Queue Redundancy	Yes	Via umestored configuration
UMQ Registration ID/Session Management	Yes	
UMQ Serial Queue Dissemination (SQD)	Yes	
UMQ Source Dissemination (SD)	Yes	
UMQ Ultra Load Balancing (ULB)	No	Not inhibited
Ultra Messaging Desktop Services (UMDS)	No	
Ultra Messaging Manager (UMM)	Yes	
UM SNMP Agent	Yes	

UM Feature	UM JMS	Notes
UMCache	No	

2.9. Unsupported JMS Specifications

The Ultra Messaging JMS does not currently support the following JMS specifications.

- Message prioritization
- JMS Application Server Facilities
- synchronous message consumption/delivery (partial support; see *Point-To-Point Model*)
- `recover()` method (for session recovery)
- pull model for delivering messages from a queue to a receiver
- provider-supplied property naming requirement
- `TopicSubscriber NoLocal` attribute
- `JMSMessageID` passed over the wire from sender to receiver

3. Life Cycle of an Ultra Messaging JMS Application

Using the Ultra Messaging JMS API, you can write end-to-end messaging applications with the programming model shown in *Ultra Messaging JMS Programming Architecture*. This section discusses the following topics.

- *Producer Application*
- *Consumer Application*

3.1. Producer Application

Producer applications take the following actions.

1. Look-up `ConnectionFactory`. The Ultra Messaging JMS implementation supports JNDI lookup of a `ConnectionFactory`.
2. Create a physical connection. The `ConnectionFactory` creates the `Connection`.
3. Create logical session(s) of a `Connection`. Sessions are light-weight connections that can multiplex over a single physical `Connection`. Sessions provide the following.
   a. Concurrent use of the physical `Connection` across multiple threads (one session per thread) and thus are resource efficient.
   b. Delineation of work between multiple producers.
   c. A factory of producers.

4. Create a `Destination`. You can create the destination programmatically or via JNDI lookup. The destination consists of a Topic or Queue name.
5. Create a producer of a session and provide the destination.
6. Create Messages. For each Message, set the business data and set the associated routing properties.
7. Send Messages to the `Destination` address using the producer.
8. Close Objects by destroying the Message, `Destination`(s), producer, session(s), and `Connection`.

The following simple JMS producer example application demonstrates how to program the above actions.

```
// Obtain a ConnectionFactory via lookup or direct instantiation
ConnectionFactory factory = (ConnectionFactory)jndiContext.lookup("uJMSConnectionFactory");

// Create a connection - assuming no username/password required for UM
Connection connection = factory.createConnection();

// Create a session
Session session = connection.createSession(false, javax.jms.Session.AUTO_ACKNOWLEDGE);

// Create a topic destination
Destination destination = session.createTopic("TOPIC.1");

// Create a producer
MessageProducer msgProducer = session.createProducer(null);

// create a bytes message
BytesMessage msg = session.createBytesMessage();

byte[] buffer = {1,2,3,4,5};

msg.writeBytes(buffer);

// Publish the message
msgProducer.send(destination, msg);

// close the connection
connection.close();
```

3.2. Consumer Application

Consumer applications normally take the following actions.

1. Look up `ConnectionFactory`. The Ultra Messaging JMS implementation supports JNDI lookup of a `ConnectionFactory`.
2. Create a physical connection. The `ConnectionFactory` creates the `Connection`.

3. Create logical session(s) of a `Connection`. Sessions are light-weight connections that can multiplex over a single physical `Connection`. Sessions provide the following.
   a. Concurrent use of the physical `Connection` across multiple threads (one session per thread) and thus are resource efficient
   b. Delineation of work between multiple consumers.
   c. A factory of consumers.
4. Create a `Destination`. You can create the destination, which consists of a Topic/Queue name, programmatically or via JNDI/**UMM** lookup. The **UMM** repository contains a Wildcard designation for a `Destination`.
5. Create a consumer of a session, providing the `Destination` created. If the messages are read asynchronously, register the listener call-back function and exception with the consumer.
6. Read Messages from the `Destination` Address. For each destination address (absolute or wildcard), the read could be synchronous, the consumer could register a listener call-back function to asynchronously receive messages, or the consumer could also register a listener call-back exception function to asynchronously receive exceptions, business or technical.
7. Process messages to access data and properties.
8. Close Objects by destroying Message; `Destination`(s), consumer, session(s), `Connection`.

The following simple JMS consumer example application demonstrates how to program the above actions.

```
// Obtain a ConnectionFactory via lookup or direct instantiation
 ConnectionFactory factory = (ConnectionFactory)jndiContext.lookup("uJMSConnectionFactory");

// Create a connection - assuming no username/password required for UM
Connection connection = factory.createConnection();

// Create a session
Session session = connection.createSession(false, javax.jms.Session.AUTO_ACKNOWLEDGE);

// Create a topic destination
Destination destination = session.createTopic("TOPIC.1");

// create the consumer
MessageConsumer msgConsumer = session.createConsumer(destination);

// start the connection
connection.start();

// read messages
while(true)
{
 // receive the message
  msg = msgConsumer.receive();
 if (msg == null)
  break;
}
```

```
// close the connection
connection.close();
```

4. JNDI Administered Objects

JMS administered objects encapsulate specific behavior so you can write producer and consumer applications without the use of specific constructions. Your applications can use JNDI to look up `LBMConnectionFactory` and `Destination` objects. The `LBMConnectionFactory` encapsulates all of the **UM** configuration properties for the selected JNDI object and also creates and initializes the **UM** Context.

4.1. Look Up Administered Objects Using JNDI

The code below shows the typical steps to create an initial context and look up a `ConnectionFactory` and a `Destination` using JNDI. The JNDI repository used depends on the values used for `providerContextFactory` and `providerUrl` and would typically need to specify a `userName` and `password`.

```
// Import of Sun's JMS interface
import javax.jms.*;

// Imports for JNDI
import javax.naming.*;
import javax.naming.directory.*;

// Create the Context
InitialContext jndiContext = new InitialContext();
// Lookup the objects
ConnectionFactory cf = (ConnectionFactory)jndiContext.lookup("uJMSConnectionFactory");
Destination d = (Destination)jndiContext.lookup("TOPIC1");
```

5. Ultra Messaging JMS Configuration

You configure options/attributes for Ultra Messaging JMS object creation in one of three general ways:

- Using a UM configuration XML file
- Using UMM
- Using a JMSConfig file

This section discusses the following topics:

- *jndi.properties*
- *Configuring Ultra Messaging JMS with a UM XML File*

- *Configuring Ultra Messaging JMS with Ultra Messaging Manager*
- *Configuring Ultra Messaging JMS with a JMSConfig XML File*

5.1. jndi.properties

The `/jmsclient/bin/jndi.properties` governs from where your JMS applications receive their configuration data. Regardless of which of the three configuration methods you decide to use, you must edit or ensure that this file matches the selected method. The `jndi.properties`, in its default form, appears below. Note that the file specifies the third configuration method (JMSConfig XML file).

```
## Use the LBM XML based context factory
#java.naming.factory.initial = com.latencybusters.jms.LBMXmlContextFactory
## where the xml config file is one of
#java.naming.provider.url = classpath:umjms.xml
#java.naming.provider.url = file:C:/umjms.xml
#java.naming.provider.url = file:/home/user1/umjms.xml

## Use the UMM based context factory
#java.naming.factory.initial = com.latencybusters.jms.UMMContextFactory
## where the ummd is running at the following url
#java.naming.provider.url = localhost:15701
#java.naming.security.principal = JMSUser
#java.naming.security.credentials = JMSUser

## Use Sun's RefFSContextFactory (with .bindings file)
java.naming.factory.initial = com.sun.jndi.fscontext.RefFSContextFactory
## where the .bindings file is
java.naming.provider.url = file:.
```

The next three sections describe how to edit the appropriate sections of this file.

5.2. Configuring Ultra Messaging JMS with a UM XML File

You can create an XML configuration file using an XML editor or by using the **UMM** GUI. See Using the UMM GUI (../UMM/umm-gui.html) for details. With this method, the UMM Daemon doesn't have to be running when your Ultra Messaging JMS applications start.

To use this configuration method, edit the `jndi.properties` file as follows:

1. Make sure that all lines in the second and third section are commented out.
2. Un-comment the line, `java.naming.factory.initial = com.latencybusters.jms.LBMXmlContextFactory`
3. Un-comment one of the `java.naming.provider.url` lines and supply the path and filename of your XML configuration file. See also *Configuring Ultra Messaging JMS with Ultra Messaging Manager*.
4. Save the `jndi.properties` file.

The **UM** XML configuration file has the following high-level structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<um-configuration version="1.0">

<applications>
 <application name="uJMS">

  <contexts>
    <context name="uJMSConnectionFactory" template="">
      <sources/>
      <receivers/>
      <wildcard-receivers/>
      <options type="context">
    </context>
    <context name="uJMSConnectionFactory-UMS">
      <sources/>
       <receivers/>
       <wildcard-receivers/>
    </context>
    <context name="uJMSConnectionFactory-UMP">
      <sources/>
      <receivers/>
      <wildcard-receivers/>
    </context>
    <context name="uJMSConnectionFactory-UMQ">
      <sources/>
      <receivers/>
      <wildcard-receivers/>
    </context>
  </contexts>

  <event-queues>
    <event-queue/>
  </event-queues>

  <application-data>
    <ConnectionFactory name="uJMSConnectionFactory">
      <options type="ConnectionFactory">
      </options>
    </ConnectionFactory>
    <Destination name="TempQueue">
      <options type="Destination">
      </options>
    </Destination>
  </application-data>

 </application>
</applications>
</um-configuration>
```

Essentially, the structure has two parts. The first part contains the contexts that allow you to specify traditional **UM** configuration option values. The second part, `<application-data>`, allows you to specify JMS options to factories and destinations. The following details some of the major sections of the above high-level structure.

- `<context name="uJMSConnectionFactory" template="">` - Accepts general **UM** configuration option values for the contexts, sources, and receivers used by your applications.
- `<context name="uJMSConnectionFactory-UMS">` - Accepts option values specific to any Streaming activities of your applications.
- `<context name="uJMSConnectionFactory-UMP">` - Accepts option values specific to Persistent activities, such as store configuration options.
- `<context name="uJMSConnectionFactory-UMQ">` - Accepts option values specific to Queueing activities, such as queue configuration options.
- `<ConnectionFactory name="uJMSConnectionFactory">` - Accepts JMS Factory option values such as `default_message_type` or `default_topic_type`.
- `<Destination name="TempQueue">` - Accepts JMS destination option values such as `type` or `dest_type` .

To see a more complete XML file, see `/jmsclient/config/umjms.xml`. Unlike the high-level structure above, this file contains relevant option values. You can also review the same information in the **UMM** GUI. The XML file you create must adhere to the high-level structure presented above.

5.3. Configuring Ultra Messaging JMS with Ultra Messaging Manager

You can use the **UMM** GUI to create configurations for your JMS applications. With this method, the UMM Daemon must be running to provide server configuration and license information to your JMS applications when they start. To use this method, edit the `jndi.properties` file as follows:

1. Make sure that all lines in the first and third section are commented out.
2. Un-comment the line, `java.naming.factory.initial = com.latencybusters.jms.UMMContextFactory`
3. Un-comment `java.naming.provider.url` and substitute an IP address and port number for `localhost:15701` to specify where the UMM Daemon runs. You can add a comma-separated list of daemons.
4. Save the `jndi.properties` file.

5.4. Configuring Ultra Messaging JMS with a JMSConfig XML File

> **Note:** This is a legacy feature and no longer recommended.

With this method, you use the `/jmsclient/bin/config.bat` or `/jmsclient/bin/config.sh` script to create a `.bindings` file from your `config.xml`. To use this method, edit the `jndi.properties` file as follows:

1. Make sure that all lines in the first and second section are commented out.
2. Un-comment the line, `java.naming.factory.initial = com.sun.jndi.fscontext.RefFSContextFactory`

3. Un-comment `java.naming.provider.url = file:.`
4. Save the `jndi.properties` file.

The JMSConfig.xml file allows you to assign **UM** configuration values to the Ultra Messaging JMS `ConnectionFactory` and specify message topics as destinations. You can create a JMSConfig format XML configuration file using an XML editor or text editor. See `/jmsclient/config/config.xml` for an example of a JMSConfig file. This is the configuration file for the **UM** JMS examples described in Ultra Messaging JMS Quick Start (../QuickStart/jms-binary-quick-start.html).

The configuration file has the following high-level structure.

```
<JMSConfig>
 <ConnectionFactories>
  <ConnectionFactory>
   <FactoryAttributes>
    <Attribute/>
   </FactoryAttributes>
   <ContextAttributes>
    <Attribute "UM configuration options, scope=context" />
   </ContextAttributes>
   <SourcetAttributes>
    <Attribute "UM configuration options, scope=source" />
   </SourcetAttributes>
   <ReceiverAttributes>
    <Attribute "UM configuration options, scope=receiver" />
   </ReceiverAttributes>
   <WildcardRceiverAttributes>
    <Attribute "UM configuration options, scope=wildcard-receiver" />
   </WildcardReceiverAttributes>
  </ConnectionFactory>
 </ConnectionFactories>
 <Destinations>
  <Destination>
   <DestinationAttributes>
    <Attribute/>
   </DestinationAttributes>
   <ReceiverAttributes>
    <Attribute/>
   </RecieverAttributes>
  </Destination>
 </Destinations>
</JMSConfig>
```

This section discusses the following topics.

- *ConnectionFactory Attributes*
- *Destination Attributes*

5.4.1. ConnectionFactory Attributes

You can configure as many ConnectionFactories as needed. The ConnectionFactory element contains the following sets of attributes.

- *FactoryAttributes*
- *ContextAttributes*
- *SourceAttributes*
- *ReceiverAttibutes*

5.4.1.1. FactoryAttributes

See Ultra Messaging JMS Options (../Config/ultramessagingjmsoptions.html) for ConnectionFactory options.

The following is an example ConnectionFactory configuration that uses all the default values.

```
<ConnectionFactories>
 <ConnectionFactory>
  <FactoryAttributes>
   <Attribute name="CLIENT_ID" value="UME1"/>
   <Attribute name="DEBUG" value="false"/>
   <Attribute name="DEFAULT_TOPIC_TYPE" value="UME"/>
   <Attribute name="DEFAULT_TEMP_TOPIC_TYPE" value="LBM"/>
   <Attribute name="USE_APP_HEADER" value="true"/>
   <Attribute name="DEFAULT_MESSAGE_TYPE" value="TextMessage"/>
  </FactoryAttributes>
 </ConnectionFactory>
</ConnectionFactories>
```

Note: The following message methods will not work if set USE_APP_HEADER to false. (`<Attribute name="USE_APP_HEADER" value="false"/>`).

- `getJMSCorrelationID/setJMSCorrelationID`
- `getJMSDeliveryMode/setJMSDeliveryMode`
- `getJMSDestination/setJMSDestination`
- `getJMSExpiration/setJMSExpiration`
- `getJMSMessageID/setJMSMessageID`
- `getJMSPriority/setJMSPriority`

5.4.1.2. ContextAttributes

A ConnectionFactory's Context Attributes consist of any **UM** Configuration Options with the scope of Context. See the **Ultra Messaging** Configuration Guide (../Config/index.html) for all configuration options. The following are examples of ContextAttributes.

```
<ContextAttributes>
 <Attribute name="operational_mode" value="sequential"/>
 <Attribute name="resolver_multicast_ttl" value="16"/>
 <Attribute name="resolver_multicast_address" value="225.72.39.173"/>
 <Attribute name="mim_address" value="225.72.39.174"/>
 <Attribute name="transport_lbtrm_multicast_address_low" value="225.73.39.200"/>
 <Attribute name="transport_lbtrm_multicast_address_high" value="225.73.39.210"/>
 <Attribute name="request_tcp_port_low" value="16000"/>
 <Attribute name="request_tcp_port_high" value="16010"/>
 <Attribute name="transport_lbtrm_source_port_low" value="15000"/>
 <Attribute name="transport_lbtrm_source_port_high" value="15500"/>
 <Attribute name="transport_tcp_maximum_ports" value="20"/>
 <Attribute name="transport_tcp_port_low" value="16500"/>
 <Attribute name="transport_tcp_port_high" value="16600"/>
 <Attribute name="resolver_unicast_port_high" value="45000"/>
 <Attribute name="transport_lbtrm_data_rate_limit" value="500000000"/>
 <Attribute name="transport_lbtrm_retransmit_rate_limit" value="1000000"/>
 <Attribute name="transport_lbtrm_receiver_socket_buffer" value="8000000"/>
 <Attribute name="request_tcp_reuseaddr" value="1"/>
</ContextAttributes>
```

5.4.1.3. SourceAttributes

A ConnectionFactory's Source Attributes consist of any **UM** Configuration Options with the scope of Source. See the **Ultra Messaging** Configuration Guide (../Config/index.html) for all configuration options. The following are examples of SourceAttributes.

```
<SourceAttributes>
 <Attribute name="transport" value="lbtrm"/>
 <Attribute name="late_join" value="1"/>
 <Attribute name="ume_store_name" value="JMSStore1"/>
 <Attribute name="ume_store_name" value="JMSStore2"/>
 <Attribute name="ume_store_name" value="JMSStore3"/>
 <Attribute name="ume_store_behavior" value="qc"/>
 <Attribute name="ume_proxy_source" value="1"/>
 <Attribute name="umq_queue_name" value="JMSQueue"/>
 <Attribute name="implicit_batching_minimum_length" value="1"/>
</SourceAttributes>
```

> **Note:** You can define and add stores to a source's store list by using either the `ume_store` or `ume_store_name` attribute. We suggest the latter (which uses a name instead of an IP/port address), as this facilitates JMS deployment to different machines and/or the UM Gateway.

5.4.1.4. ReceiverAttibutes

A ConnectionFactory's Receiver Attributes consist of any **UM** Configuration Options with the scope of Receiver. See the **Ultra Messaging** Configuration Guide (../Config/index.html) for all configuration options. The following are examples of Receiver Attributes.

```
<ReceiverAttributes>
 <Attribute name="umq_receiver_type_id" value="100"/>
</ReceiverAttributes>
```

5.4.2. Destination Attributes

Destinations correspond to **Ultra Messaging** topics. See Ultra Messaging JMS Options (../Config/ultramessagingjmsoptions.html) for destinations options.

The following is an example destination configuration.

```
<Destination name="ReplyTopic" type="Topic">
 <DestinationAttributes>
         <Attribute name="REGID" value="4400"/>
  <Attribute name="WILDCARD" value="false"/>
  <Attribute name="TYPE" value="LBM"/>
 </DestinationAttributes>
</Destination>
```

6. Asynchronous Message Delivery

You can program asynchronous message delivery using the `MessageListener` class. The application registers a callback handler to receive messages asynchronously.

```
// Obtain a ConnectionFactory via lookup or direct instantiation
ConnectionFactory factory = (ConnectionFactory)jndiContext.lookup("uJMSConnectionFactory");

// Create a connection - assuming no username/password required for UM
Connection connection = factory.createConnection();

// Create a Session
Session session = connection.createSession(false, javax.jms.Session.AUTO_ACKNOWLEDGE);

// set the exception listener callback
connection.setExceptionListener(this);

// Create a topic destination
Destination destination = session.createTopic("TOPIC.1");
```

```
// create the consumer
MessageConsumer msgConsumer = session.createConsumer(destination);

// set the message listener callback
msgConsumer.setMessageListener(this);

// start the connection
connection.start();

// The exception listener
public void onException(JMSException e)
{
 // print the connection exception status
  System.err.println("Exception occurred: "+ e.getMessage());
}

// The message listener callback
public void onMessage(Message msg)
{
 try
 {
  System.err.println("Received message: " + msg);
 }
 catch(Exception e)
 {
   System.err.println("Exception occurred: "+ e.getMessage());
    System.exit(-1);
  }
}
```

7. Message Selectors

This section discusses the following topics.

- *Publish/Subscribe*
- *Point-To-Point*
- *Native Applications*

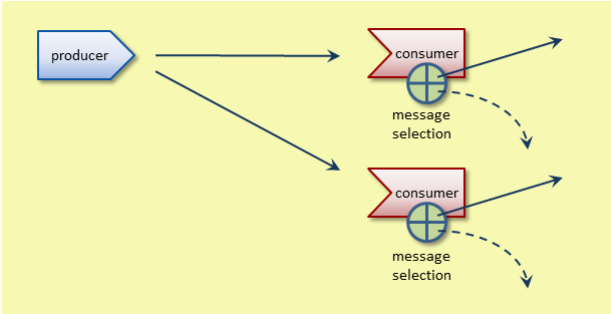
Message Selection provides a way to filter messages at the consumer side utilizing message properties. A message selector is an SQL92-compliant expression that guides a consumer to reject the message if it is not a match.

At the time of consumer creation, an application passes the message selector string to JMS. As messages arrive, the consumer compares their header and properties information to the message selector and rejects messages that evaluate false. For example, a message selector `"MyProp > 5"` allows consumers to receive messages whose JMS message header field `MyProp` value is greater than 5.

7.1. Publish/Subscribe

In the publish/subscribe scenario, you can have many consumers subscribed to the same topic, but use message selectors to filter out selected consumers, and hence, selected receiving clients and applications. In this scenario, each consumer decides whether or not to discard the message.

Figure 8. Message selectors, publish/subscribe



For example, to create a topic subscriber with a message selector for consuming messages with one property greater than 5 and another property equal to 3:

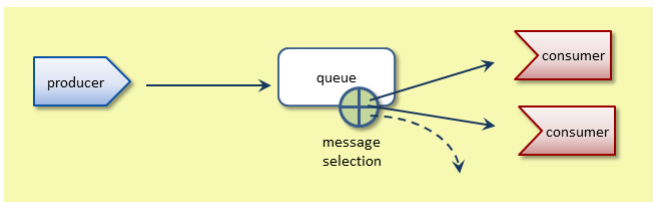
```
// Create a selector to receive only messages with MyProp1 greater than 5 and MyProp2 equal to 3.
String selector = "MyProp1 > 5 AND MyProp2 = 3";

// Create a topic subscriber using the selector.
TopicSubscriber topicSubscriber = topicSession.createSubscriber(queue, selector);
```

7.2. Point-To-Point

In the point-to-point scenario, you can employ message selectors at the queue to determine which of multiple consumers to send messages to. To ensure once-and-only-once delivery, the queue typically ensures that only the consumer assigned per message selector consumes a given message.

Figure 9. Message selectors, point-to-point



For example, to create a queue receiver with a message selector for consuming messages with one property greater than 5 and another property equal to 3:

```
// Create a selector to receive only text messages with MyProp greater than 5.
String selector = "MyProp1 > 5 AND MyProp2 = 3";

// Create a queue receiver using the selector.
QueueReceiver queueReceiver = queueSession.createReceiver(queue, selector);
```

If a consumer is not available for a particular filtered message, the queue skips and retains this message, and then continues processing subsequent messages.

When you configure a queuing application to use application sets, the message selectors can target individual consumers in an application set. Application sets essentially split consumers into logical queues so that the desired once-and-only-once message delivery behavior applies to all consumers inside them. A consumer in one application set can receive the same message as a consumer in another. Message selectors in different application sets let the queue assign messages to selected consumers within the same application set.

7.3. Native Applications

You can use message selectors at the native provider level (i.e., **UM** sources or receivers written in C, Java, or .NET). Message selectors can work in the native application scenarios listed below.

JMS Producer to UM Receiver - Set option `message_selector` (../Config/majoroptions.html#RECEIVERMESSAGESELECTOR) to the desired message selector string when creating the consumer. This string must follow SQL92 syntax as described in the JMS Specification, for example `"MyProp1 = 6 AND MyProp2 < 7"`. See also Ultra Messaging JMS Options (../Config/ultramessagingjmsoptions.html).

UM Source to JMS Consumer - Set message properties so that the name matches that used by the JMS consumer's message selector.

UM Source to UM Receiver - You can use the message selector feature outside of the JMS environment by setting message properties at the **UM** source, and the `message_selector` (../Config/majoroptions.html#RECEIVERMESSAGESELECTOR) option at the **UM** receiver.

> **Note:** For a **UM** receiver, used with **UMP**, and with an event queue or if retaining/promoting messages outside of the receiver callback function, we recommended you enable either explicit ACK'ing (`ume_explicit_ack_only` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMEEXPLICITACKONLY)), or ACK batching (`ume_use_ack_batching` (../Config/ultramessagingpersistenceoptions.html#RECEIVERUMEUSEACKBATCHING)). This prevents undesired/unwarranted ACKs for messages still waiting to be processed.

8. Session IDs

This section discusses the following topics.

- *UMP Session IDs*
- *UMQ Session IDs*

When using **UMP** stores or **UMQ** queues, **UM** objects such as sources, receivers, and/or contexts must register with the stores or queues, at which time they acquire registration IDs. You can use Session IDs to manage these registration IDs.

8.1. UMP Session IDs

A **UMP** Session ID allows stores to identify an application's sources and receivers by automatically assigning them registration IDs. With session IDs, you do not need to directly assign registration IDs. You assign a **UMP** session ID by first setting option `use_ump_session_ids` (../Config/ultramessagingjmsoptions.html#CONNECTIONFACTORYUSEUMPSESSIONIDS) to `True`, then assigning the session ID with the `setClientID()` method, or by setting the `client_id` (../Config/ultramessagingjmsoptions.html#CONNECTIONFACTORYCLIENTID) option. Note that with this option enabled, **UM** ignores attempts to maintain registration IDs directly.

When using this option, we recommend that you call the `setClientID()` function after creating a Connection rather than setting the `client_id` option, to ensure that each created connection has a unique client identifier .

For more information about **UMP** registration IDs and session IDs, see Registration Identifiers (../UME/ume.html#REGISTRATION-IDENTIFIERS), and more specifically, Managing RegIDs with Session IDs (../UME/designing-persistent-applications.html#SESSION-IDS).

8.2. UMQ Session IDs

You can use **UMQ** Session IDs to manage context registration IDs and receiver assignment IDs. For expanded information on **UMQ** Session IDs, see Queue Session IDs (../UME/ume.html#QUEUE-SES-IDS).

In configurations where a queue must always send messages, in order, to a specific assigned receiver, use **UMQ** Session IDs, as described in the next paragraphs. This ensures that if a receiver fails, the queue retains messages for that receiver until it recovers.

To configure this scenario, perform the following steps:

1. Set option `message-reassignment-timeout` (see Options for a Topic's ume-attributes Element (../UME/ume.html#UMESTORED-TOPIC-OPTIONS)) for the queue to a value of 0.
2. Set option `umq_session_id` (../Config/ultramessagingqueuingoptions.html#CONTEXTUMQSESSIONID) to a unique value. Do not replicate this value elsewhere, even for sending applications.

9. Request/Reply Sample Applications

The following example applications demonstrate the request/reply model.

9.1. Request Sample Application

```
// Obtain a ConnectionFactory via lookup or direct instantiation
ConnectionFactory factory = (ConnectionFactory)jndiContext.lookup("uJMSConnectionFactory");


// Create a connection - assuming no username/password required for UM
Connection connection = factory.createConnection();

// Create a Session
Session session = connection.createSession(false,
         javax.jms.Session.AUTO_ACKNOWLEDGE);
// Create request and reply destinations
Destination requestTopic = lookupDestination(requestTopicName);
Destination replyTopic = lookupDestination(replyTopicName);

MessageProducer requestProducer = session.createProducer(requestTopic);
MessageConsumer replyConsumer = session.createConsumer(replyTopic);

TextMessage requestMessage = session.createTextMessage();
requestMessage.setText("Hello world.");
requestMessage.setJMSReplyTo(replyTopic);
requestProducer.send(requestMessage);

// start the connection
connection.start();

// Wait for the reply
Message replyMessage = replyConsumer.receive();

// Could check correlationID, not really necessary with sync request reply
if (replyMessage.getJMSCorrelationID() != requestMessage.getJMSMessageID())
{
 System.err.println("Unexpected reply message"+ e.getMessage());
}
```

9.2. Reply Sample Application

```
// Obtain a ConnectionFactory via lookup or direct instantiation
ConnectionFactory factory = (ConnectionFactory)jndiContext.lookup("uJMSConnectionFactory");


// Create a connection - assuming no username/password required for UM
Connection connection = factory.createConnection();

// Create a Session
Session session = connection.createSession(false,
         javax.jms.Session.AUTO_ACKNOWLEDGE);
// Create request destination
Destination requestTopic = lookupDestination(requestTopicName);
```

```
// Create consumer on request topic
MessageConsumer requestConsumer = session.createConsumer(requestTopic);

// Create a producer, don't know reply destination at this point.
MessageProducer replyProducer = session.createProducer(null);


// set the message listener callback
msgConsumer.setMessageListener(this);

// start the connection
connection.start();


// The message listener callback
public void onMessage(Message message)
{
 try
 {
  TextMessage requestMessage = (TextMessage) message;
  String contents = requestMessage.getText();

  // get the reply to destination
  Destination replyDestination = requestMessage.getJMSReplyTo();

  TextMessage replyMessage = session.createTextMessage();
  replyMessage.setText(contents);

  replyMessage.setJMSCorrelationID(requestMessage.getJMSMessageID());
  replyProducer.send(replyDestination, replyMessage);
 }
 catch(Exception e)
 {
   System.err.println("Exception occurred: "+ e.getMessage());
    System.exit(-1);
  }
}
```